8.1 Mathematical tools for image formation

Image formation is the process of projecting the elements appearing in a **3D scene** (objects, surfaces, landscapes, etc.) on a **2D image plane**. This process happens, for instance, when you are capturing an image of the real world by means of a camera, or when you are playing a first-person videogame and the 3D virtual environment of the game is projected to a plane that is shown to you as the player's view.

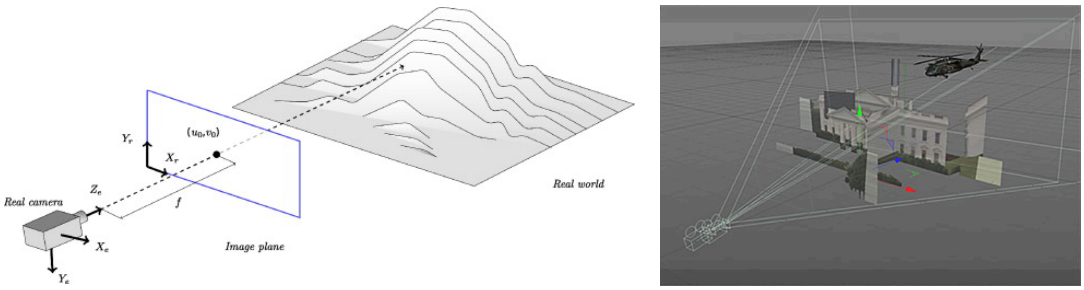


Figure 1. Left, the main actors in image formation in a real setup: the camera, the image plane (we put the (x,y) axes on the top left corner) and the scenario. Right, an example of virtual environment.

The *image formation* topic involves two different perspectives of the problem of generating images from 3D scenes:

- **The geometric problem:** Where does each 3D point project on the image?
- **The radiometric problem:** What will be the color on the image of each 3D point? *(not addressed here)*

In this introductory notebook, we will see some basic mathematical tools that are required for studying image formation models as the common pinhole model. Although it might seem at first that they are not related with the problem at hand, their understanding is of capital importance when studying about elements and transformations between the 3D world and the 2D image plane.

These tools include:

- **Euclidean transformations in 3D**.
  - Introducing and displaying vectors (8.1.1).
  - Products of 3D vectors (8.1.2).
  - Linear transformation of vectors (8.1.3).
  - Rotation matrix (8.1.4).
- ***Next notebook:*** **Homogeneous transformations (8.2)**.

Problem context - Camera in first-person videogames

In video games, it is called *first person* to any graphical perspective rendered from the viewpoint of the player's character, or a viewpoint from the cockpit or front seat of a vehicle driven by the character. Many genres incorporate first-person perspectives, including adventure games, driving, sailing, and flight simulators. Perhaps the more extended are first-person shooters, in which the graphical perspective is an integral component of the gameplay.

Usually, when someone plays a first-person computer game, the `WASD` keys of our keyboard are used to move the camera position in the x and z axes (planar movement) and `SPACE` is used for jumping (this moves the camera in the y axis, hence allowing it to move in 3D). Then, the mouse is used for changing the camera orientation (this applies *pitch* and/or *yaw* rotations to the camera coordinate system). The combination of 3D movement and 3D rotations creates a full 6D motion control for our character!

Figure 2. Example of a virtual sceneario projected on an image.

Our task in this notebook is **to program these camera displacements and rotations so they can be integrated into a graphic engine!**. Obviously, this implies learning all the maths involved in here, and most of them are related to the manipulation of vectors and matrices. So, let's start the fun!

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
import scipy.stats as stats
from ipywidgets import interact, fixed, widgets
from mpl_toolkits.mplot3d import Axes3D
from math import sin, cos, radians
%matplotlib widget
```

```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)
images_path = './images/'
```

3D Euclidean transformations

8.1.1 Introducing and displaying vectors

Since we are going to employ lots of vectors in this notebook (and the following ones), first of all, we need to know how vectors are defined and how to transform them.

We consider two different types of vectors:

- A **bound vector** defined by two points $\{\mathbf{p}, \mathbf{q}\}$:

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} \in \mathbb{R}^3, \ \mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} \in \mathbb{R}^3$$

- A **free vector** $\mathbf{v}$ (or just vector) that can be defined as a mathematical entity represented as an *oriented segment* between two points, whose elements (in 3D) can be computed as:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} q_1 - p_1 \\ q_2 - p_2 \\ q_3 - p_3 \end{bmatrix} \in \mathbb{R}^3$$

This way, different bound vectors can share the same free vector (indeed a free vector is defined as the set of bound vectors having the same module, orientation and direction).

Note from the notation in $\mathbf{p}$, for example, that a vector can also represent the coordinates of a certain point with respect to the origin of coordinates. In fact, in this notebook, we will use vectors to refer to several entities such as points, axes or even transformations (translations).

Thus, it is important to know how to draw them, in order to have a visual reference of what we are using! In python, we can plot a set of 3D vectors using matplotlib's method `quiver()`, which takes 6 main arguments:

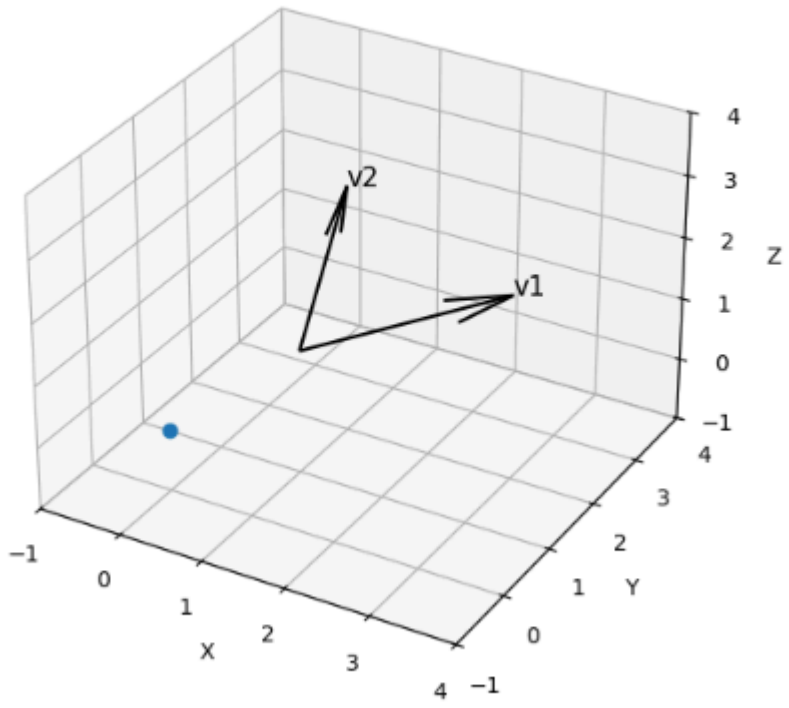
- `X,Y,Z`: tuples containing the *X*, *Y* and *Z* coordinates of the origin point in the set of vectors.
- `U,V,W`: tuples containing the *X*, *Y* and *Z* coordinates of all vectors w.r.t. such origin point.

The next code illustrates how to plot two free vectors $\mathbf{v}_1 = (2, 1, 1)$, and $\mathbf{v}_2 = (0, 1, 2)$, with origin point $\mathbf{p} = (1, 1, 1)$

In [2]:
```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Vector coordinates
```

```
v1 = np.array([2,1,1])
v2 = np.array([0,1,2])
v = np.column_stack((v1,v2))

# Vector labels
labels = ["v1","v2"]

# Origin coordinates
p = np.array([1,1,1])
origin = np.column_stack((p,p))

# Prepare vector for plotting input
X,Y,Z = origin[0,:], origin[1,:], origin[2,:]
U,V,W = v[0,:], v[1,:], v[2,:]

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')

# Set axes limits
ax.set_xlim3d(-1, 4)
ax.set_ylim3d(-1, 4)
ax.set_zlim3d(-1, 4)

# Add axis labels and aspect ratio
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
#ax.set_aspect('equal')

# Plot vectors
ax.quiver(X, Y, Z, U, V, W, color="black")

# Add labels
for i in range(len(labels)):
    ax.text(U[i]+X[i], V[i]+Y[i], W[i]+Z[i] ,labels[i], fontsize=12)

# And show the figure!
plt.plot(0,0,0,'o') # Draws the origin (0,0,0)
plt.show()
```

Figure



ASSIGNMENT 1a: We want to see how these things work

Let's convert the previous code displaying vectors in a method that we can use later. **Your first task is** to define the method called `plot_vectors()`, which returns a figure plotting a number of input free vectors as defined in `v` with a given `origin` and axes `labels`.

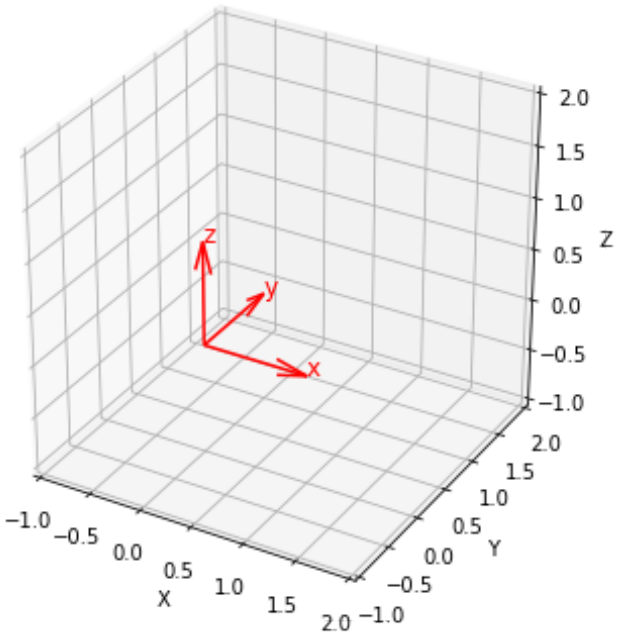
Note that:

- This function also expects as inputs a figure and some axes. This permits us to call the method **multiple times** while working the same figure.
- For that, the first time that the method is called, the axes must be previously initialized outside the method. You can use ax = plt.axes(projection='3d') for that.
- You also have to permit to introduce the axes limits as argument, to select the area of the plot we want to see.

In [3]:
```
# ASSIGNMENT 1
def plot_vectors(fig, ax, v, origin, labels, color, axes_lim):
    """ Plot 3D vectors using matplotlib

        Args:
            fig, ax: Figure and axes (must be 3D)
            v: Array containing vector coordinates, each column contains a 3D ve
            origin: Array containing vector origin points for 'v' array
```

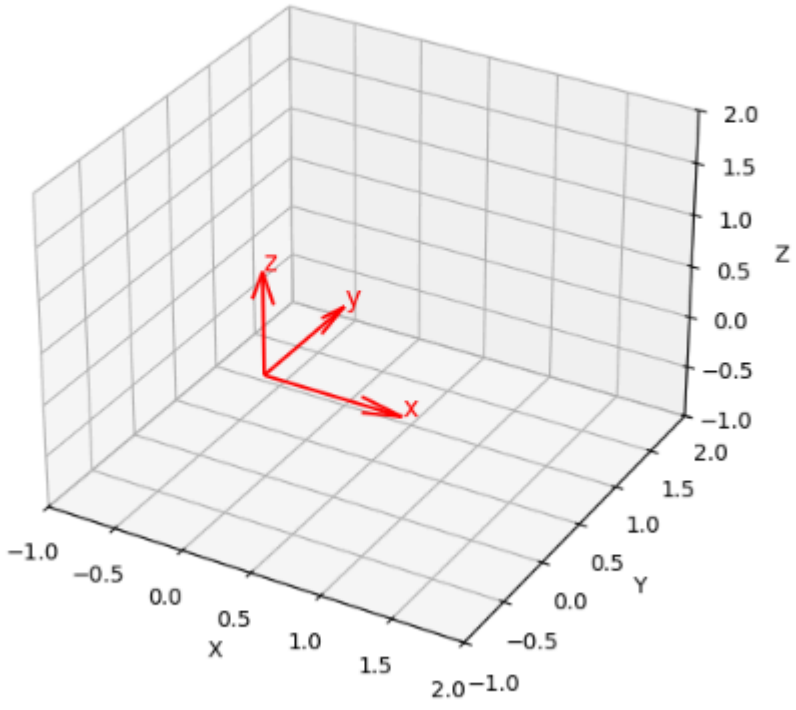
```
            labels: Array of strings containing labels for the vectors, it shoul
            color: String containing the color of the vectors
            axes_lim: 6-size vector containing the minimum and maximum limits fo

        Returns:
            fig, ax: Figure and axis of a 3D plot with the vectors plotted
    """
    # Write your code here

    # Prepare vector for plotting input
    X,Y,Z = origin
    U,V,W = v

    # Set axes limits
    ax.set_xlim3d(axes_lim[0], axes_lim[1])
    ax.set_ylim3d(axes_lim[2], axes_lim[3])
    ax.set_zlim3d(axes_lim[4], axes_lim[5])

    # Add axis labels and aspect ratio
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    #ax.set_aspect('equal')

    # Plot vectors
    ax.quiver(X, Y, Z, U, V, W,color=color)

    # Add labels
    for i in range(len(labels)):
        ax.text(U[i]+X[i], V[i]+Y[i], W[i]+Z[i], labels[i], fontsize=12, color=c

    return fig,ax
```

To **test your function**, the next code should show an orthonormal basis (unit vectors) centered at $\mathbf{p}_0 = (0, 0, 0)$, so the expected output is:



In [4]:
```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')

# Vector coordinates
v = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

# Vector labels
labels = ["x","y","z"]

# Call plot_vectors
fig, ax = plot_vectors(fig, ax, v, origin, labels, "red", [-1,2,-1,2,-1,2])
```

Figure



ASSIGNMENT 1b: Plotting a free vector

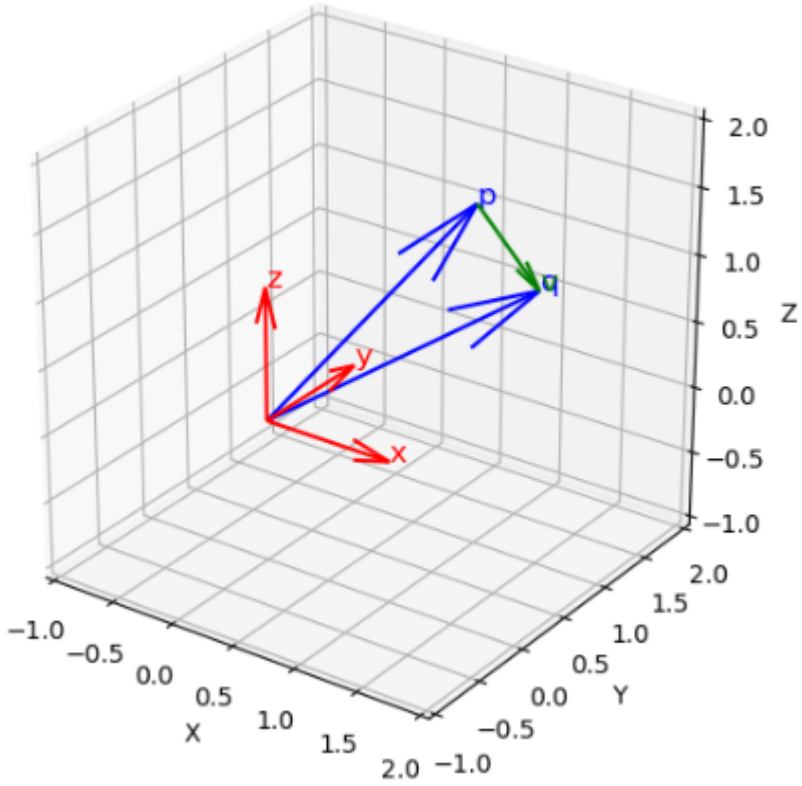
Let's make use of the `plot_vectors()` method for plotting a free vector `v` built from the coordinates of other two vectors `p` and `q`. Recall that:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} q_1 - p_1 \\ q_2 - p_2 \\ q_3 - p_3 \end{bmatrix} \in \mathbb{R}^3$$

As a result, the coordinates of the free vector `v` are expressed w.r.t. the endpoint of the `p` vector, that is, `p` is its origin. Consider the following definitions of vectors:

- $p = [1, 1, 1.5]$
- $q = [1.5, 1, 1]$

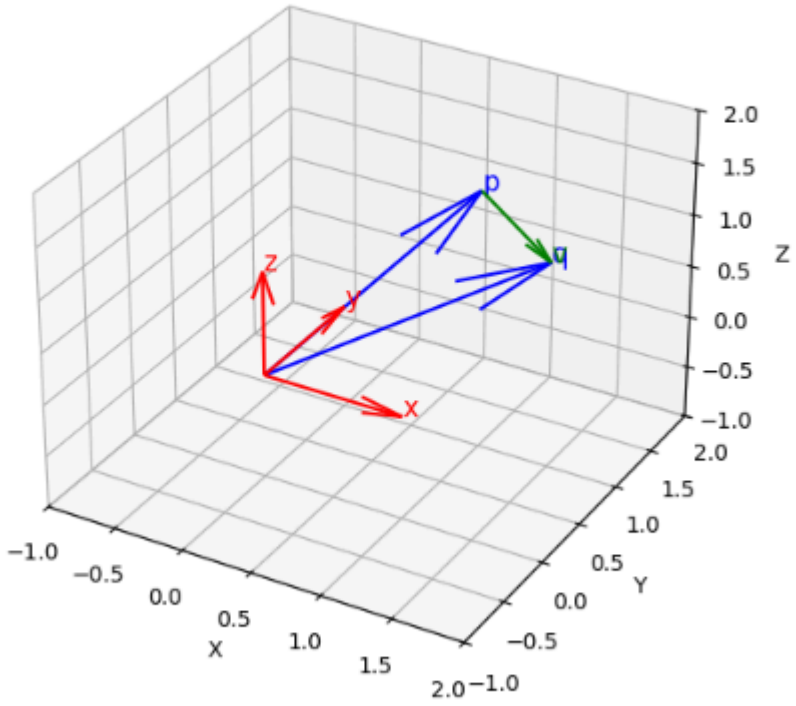
This is the result we are looking for:



In [5]:
```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')

# System of reference vector coordinates
v_o = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates of the reference system
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

# Define the vectors
p = np.vstack([1,1,1.5])
q = np.vstack([1.5,1,1])
v = q - p
```

```
# Define the origin of each vector
origin_p = np.vstack([0,0,0])
origin_q = np.vstack([0,0,0])
origin_v = p

# Vector labels
labels = ["x","y","z"]
labels_p = ['p']
labels_q = ['q']
labels_v = ['v']

# Call plot_vectors
fig, ax = plot_vectors(fig, ax, v_o, origin, labels, "red", [-1,2,-1,2,-1,2])
fig, ax = plot_vectors(fig, ax, p, origin_p, labels_p, "blue", [-1,2,-1,2,-1,2])
fig, ax = plot_vectors(fig, ax, q, origin_q, labels_q, "blue", [-1,2,-1,2,-1,2])
fig, ax = plot_vectors(fig, ax, v, origin_v, labels_v, "green", [-1,2,-1,2,-1,2]
```

Figure



8.1.2 Product of 3D vectors

Now that you know how to define and plot a set of vectors, you are ready to learn two basic operations with them: the **dot** and **cross** products.

Dot product of two vectors

The dot product (also called scalar product or inner product) of two vectors is the sum of the element-wise product between them, which results in a **scalar**.

Algebraic definition:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a} \cdot \mathbf{b} = \mathbf{a}^\mathrm{T}\mathbf{b} = \mathtt{trace}(\mathbf{a}\,\mathbf{b}^\mathrm{T}) = a_1 b_1 + a_2 b_2 + a_3 b_3 \in \mathbb{R}$$

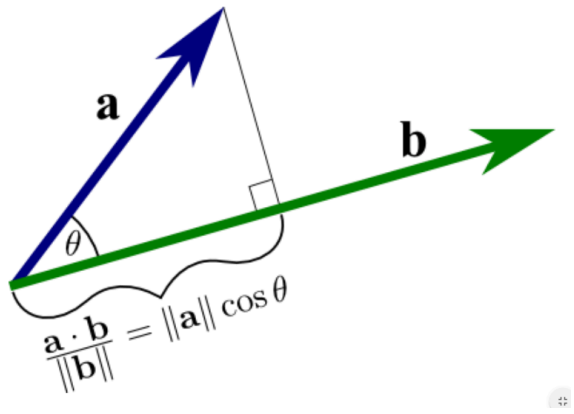
The dot product is involved in the definition of the **Euclidean norm** of a vector, which, geometrically, represents the distance between the two points that delimits the vector:

$$\|\mathbf{a}\| = \sqrt{\mathbf{a}^\mathrm{T}\mathbf{a}} = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

Geometric definition: From a geometric point of view, the dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ is defined by:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\,\|\mathbf{b}\| \cos\theta$$

where θ is the angle between both vectors. That is, the dot product represents the product of their Euclidean norms and the cosine of the angle between them. Such a definition can be rearranged to calculate the projection of one vector onto the other as shown in the figure:



ASSIGNMENT 2: Playing with the dot product

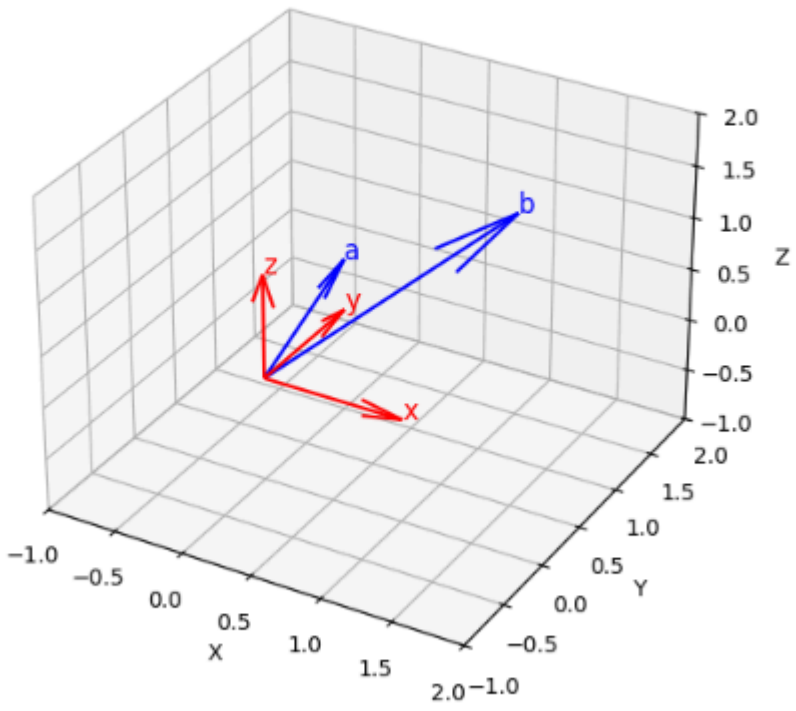
We are going to work with the vectors $\mathbf{a} = (0, 1, 0.5)$ and $\mathbf{b} = (1, 1.5, 1)$ and the dot product:

1. First of all, plot these vectors using `plot_vectors()`.

In [6]:
```
# ASSIGNMENT 2
# Write your code here!

matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')
```

```
# Vector for basis coordinates
v_o = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

# Matrix of vectors
v = np.array([[0,1],[1,1.5],[0.5,1]])
origin_v = np.array([[0,0],[0,0],[0,0]])

# Vector labels
labels = ["x","y","z"]
labels_v = ["a","b"]

# Call plot_vectors
fig, ax = plot_vectors(fig, ax, v_o, origin, labels, "red", [-1,2,-1,2,-1,2])
fig, ax = plot_vectors(fig, ax, v, origin_v, labels_v, "blue", [-1,2,-1,2,-1,2])
```

Figure



2. Now, compute the **dot product** of these vectors using the described **algebraic** method. Check that you get the same result as using `np.dot()`.

In [7]:
```
# Dot product (algebraic)
dot = sum(v[:,0] * v[:,1])
dot_np = np.dot(v[:,0], v[:,1])

print('Dot product     : ' + str(dot))
print('Dot product (np): ' + str(dot_np))
```

```
Dot product     : 2.0
Dot product (np): 2.0
```

Expected output:

```
Dot product     : 2.0
Dot product (np): 2.0
```

3. Finally, let's take a look at the geometric interpretation of the dot product. For that, compute the norms of both vectors, and compute the projection of the first vector `a` onto the second one `b`, that is:

$$\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|}$$

$\[3pt]$ Check that you get the same norm than using the numpy method for computing normals `np.linalg.norm()`.

In [8]:
```
# Norm of second vector
norm_a = sum(v[:,0]**2)**(1/2) # Norm of the first vector
norm_b = sum(v[:,1]**2)**(1/2) # Norm of the second vector
norm_a_np = np.linalg.norm(v[:,0])
norm_b_np = np.linalg.norm(v[:,1])

# Projection of first vector a onto the second one b
projection = dot / norm_b

print("a norm     : " + str(round(norm_a,2)))
print("b norm     : " + str(round(norm_b,2)))
print("a norm (np): " + str(round(norm_a_np,2)))
print("b norm (np): " + str(round(norm_b_np,2)))
print("\nProjection of a onto b: " + str(round(projection,2)))
```

```
a norm     : 1.12
b norm     : 2.06
a norm (np): 1.12
b norm (np): 2.06

Projection of a onto b: 0.97
```

Expected output:

```
a norm     : 1.12
b norm     : 2.06
a norm (np): 1.12
b norm (np): 2.06

Projection of a onto b: 0.97
```

Cross product of two vectors

The cross product of two linearly independent vectors $\mathbf{a}$ and $\mathbf{b}$ (also called vector product) is a linear transformation which results in **another vector $\mathbf{c}$** perpendicular to both, and thus to the plane containing them. Given two vectors:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Their cross product is defined as:

$$\mathbf{c} = \mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2 b_3 \mathbf{i} + a_3 b_1 \mathbf{j} + a_1 b_2 \mathbf{k}) - (a_3 b_2 \mathbf{i} + a_1 b_3 \mathbf{j} + a_2 b_1 \mathbf{k}) = (a_2$$

Although it is more convenient to express it as a linear transformation using matrix multiplication:

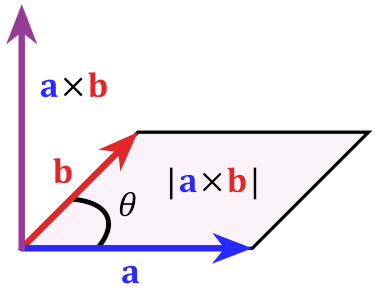
$$\mathbf{c} = \mathbf{a} \times \mathbf{b} = [\mathbf{a}]_\times \mathbf{b} \quad \text{(just a matrix multiplication!)} \qquad / \qquad [\mathbf{a}]_\times = \begin{bmatrix} 0 \\ a_3 \\ -a_2 \end{bmatrix}$$

where $[\mathbf{a}]_\times$ is called the *skew-symmetric* matrix of $\mathbf{a}$.

The norm of the resultant vector is computed as:

$$\|\mathbf{c}\| = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin\theta$$

This has an interesting **geometric meaning**: the norm of the cross product of two vectors $\mathbf{a}$ and $\mathbf{b}$ can be interpreted as the area of the parallelogram that has such vectors as sides. Also, as mentioned before, the resulting vector is orthogonal to both $\mathbf{a}$ and $\mathbf{b}$ vectors:



ASSIGNMENT 3: It's cross product time

Let's play with the cross product! For that **you are tasked to**:

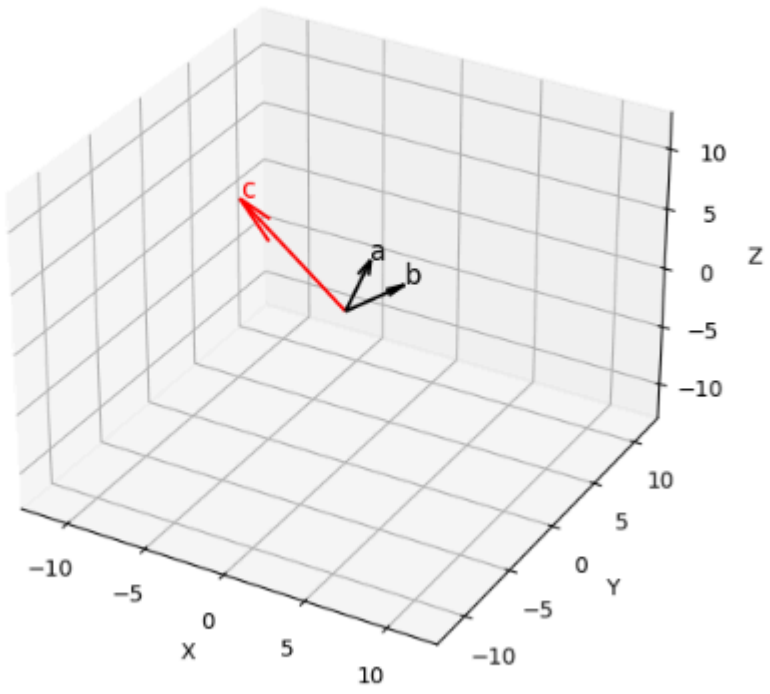
1. Compute the cross product of two vectors $\mathbf{a} = [1, 1, 4]$ and $\mathbf{b} = [2, 3, 1]$ **using matrix multiplication** (that is, using $\hat{\mathbf{a}}$) and check that you get the same result as using `np.cross()`.
2. Compute the norm of the resultant vector $\mathbf{c}$ which, as commented, represents the area of the parallelogram that has $\mathbf{a}$ and $\mathbf{b}$ as sides.
3. Then plot $\mathbf{a}$ and $\mathbf{b}$ as black vectors and their cross product $\mathbf{c}$ in red using your previous method `plot_vectors()`.

Tip: In numPy, matrix multiplication is defined with the `@` operator (`A @ B`) instead of the `*` operator, which performs element-wise matrix multiplication. To transform a horizontal vector (row) to a vertical vector (column) you can use `np.vstack()` .

In [9]:
```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# ASSIGNMENT 3
# Write your code here!

# Vectors coordinates
v = np.array([[1,2],[1,3],[4,1]])

# Vector origin points
origin = np.array([[0,0],[0,0],[0,0]])

# Isolate vectors
a = v[:,0]
b = v[:,1]

# Compute â
hat_a = np.array([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])

# Compute cross product
cross = hat_a @ b
cross_np = np.cross(a,b)

print('a x b     : ' + str(cross))
print('a x b (np): ' + str(cross_np))

# Compute the norm of c = a x b
norm_c = np.linalg.norm(cross)

print('Norm of c : ' + str(round(norm_c,2)))

# Transform to vertical vector
cross = np.vstack(cross)

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')
axes_lim = [-13,13,-13,13,-13,13]

# Call plot_vectors
fig, ax = plot_vectors(fig, ax, v, origin, ["a","b"], 'black', axes_lim)
fig, ax = plot_vectors(fig, ax, cross, np.array([[0],[0],[0]]), ["c"], 'red', ax

plt.show()
```

```
a x b     : [-11   7   1]
a x b (np): [-11   7   1]
Norm of c : 13.08
```

Figure



Expected output:

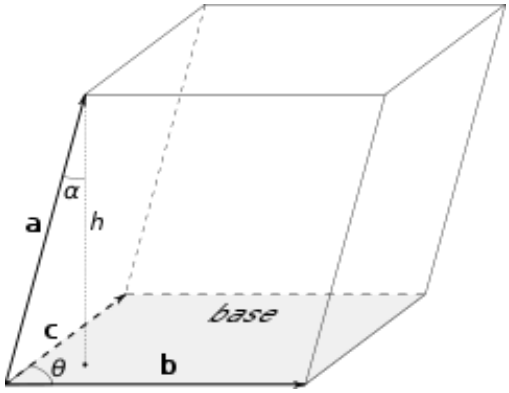
```
a x b     : [-11   7   1]
a x b (np): [-11   7   1]
Norm of c : 13.08
```

Product of three vectors

The product of three vectors (or scalar triple product) is a combination of the two previous ones: it is defined as the dot product of a vector $\mathbf{c}$ with the cross product of the other two, namely $\mathbf{a}$ and $\mathbf{b}$, that is:

$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$$

This product also has a **geometric meaning**: it is the (signed) volume of the parallelepiped defined by those vectors:



ASSIGNMENT 4: Concluding with the triple product

Compute the triple product of $\mathbf{a} = (1, 2, 3)$, $\mathbf{b} = (3, 1, 2)$ and $\mathbf{c} = (2, 2, 1)$:

In [10]:
```
# ASSIGNMENT 4
# Write your code here!

# Vector coordinates
v = np.array([[1,3,2],[2,1,2],[3,2,1]])

#Isolate vectors
a = v[:,0]
b = v[:,1]
c = v[:,2]

# Compute â
hat_a = np.array([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])

# Compute cross product
cross = hat_a @ b

# Compute triple product
triple_product = sum(cross * c)
# triple_product = np.dot(np.cross(a,b),c)

print('Triple product:', triple_product)
```

```
Triple product: 11
```

Expected output:

```
Triple product: 11
```

8.1.3 Linear transformation of vectors

As we saw before, the cross product of two vectors is a linear transformation, i.e., a linear function that transforms the elements of the involved vectors. But the cross product is just one particular case of linear transformation where the matrix that defines such linear function is created from the elements of a vector. A more general definition for linear transformations in the 3D-space would be:

$$\mathbf{Av} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a_1v_1 + a_2v_2 + a_3v_3 \\ a_4v_1 + a_5v_2 + a_6v_3 \\ a_7v_1 + a_8v_2 + a_9v_3 \end{bmatrix}$$

Note that we can stack a set of N **column** vectors $\{\mathbf{v}_i\}$ forming a $3 \times N$ matrix $\mathbf{V}$ so that the linear transformation is applied to every individual vector. The resulting matrix will be a **stack of transformed column vectors**.

$$\mathbf{AV} = \mathbf{A}\begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_N \end{bmatrix} = \begin{bmatrix} \mathbf{Av}_1 & \mathbf{Av}_2 & \cdots & \mathbf{Av}_N \end{bmatrix}$$

This property will be extremelly useful when implementing transformations that must be applied to a large number of points.

But, what means that an operation (in this case a matrix multiplication) is linear? It means that the following two properties hold:

- **Aditivity:** $f(\mathbf{x}_1 + \mathbf{x}_2) = f(\mathbf{x}_1) + f(\mathbf{x}_2) \backslash [5pt]$
- **Scaling:** $f(\alpha\mathbf{x}_1) = \alpha f(\mathbf{x}_1)$

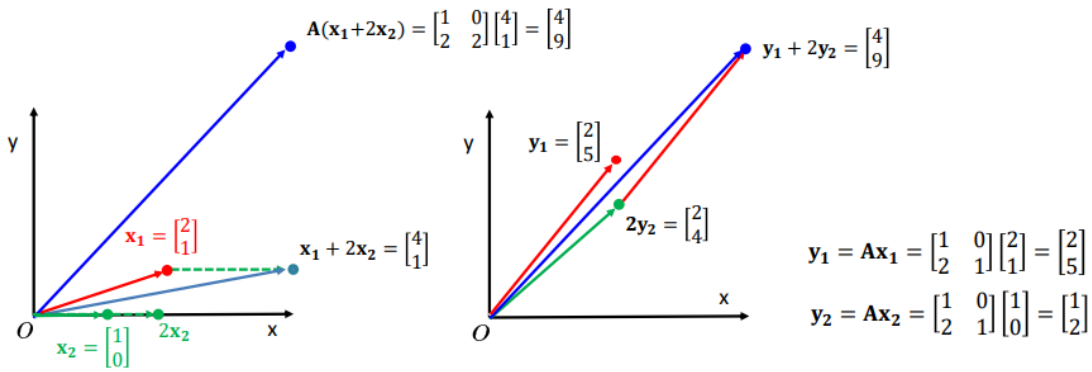
And, equivalently, the superposition principle also holds:

- **Superposition:** $f(\alpha\mathbf{x}_1 + \beta\mathbf{x}_2) = \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$

Luckily, matrix multiplication satisfies this property!

$$\mathbf{A}(\alpha\mathbf{x}_1 + \beta\mathbf{x}_2) = \alpha\mathbf{Ax}_1 + \beta\mathbf{Ax}_2 = \alpha\mathbf{y}_1 + \beta\mathbf{y}_2$$

that is, the transformation of linear combinations of vectors $(\alpha\mathbf{x}_1 + \beta\mathbf{x}_2)$ is the linear combination of the transformed vectors $(\alpha\mathbf{y}_1 + \beta\mathbf{y}_2)$. The following figure shows an example of this property in action, given two vectors $\mathbf{x_1} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ and $\mathbf{x_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ with $\alpha = 1$ and $\beta = 2$, and a matrix $\mathbf{A} == \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$:



ASSIGNMENT 5: Playing with linear transformations

Implement a complete method `apply_transformation()`, which:

1. transforms a set of vectors `v` according to a certain input transformation (matrix $\mathbf{A}$), and then

2. shows the original vectors in black and the transformed ones in red.

Reuse the method `plot_vectors()` for this assignment.

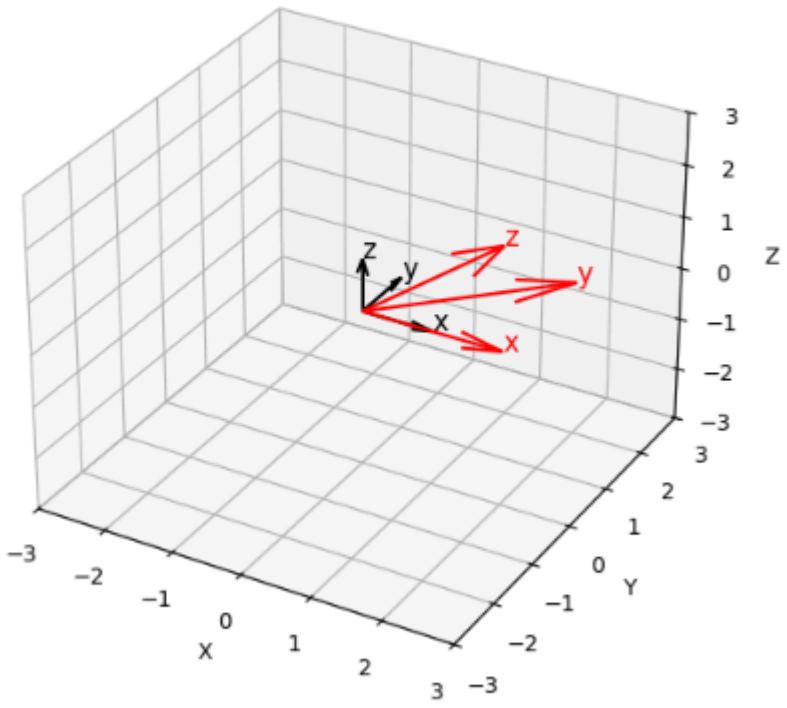
NOTE: The labels for the transformed vectors should be the same that for the originals (but shown in red).

In [11]:
```
# ASSIGNMENT 5
def apply_transformation(transformation, v, origin, labels, axes_lim):
    """ Apply a linear transformation to a set of 3D-vectors and plot them

        Args:
            transformation: 3x3 matrix that defines the linear transformation
            v: Array containing vector coordinates, each column contains a 3D ve
            origin: Array containing vector origin points for 'v' array
            labels: Array of strings containing labels for the vectors, it shoul
            axes_lim: 6-size vector containing the minimum and maximum limits fo
    """
    # Write your code here!

    # Apply transformation
    transformed_v = transformation @ v

    # Create figure
    fig = plt.figure()

    # Prepare figure for 3D data
    ax = plt.axes(projection='3d')

    # Call plot_vectors
    fig, ax = plot_vectors(fig, ax, v, origin, labels, "black", axes_lim)
    fig, ax = plot_vectors(fig, ax, transformed_v, origin, labels, "red", axes_l

    plt.show()
```

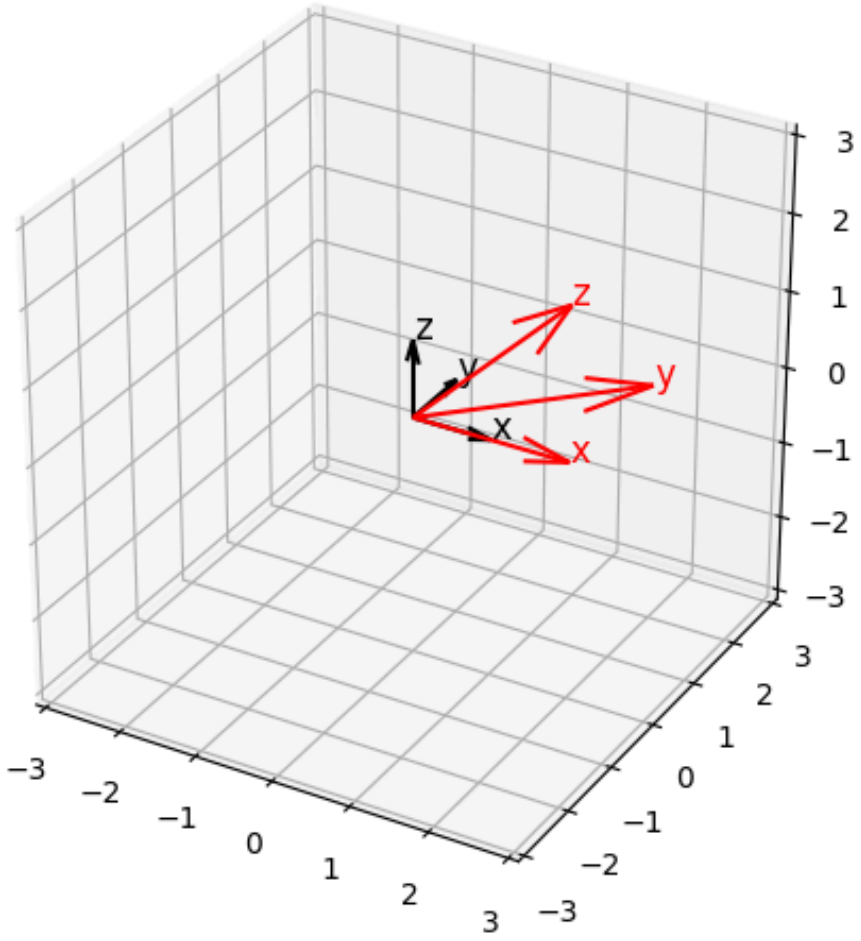
You can use the next code **to test if your results are correct**:

In [12]:
```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Transformation
transformation = np.array([[2,2,2],[0,2,0],[0,0,2]])

# Vector coordinates
v = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

apply_transformation(transformation,v,origin,["x","y","z"],[-3,3,-3,3,-3,3])
```

Figure



Expected output:



8.1.3 Rotation matrix

A rotation matrix represents a special linear transformation that rotates vectors preserving their length.

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{32} & r_{32} & r_{33} \end{bmatrix} = \underbrace{[r_x\ r_y\ r_z]}_{\substack{\text{coordinates of the original} \\ \text{basis in the new one}}} \in \mathbb{R}^{3x3}$$

In this way, let's say that we have a certain vector $\mathbf{p}^W$ that represents the coordinates of a point in a certain reference system that we will call `WORLD` (note the superscript): $\mathbf{p}^W = [p_x, p_y, p_z]^\intercal$. Now, we can rotate it around the origin of coordinates and obtain its new coordinates through:

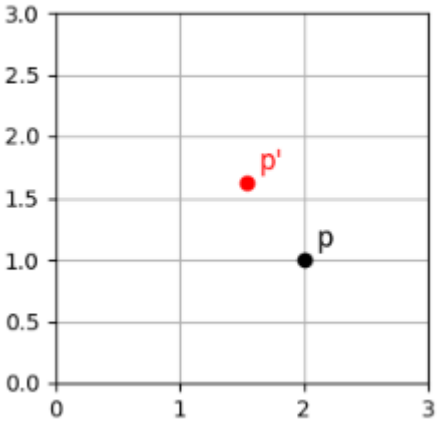
$$\mathbf{p}^{W'} = \mathbf{R}\mathbf{p}^W \quad \rightarrow \quad \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \mathbf{R} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

If we go back to our graphical engine for the videogame, this could correspond to the case of *a point rotating around the camera* (e.g. a bird flying around you), and is called **active rotation**, because it is the point who is rotating around the reference system.

The following code exemplifies one of these rotations (in 2D):

In [13]:
```
p = np.array([2, 1]).T; # defines the point

alpha = radians(20) # 20 degrees
R = np.array([[cos(alpha),-sin(alpha)],[sin(alpha),cos(alpha)]]) # Defines the r

p_prima = R @ p # Applies the rotation

# Show the results!
fig = plt.figure(figsize=(3, 3))
plt.plot(p[0],p[1],'ko')
plt.xlim([0, 3])
plt.ylim([0, 3])
plt.plot(p_prima[0],p_prima[1],'ro')
plt.grid()
plt.text(p[0]+0.1, p[1]+0.1, "p", fontsize=12)
plt.text(p_prima[0]+0.1, p_prima[1]+0.1, "p'", fontsize=12,color="red");
```

Figure



But we can use the rotation matrix not only to rotate points but also to **express rotations between different reference systems**. Imagine now that we have two coordinate systems: a global one, which we will call `WORLD` and a local one, which we will denote by `CAMERA`. And, not only that, the `CAMERA` reference system is rotated w.r.t. the `WORLD` according to a certain rotation matrix $\mathbf{R}_C^W$. *Note: The subscript and superscript in the notation of the $\mathbf{R}_C^W$ matrix indicates that it must be understood as the rotation of the `CAMERA` reference frame as seen from the `WORLD` one.*

Now, imagine we have a certain point $\mathbf{p}^C$ with coordinates in the `CAMERA` reference system $\mathbf{p}^C = [p_x, p_y, p_z]^\mathrm{T}$ and we want to know its coordinates $\mathbf{q}^W = [q_x, q_y, q_z]^\mathrm{T}$ **within** the `WORLD`.

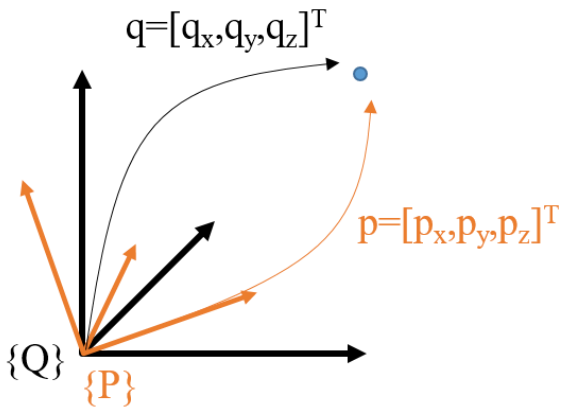


Figure 3

Well, this can be computed with the exact same equation!:

$$\mathbf{q}^W = \mathbf{R}_C^W \mathbf{p}^C \rightarrow \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} = \mathbf{R}_C^W \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

To understand this, let's have a look to the figure above. Note that determining the coordinates of the blue point within the global reference system (in black) is the same that rotating the blue point within the local reference system (in orange) and getting its new coordinates, and that's why they are computed exactly the same way! Doing this we could know the coordinates of the bird in the `WORLD`.

But one step further, imagine that we want now to do exactly the **opposite**, that is, knowing the value of the global coordinates $\mathbf{q}^W$, obtaining the local coordinates $\mathbf{p}^C$ in the rotated reference system. That is, you know the coordinates of the bird in the `WORLD` reference system, but you need them in the `CAMERA` coordinate system. In this case we can use the **inverse of the rotation matrix** to obtain them:

$$\mathbf{p}^C = \left(\mathbf{R}_C^W\right)^{-1} \mathbf{q}^W = \left(\mathbf{R}_C^W\right)^{\mathrm{T}} \mathbf{q}^W = \left(\mathbf{R}_W^C\right) \mathbf{q}^W \rightarrow \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \mathbf{R}^{\mathrm{T}} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

In this case, we can understand this as if the point was **static** in the environment and it is the camera (or the player point-of-view, if you want) which is rotating within it. This is called **passive rotation**.

*Note: It is **really important** to understand that, in these equations, $\mathbf{R}_C^W$ represents the rotation of the `CAMERA` reference system with respect to the `WORLD` reference system while $\mathbf{R}_W^C$ represents the other way around.*

You might also have noticed in this equation that we wrote that $\left(\mathbf{R}_C^W\right)^{-1} = \left(\mathbf{R}_C^W\right)^{\mathrm{T}}$, and that is because rotation matrices are **orthogonal**, hence fulfilling these two properties:

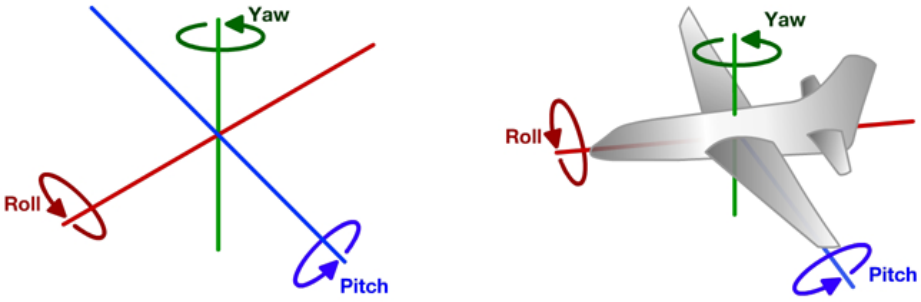
- Its inverse equals its transpose: $\mathbf{R}^{\mathrm{T}}\mathbf{R} = \mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I} \rightarrow \mathbf{R}^{\mathrm{T}} = \mathbf{R}^{-1}$

- $\mathbf{R}$ also verifies that $\det(\mathbf{R}) = +1$

Note: So, remember, if $\mathbf{R}$ is the rotation matrix between systems `A` and `B`, then $\mathbf{R}^{\mathrm{T}}$ is the rotation matrix between system `B` and `A`.

3D rotations

Focusing now on rotations in 3D-space, we can define **three elemental rotations**, one in each axis Z, Y and X, denoted by *yaw*, *pitch* and *roll*, respectively:



These elemental rotations are represented by three different matrices:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix} \qquad \mathbf{R}_x$$

Note: These elemental rotation matrices keep static the transformation in a certain axis, (e.g. pitch rotation does not modify the y values).

Using each one of these matrices rotates a vector in one axis, but **what if we want to perform more than one rotation at a time?** Well, we just need to multiply them in a certain order!

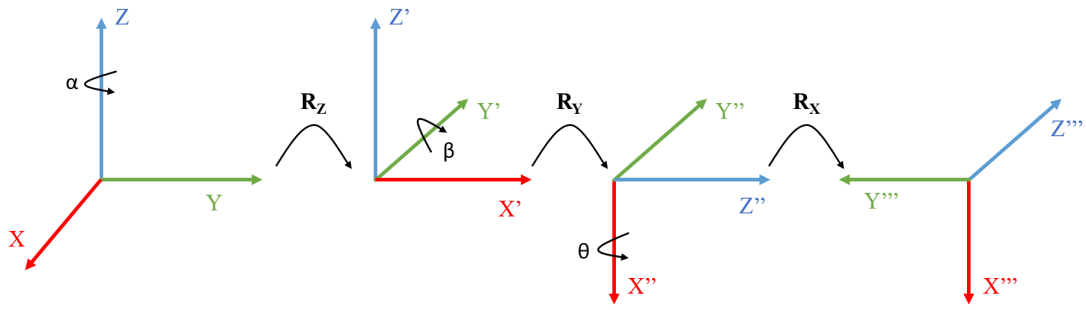
$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \mathbf{R}_z(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_x(\theta)$$

where α = *yaw* angle, β = *pitch* angle and θ = *roll* angle.

But why the order of the multiplication matters? Because matrix multiplication is not commutative! They are several different orders how these matrices can be multiplied but the one presented here (i.e. $Z-Y-X$) is one of the most used (for example in fields like robotics, airborne navigation, etc.).

This sequence means that:

- First, the original reference system is rotated a certain α angle around its Z axis,
- Then the resulting (**rotated**) system is rotated again a certain β angle around its **rotated** Y' axis
- And finally it is rotated again a certain θ angle around its **rotated** X'' axis.



Note that each rotation is applied to the rotated axis (this is called *intrinsic* rotation) and it is performed through **post-multiplication**. If we wanted to rotate around the original (unrotated axis) we would have used pre-multiplication (this is called *extrinsic* rotation). In general this second approach is less intuitive and the first one is preferred. So, just keep the stated order when multiplying elemental rotations.

Wait, but, what about translations?

If we focus on general transformations between reference systems, you can imagine that **usually cameras are not placed at the center of the `WORLD` coordinate system**, but they are moving within the world, that is, they don't share their origin of coordinates. That means that the camera is also translated w.r.t. the `WORLD` reference system, so we have to include a translation vector $\mathbf{t}_C^W$ in order to be able to transform points between the two reference frames, not only rotate them!

Let's go back to Figure 1 and update it with a more general transformation (remember, {Q} is the `WORLD` reference system and {P} is the `CAMERA` one):

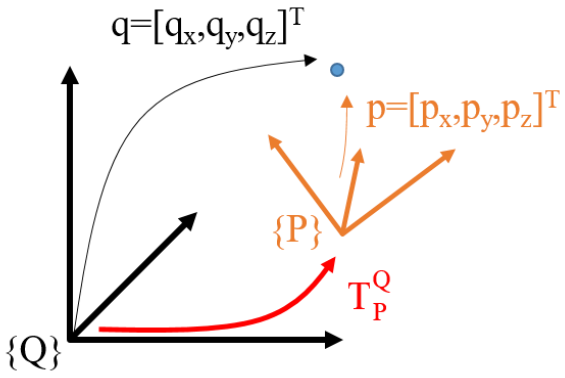


Figure 4

Now that the transformation between the systems **includes a translation**, we have that:

$$\mathbf{q}^W = \mathbf{R}_C^W \mathbf{p}^C + \mathbf{t}_C^W$$

$$\begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Again, in this equation $\mathbf{R}_C^W$ and $\mathbf{t}_C^W$ express the rotation and translation, respectively, of the `CAMERA` reference system w.r.t. the `WORLD` one, that is, the position and orientation of the `CAMERA` within the `WORLD`.

ASSIGNMENT 6: Rotating and translating

Now, we are going to implement the `apply_rotation_translation()` method, which accepts *yaw*, *pitch* and *roll* rotation angles (in degrees) and a translation vector and applies such transformations to an input vector. For that **your are tasked with**:

1. build the $\mathbf{R}_x$, $\mathbf{R}_y$ and $\mathbf{R}_z$ matrices, and combine them into the $\mathbf{R}$ one,
2. apply the rotation to the input vectors,
3. apply the translation to the result of the previous step, and
4. show both, the initial set of vectors and the transformed one.

Tip: you can transform degrees to radians using `radians()`.

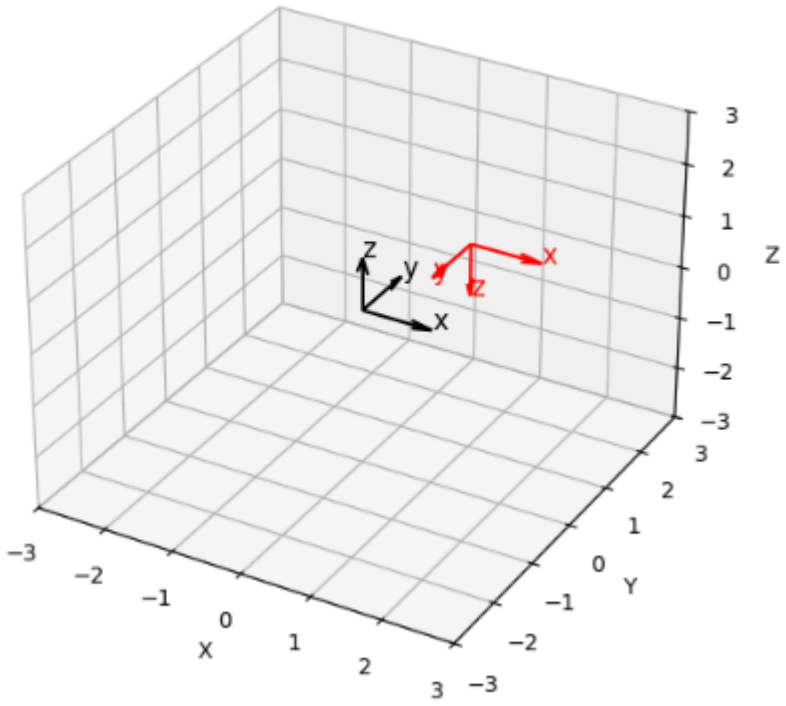
In [14]:
```
# ASSIGNMENT 6
def apply_rotation_translation(v, origin, yaw, pitch, roll, translation, labels,
    """ Apply a linear transformation to a set of 3D-vectors and plot them

        Args:
            v: Array containing vector coordinates, each column contains a 3D ve
            origin: Array containing vector origin points for 'v' array.
            yaw: Degrees to rotate the coordinate system around the 'Z' axis
            pitch: Degrees to rotate the coordinate system around the 'Y' axis
            roll: Degrees to rotate the coordinate system around the 'X' axis
            translation: Column vector containing the translation for each axis
            labels: Array of strings containing labels for the vectors, it shoul
            axes_lim: 6-size vector containing the minimum and maximum limits fo
    """
    # Write your code here!

    # Transform to radians
    yaw = radians(yaw)
    pitch = radians(pitch)
    roll = radians(roll)

    # 1. Construct rotation matrices
    Rx = np.array([[1, 0, 0],[0, cos(roll), -sin(roll)],[0, sin(roll), cos(roll)
    Ry = np.array([[cos(pitch), 0, sin(pitch)],[0, 1, 0],[-sin(pitch), 0, cos(pi
    Rz = np.array([[cos(yaw), -sin(yaw), 0],[sin(yaw), cos(yaw), 0],[0, 0, 1]])

    # Combine rotation matrices
    R = Rz @ Ry @ Rx

    # Apply transformation
    transformed_v = R @ v # apply rotation
    transformed_origin = origin + translation # apply translation

    # Create figure
    fig = plt.figure()

    # Prepare figure for 3D data
    ax = plt.axes(projection='3d')
```

```
    # Call plot_vectors
    fig, ax = plot_vectors(fig, ax, v, origin, labels, "black", axes_lim)
    fig, ax = plot_vectors(fig, ax, transformed_v, transformed_origin, labels, "

    plt.show()
```

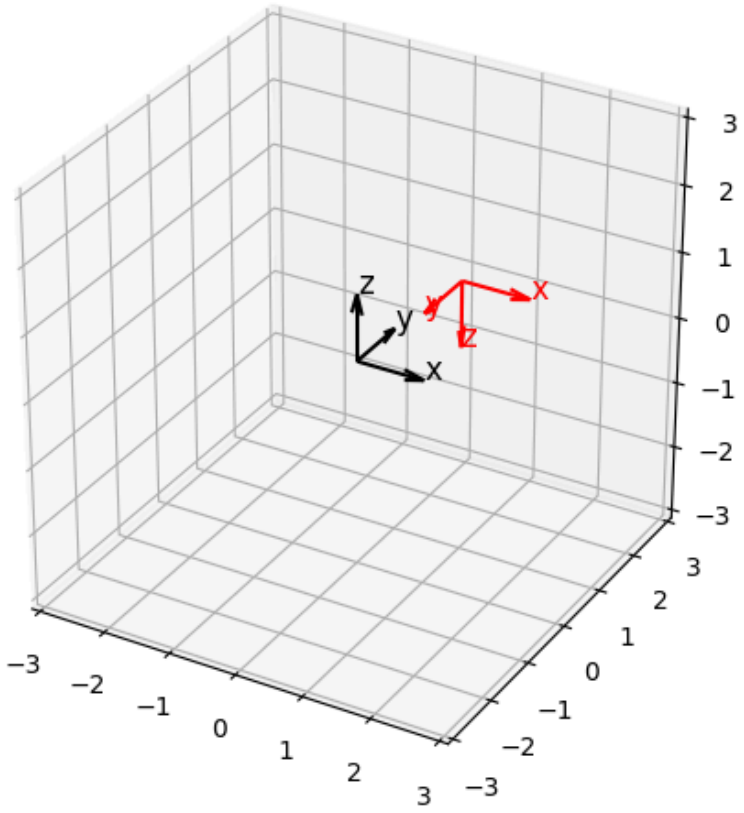
You can use next code **to test if your results are correct**:

In [15]:
```
# Vector coordinates
v = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

# Traslation vector
translation = np.array([[1],[1],[1]])

# Apply transformation
apply_rotation_translation(v,origin,0,0,180,translation,["x","y","z"],[-3,3,-3,3
```

Figure



Expected output:

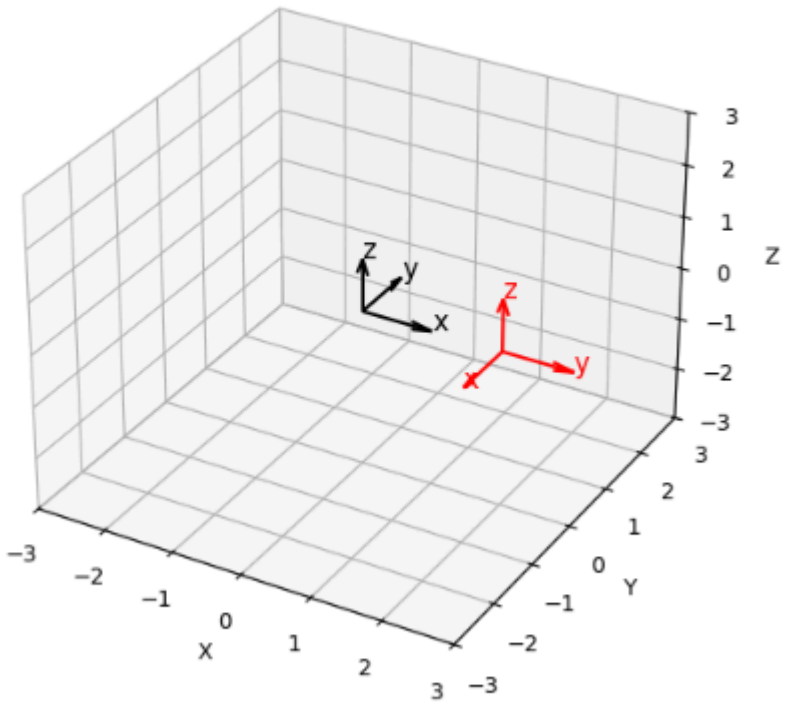


These transformations are strongly related to a camera system in a videogame. For example, it could represent a movement of the player where he/she jumps and rotates. That is, we are moving the `CAMERA` reference system within the `WORLD` one. Next, **you are asked to** simulate the following camera movements in such a context, using the previously defined method.

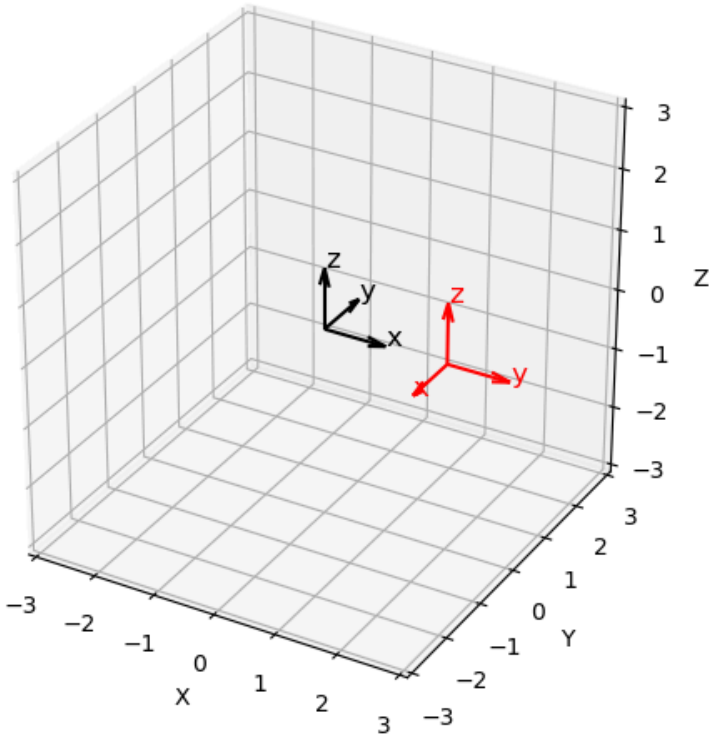
- Look to the **right 90 degrees** and walk **forward 2 units**

In [16]:
```
# Write your code here!

# Vector coordinates
v = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

# Traslation vector
translation = np.array([[2],[0],[0]])

# Apply transformation
apply_rotation_translation(v,origin,-90,0,0,translation,["x","y","z"],[-3,3,-3,3
```

Figure



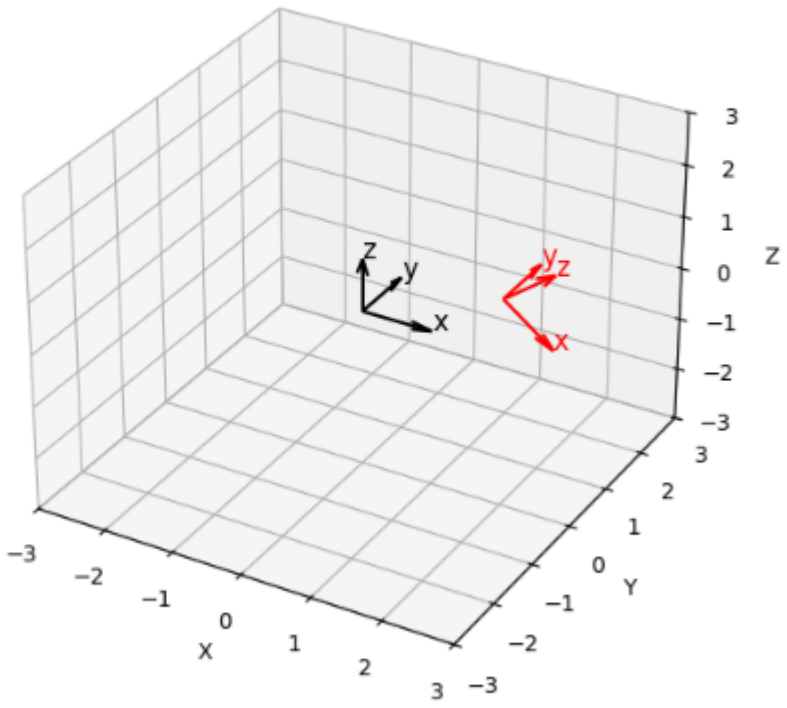
Expected output:



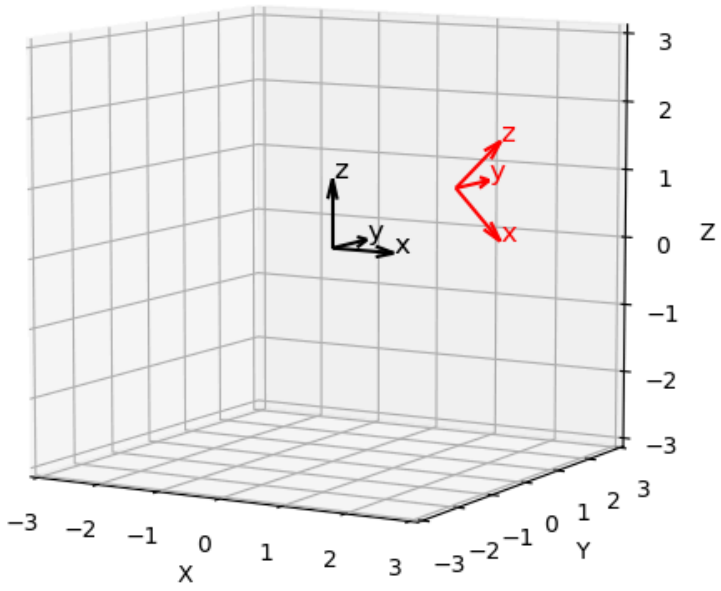
- While you are walking (**two units forward**), jump (our character can **jump one unit of height**), and look **45 degrees to the floor** (e.g. jumping an obstacle).

In [17]:
```
# Write your code here!

# Vector coordinates
v = np.array([[1,0,0],[0,1,0],[0,0,1]])

# Origin coordinates
origin = np.array([[0,0,0],[0,0,0],[0,0,0]])

# Traslation vector
translation = np.array([[2],[0],[1]])

# Apply transformation
apply_rotation_translation(v,origin,0,45,0,translation,["x","y","z"],[-3,3,-3,3,
```

Figure



Expected output:



Conclusion

Awesome!

This was an introductory notebook covering some mathematical tools that are the building blocks for image formation. We made an excellent work trying to master these maths and we have learned:

- How to plot vectors and perform products between them.
- The ideas behind the linear transformation of vectors.
- How to rotate and translate points or vectors using Cartesian coordinates.

8.2 Mathematical tools for image formation. Homogeneous transformations

Homogeneous (also called projective) transformations are linear transformations (i.e. matrix multiplications) **between homogeneous coordinates** (vectors). Such coordinates are obtained from Cartesian (inhomogeneous) vectors by **extending them with a non-negative number** (typically 1, for convenience).

Although we are going to explain homogeneous transformations using the 3D space, **this is generalizable to any other number of dimensions**.

A 3D vector (or a 3D point) in **inhomogeneous coordinates** looks like:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{R}^3$$

while the same vector in **homogeneous coordinates** has the form (note the tilde in the notation):

$$\tilde{\mathbf{x}} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda z \\ \lambda \end{bmatrix} \in \mathbb{R}^4$$

We can go back by dividing the three first coordinates by the fourth:

$$\tilde{\mathbf{x}} = \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda z \\ \lambda \end{bmatrix} \Rightarrow \mathbf{x} = \begin{bmatrix} x/\lambda \\ y/\lambda \\ z/\lambda \end{bmatrix} \in \mathbb{R}^3$$

This way, the family of homogeneous vectors with $\lambda \neq 0$ represents the same point in $\mathbb{R}^3$, since λ doesn't affect. Another consequence of λ is that **any transformation in homogeneous coordinates holds for any scaled matrix**:

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{bmatrix} \begin{bmatrix} \lambda x_1 \\ \lambda x_2 \\ \lambda x_3 \\ \lambda \end{bmatrix} = \lambda \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$$

Then, the following transformations are equivalent:

$$\begin{bmatrix} 1 & -3 & 2 & 5 \\ 4 & 2 & 1 & 2 \\ 4 & -1 & 0 & 2 \\ -6 & 2 & 1 & 2 \end{bmatrix} \equiv \lambda \begin{bmatrix} 1 & -3 & 2 & 5 \\ 4 & 2 & 1 & 2 \\ 4 & -1 & 0 & 2 \\ -6 & 2 & 1 & 2 \end{bmatrix}$$

This indetermination is typically handled by fixing one entry of the matrix, (e.g. $p_{44} = 1$). Also, these matrices must be non-singular (Rank = 4).

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
import scipy.stats as stats
from ipywidgets import interact, fixed, widgets
from mpl_toolkits.mplot3d import Axes3D
from math import sin, cos, radians
%matplotlib widget

matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)
images_path = './images/'
```

8.2.1 Why do we want this? Reason one

Now that we know how homogenous coordinates and homogenous transformations works, it's time for understanding **why this is interesting**.

For now, we were performing complete transformations (rotations and translations) by using a rotation matrix and adding a translation vector to the rotated points ($\mathbf{p}' = \mathbf{R}\mathbf{p} + \mathbf{t}$).

The problem of this transformation is that the **concatenation of transformations** when a sequence of transformations has to be done becomes a mess:

$$\mathbf{p}' = \mathbf{R}_1\mathbf{p} + \mathbf{t}_1$$
$$\mathbf{p}'' = \mathbf{R}_2\mathbf{p}' + \mathbf{t}_2 = \mathbf{R}_2(\mathbf{R}_1\mathbf{p} + \mathbf{t}_1) + \mathbf{t}_2 = \mathbf{R}_2\mathbf{R}_1\mathbf{p} + \mathbf{R}_2\mathbf{t}_1 + \mathbf{t}_2$$

In the video game context, in hierarchical 3D models appearing in the scene, like a human figure, each body part's position depends on the transformations of its parent parts, so concatenations are necessary to ensure the correct relative positioning (e.g., the hand moves with the arm and shoulder). Without concatenation, moving one part independently would break the hierarchy, leading to unrealistic or disjointed animations.

What happens if we use homogenous coordinates?

We can express a transformation consisting of a rotation + translation using only a matrix multiplication:

$$\tilde{\mathbf{p}}' = \mathbf{T}_1\tilde{\mathbf{p}} \Rightarrow \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11}x + r_{12}y + r_{13}z + t_x \\ r_{21}x + r_{22}y + r_{23}z + t_y \\ r_{31}x + r_{32}y + r_{33}z + t_z \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} r \\ r \\ r \end{bmatrix}$$

Note that the 3×3 left-top submatrix of the $\mathbf{T}_1$ matrix is a rotation matrix while the last column contains the desired translation. This is the main reason for using homogeneous coordinates, look **how concatenation is applied now!**

$$\tilde{\mathbf{p}}' = \mathbf{T}_1\tilde{\mathbf{p}}$$
$$\tilde{\mathbf{p}}'' = \mathbf{T}_2\tilde{\mathbf{p}}' = \mathbf{T}_2\mathbf{T}_1\tilde{\mathbf{p}}$$

Concatenation becomes much easier, being only consecutive matrix multiplications (remember that, nowadays, matrix multiplications are very fast using GPUs).

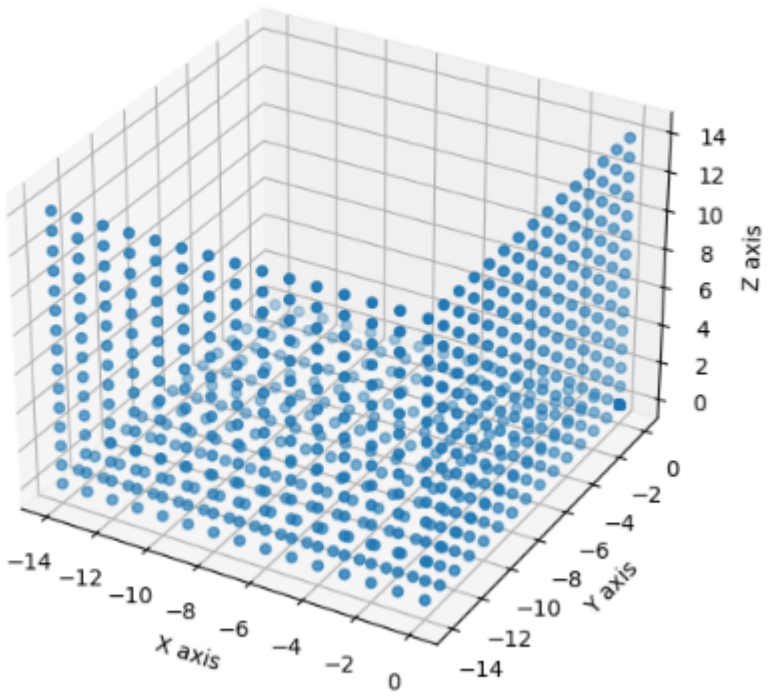
Also in first-person video games, the camera transformation matrix is updated every frame (e.g., 60 times per second for 60 FPS) to reflect the latest position and orientation. Thanks to homogeneous coordinates, both rotation and translation can be efficiently applied in a single matrix operation, which is crucial for real-time performance.

Let's play a bit with homogeneous coordinates. We are going to apply a homogenous transformation to a 3D object (a set of 3D-points, in fact). For this, we have defined `data` $\in \mathbb{R}^4$, a **matrix containing more than 3k points in homogenous coordinates**:

In [2]:
```
# Load data
data = np.load("./data/data.npy")
print('Number of points:', data.size/4)

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')

# Name axes
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

# Plot points
ax.scatter(data[0,:], data[1,:], data[2,:]);
```

```
Number of points: 3375.0
```

Figure



ASSIGNMENT 1: Homogeneous transformations for the win

Now, create a new method called `apply_homogeneous_transformation()` that builds a homogeneous matrix from some *yaw*, *pitch* and *roll* values as well as a translation vector and applies it to the input data matrix `data`. *Note that we are not transforming vectors, but points, so use* `scatter()` *instead of* `quiver()`.

Notice that opposite to the euclidean case, here both rotation and translation are applied just with one matrix multiplication! (`t` in the following code).

Recall the matrices defining the elemental rotations:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix} \qquad \mathbf{R}_x$$

In [3]:
```
# ASSIGNMENT 7
def apply_homogeneous_transformation(data, yaw, pitch, roll, translation):
    """ Apply a linear transformation to a set of 3D-vectors and plot them

        Args:
            data: Input set of points to transform
```

```
            yaw: Degrees to rotate the coordinate system around the 'Z' axis
            pitch: Degrees to rotate the coordinate system around the 'Y' axis
            roll: Degrees to rotate the coordinate system around the 'X' axis
            translation: Column vector containing the translation for each axis
    """
    # Write your code here!

    # Transform to radians
    yaw = radians(yaw)
    pitch = radians(pitch)
    roll = radians(roll)

    # Construct rotation matrices
    Rx = np.array([[1, 0, 0],[0, cos(roll), -sin(roll)],[0, sin(roll), cos(roll)
    Ry = np.array([[cos(pitch), 0, sin(pitch)],[0, 1, 0],[-sin(pitch), 0, cos(pi
    Rz = np.array([[cos(yaw), -sin(yaw), 0],[sin(yaw), cos(yaw), 0],[0, 0, 1]])

    # Combine rotation matrices
    R = Rz @ Ry @ Rx

    # Create homogenous transformation matrix
    t = np.zeros((4,4))
    t[0:3,0:3] = R
    t[0:3,3] = translation
    t[3,3] = 1

    transformed = t @ data

    # Create figure
    fig = plt.figure()

    # Prepare figure for 3D data
    ax = plt.axes(projection='3d')

    # Name axes
    ax.set_xlabel('X axis')
    ax.set_ylabel('Y axis')
    ax.set_zlabel('Z axis')

    # Plot points
    ax.scatter(transformed[0,:], transformed[1,:], transformed[2,:]);
```

Now apply the following transformation to the object previously loaded:

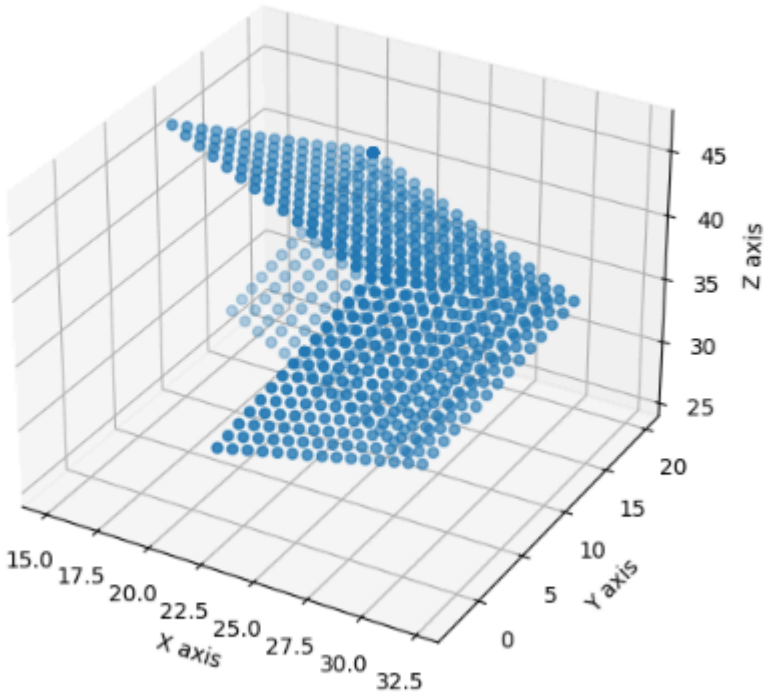
- **Yaw** rotation: 45 degrees
- **Pitch** rotation: -60 degrees
- **Roll** rotation: 20 degrees
- **Translation**:
  - X-axis: 20 units
  - Y-axis: 20 units
  - Z-axis: 40 units

Remember that this is going to be performed only using a homogeneous transformation (one unique matrix multiplication)!

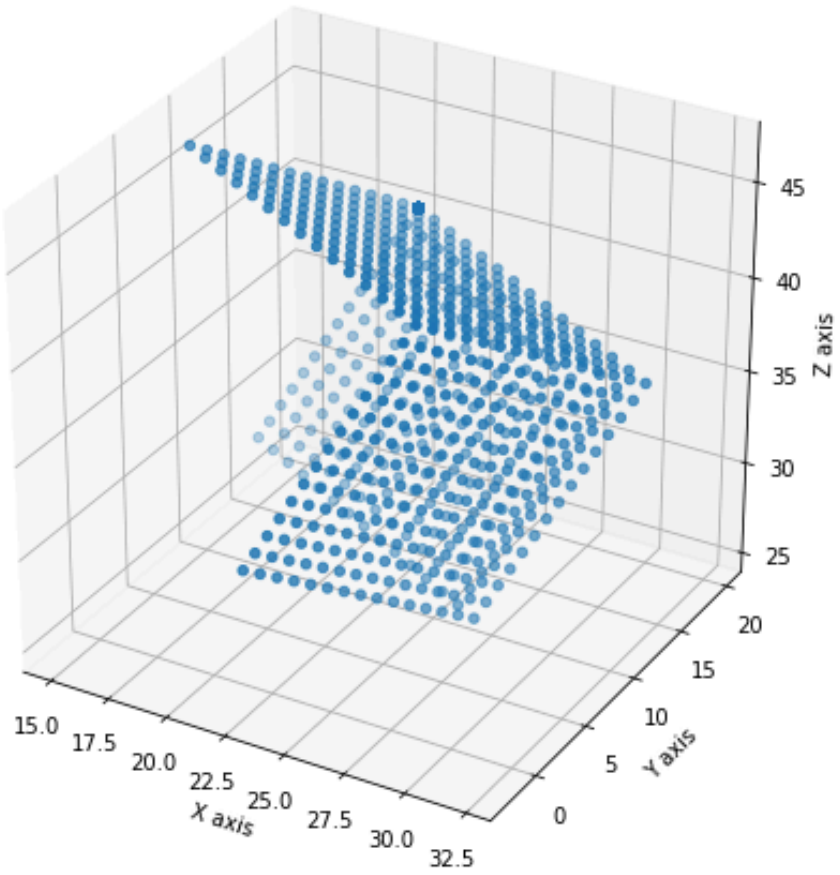
In [4]:
```
# Load data
data = np.load("./data/data.npy")
```

```
# Apply transformation
apply_homogeneous_transformation(data, 45, -60, 20, [20,20,40])
```

Figure



Expected output:



By doing these transformations, our graphic engine could represent moving objects in the `WORLD` when the player is still (e.g. flying birds, cars, other players, etc.).

Checking execution time

Just as a curiosity, let's check how much time is required by python to apply a transformation in both euclidean and homogeneous coordinates. *Note: take care with the length, if it is too large it could freeze your computer!*

In [5]:
```
# Set the number of points to transform
length = 6000000

# Prepare data in Euclidean coordinates
Pts = np.random.rand(3,length)
R = np.random.rand(3,3)
T = np.random.rand(3,1)

# Prepare data in Homogeneous coordinates
Ones = np.ones(length)
Pts_H = np.row_stack((Pts,Ones))
T_H = np.random.rand(4,4)

import time

# Apply euclidean transformation
start_time = time.time()
res = R@Pts+T
print("Time spent with Euclidean tranformation: %s seconds ---" % (time.time() -
```

```
# Apply Homogeneous transformation
start_time = time.time()
res = T_H@Pts_H
print("Time spent with Homogeneous tranformation: %s seconds ---" % (time.time()
```

```
Time spent with Euclidean tranformation: 0.05561256408691406 seconds ---
Time spent with Homogeneous tranformation: 0.036771297454833984 seconds ---
```

```
C:\Users\vital\AppData\Local\Temp\ipykernel_10196\933578799.py:11: DeprecationWar
ning: `row_stack` alias is deprecated. Use `np.vstack` directly.
  Pts_H = np.row_stack((Pts,Ones))
```

ASSIGNMENT 2: How the player sees the world

One final example, consider the following image, in where our character Joel is looking at a dystopian, post-apocalyptic scenario.

The `WORLD` reference system is displayed in orange and labeled with $\{\mathbf{W}\}$ while the reference system of the character's view is displayed in red and labeled as $\{\mathbf{C}\}$. We know that:

- the position and orientation of $\{\mathbf{C}\}$ w.r.t. $\{\mathbf{W}\}$ is given by a *yaw* angle of -45º, a *roll* angle of -90 and a translation of [0.0,-4.0,1.2] meters in $[x,y,z]$ axes, respectively, and
- the coordinates of the point $\mathbf{p}^W$ in the world are [30.0,1.0,0.5] meters.

Could you compute what are its coordinates w.r.t our character's point of view in homogeneous coordinates ($\tilde{\mathbf{p}}^C$)? In other words, we have to build the homogeneous transformation $\mathbf{T}$ that produces $\tilde{\mathbf{p}}^C = \mathbf{T}\tilde{\mathbf{p}}^W$. As we will see in future notebooks, knowing such coordinates is vital to get the position of the 3D point **in the image** that Joel would see if this game were in first-person (fortunately it's not!).

In [6]:
```
# ASSIGNMENT 8
# Write your code here!
```

```
p = np.array([30,1,0.5,1])
p = np.vstack(p)

# Transform to radians
yaw = radians(-45)
pitch = radians(0)
roll = radians(-90)

# Construct rotation matrices
Rx = np.array([[1, 0, 0],[0, cos(roll), -sin(roll)],[0, sin(roll), cos(roll)]])
Ry = np.array([[cos(pitch), 0, sin(pitch)],[0, 1, 0],[-sin(pitch), 0, cos(pitch)
Rz = np.array([[cos(yaw), -sin(yaw), 0],[sin(yaw), cos(yaw), 0],[0, 0, 1]])

# Combine rotation matrices
R = Rz @ Ry @ Rx

# Create homogenous transformation matrix
t = np.zeros((4,4))
t[0:3,0:3] = R
t[0:3,3] = np.array([0, -4, 1.2])
t[3,3] = 1

transformed = np.linalg.inv(t) @ p
print(transformed)
```

```
[[17.67766953]
 [ 0.7       ]
 [24.74873734]
 [ 1.        ]]
```

Expected output (homogeneous):

```
[[17.67766953]
 [ 0.7       ]
 [24.74873734]
 [ 1.        ]]
```

Thinking about it (1)

Now you are in a good position to answer these questions:

- What is the length of a 3D cartesian vector in homogeneous coordinates?

  The length of a 3D cartesian vector in homogeneous coordinates would be 4, since we add a positive number (usually 1) to the other 3 it had.

- How many operations do you need to transform a point from the world frame to the camera one using euclidean coordinates? and using homogeneous coordinates?

  You need 2 operations using euclidean coordinates: a matrix product (rotation) and a sum (translation), while you only need 1 operation using homogeneous coordinates: a matrix product (rotation and translation simultaneously).

- Explain the difference in the execution time when using the two types of transformations.

As mentioned above, since you only need 1 operation for homogeneous coordinates transformation, this one takes less time to compute than euclidean coordinates transformation, which requires 2 operations, thanks to matrix multiplications being optimised in GPUs.

- Why are the rotations applied in that order? Could they have been applied differently?

  The rotations are applied in that order because it is the most popular among its application fields. They can not be applied differently since matrix multiplications are not commutative.

8.2.2 Why do we want this? Reason two

There is another reason justifying the utilization of homogeneous coordinates when dealing with transformations: they result in a natural model for the camera, since points in the image plane are projection rays in $\mathbb{R}^3$.

In 1D:

- Cartesian coordinates: $x = x_1 = 3$.
- Homogeneous coordinates: $x = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} 6 \\ 2 \end{bmatrix} \equiv \begin{bmatrix} 3k \\ k \end{bmatrix}, \quad k \neq 0$

All the points in homogeneous coordinates represents the same point in cartesians since $x = x_1/x_2 = 3$.

In 2D:

- Cartesian coordinates: $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2$.
- Homogeneous coordinates: $x = k \begin{bmatrix} x_2 \\ x_1 \\ 1 \end{bmatrix}, \quad k \neq 0$

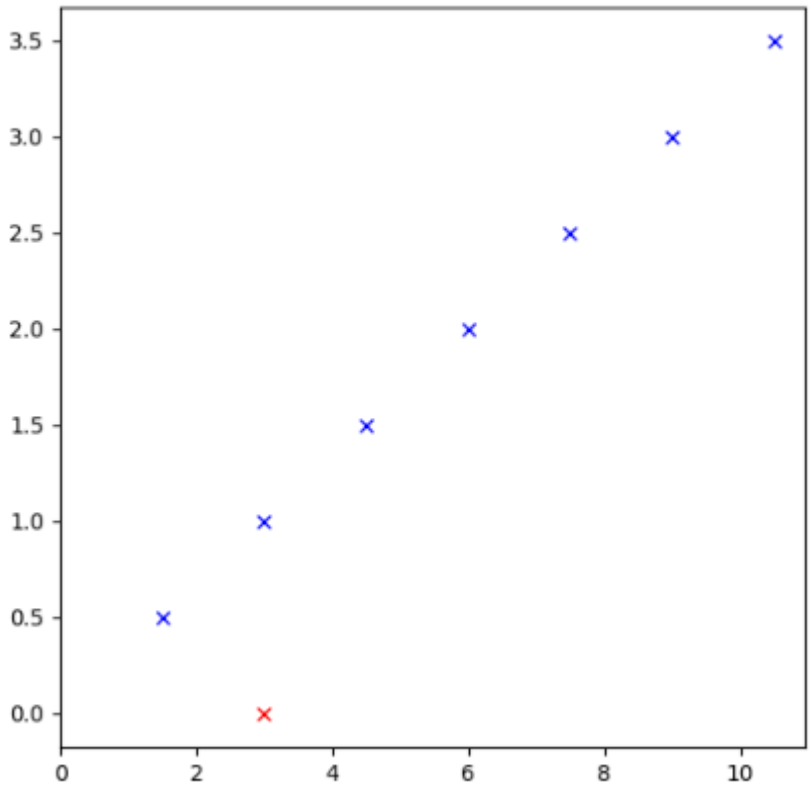
So in homogeneous coordinates, a point in the plane $\mathbb{R}^2$ transforms to a line passing through the origin in a reference frame parallel to the image plane (perpendicular to x_3).

The following code illustrates this for a 1D point.

In [7]:
```
# 1D point
x1 = 3
plt.figure()
plt.plot(x1,0,'rx')

# Point in homogeneous
x = np.array([3,1])

# Equivalent points by multiplying by lambda
for lambda_ in np.arange(0.5,4,0.5):
    plt.plot(x[0]*lambda_,x[1]*lambda_,'bx')
```

```
plt.xlim(0)
plt.show()
```

Figure



Conclusion

Awesome!

In his notebook we have learned:

- The principles of homogeneous coordinates.
- How to rotate and translate points or vectors using homogeneous coordinates.

Let's keep learning!

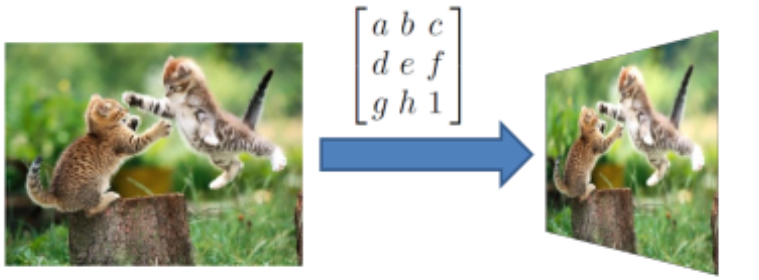
8.3 2D Homographies

A 2D homography is a very general linear transformation between planes. In the previous notebook we applied rotations and translations to 3D points, which are examples of 3D euclidean homographies.

In computer vision, a homography is usually defined as a **perspective transformation of a plane**, in other workds, a reprojection of a plane from one camera image into a different one, where the camera may have been translated and/or rotated. As a consequence, **any two images of the same planar surface in space are related by a homography**. This has many practical applications, such as image rectification, image registration, augmented reality, etc.

For 2D homographies, we have a $3x3$ matrix containing the linear transformation:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \qquad \begin{aligned} x' &= u \,/\, w \\ y' &= v \,/\, w \end{aligned}$$



Notice that in such transformation:

- Lines are kept straight.
- Incident lines remain.

In this notebook we will learn:

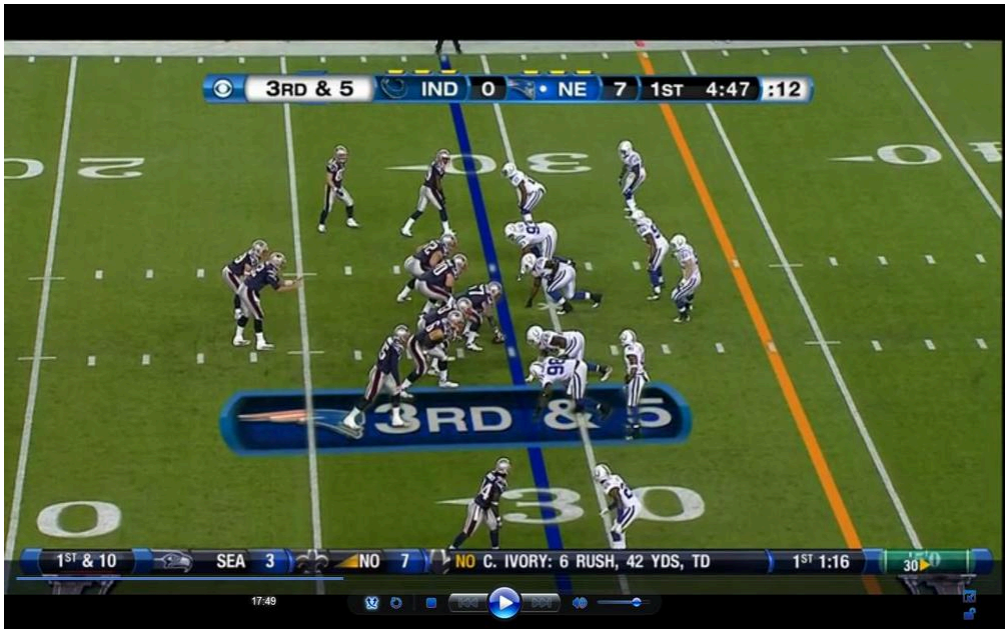
- different types of homographies (section 8.3.1).
- how to find the parameters for a given homography in 2D (section 8.3.2).
- under which conditions a homography exists (section 8.3.3).

Problem context - Homography in american football

American football is a team sport played by two teams of eleven players on a rectangular field with goalposts at each end. For the cool things we are going to implement we just

need to know some points about this sport:

- The offense (the team with possession of the football) has 4 tries (called downs) for making the ball advance at least 10 yards (approx 9 meters).
- If a player passes the **down line** (yellow) with the ball, the offense have another 4 tries to advance 10 yards (from the point where the football was lost).
- **On television, a yellow line is electronically superimposed on the field to show the first down line to the viewing audience. Also, a blue line is superimposed showing where the play (the football) starts.**



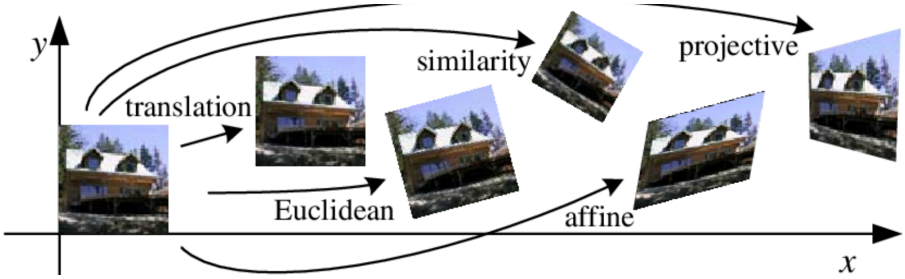
As you can see in the image above, the offense is in the **3rd try** for advancing 10 yards (they advanced 5 yards in previous tries, so **the objective is 5 yards**). That information is given by a rectangle placed in the field showing **3rd (try) & 5 (yards to go)**.

Our objective in this notebook is to **place the blue line** (where the ball starts), the **yellow line** (down line) and the placed **rectangle** (which gives to the audience some information) **using homographies**. Cool, isn't?

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
from math import cos,sin,pi

matplotlib.rcParams['figure.figsize'] = (12.0, 6.0)
images_path = './images/'
```

8.3.1 2D homography hierarchy levels

But let's start from the beginning. All 2D homographies are linear transformations of 2D points, but there is a hierarchy level where we can distinguish different types of homographies (and, hence, transformations):



Throughout the next section we will cover these homography types. Let's go!

LEVEL 1: Translation

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

Translation is the most simple homography, since pixels of the plane are **only shifted** in both X and Y directions.

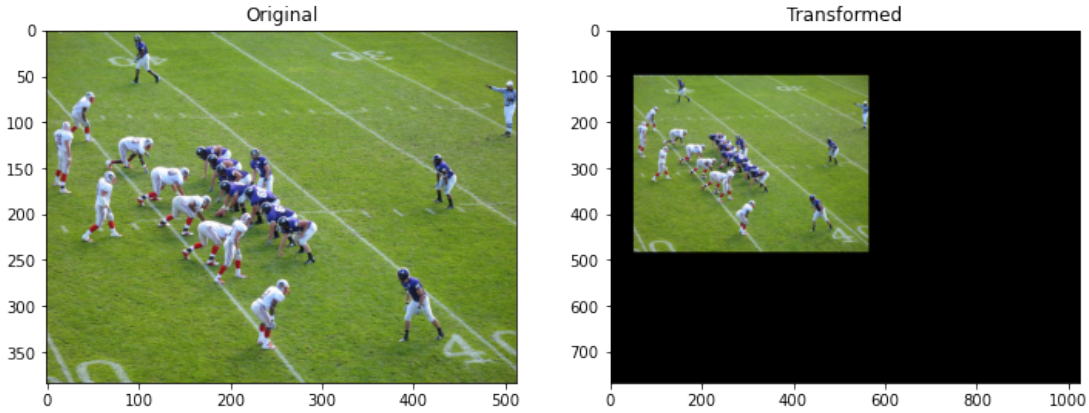
ASSIGNMENT 1a: Playing with... wait for it... translation!

Your first task is to:

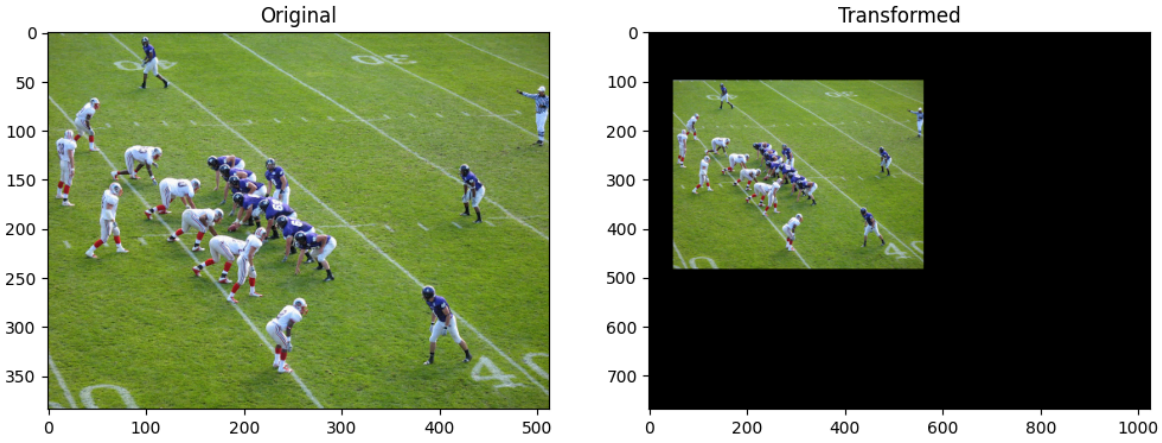
1. Define a transformation matrix `M` that applies an arbitrary translation to the image `football.jpg`. Define such a matrix as having a `float64` format (more info about NumPy data types here).
2. Apply such a homography to an image using the OpenCV function `cv2.warpPerspective()`. Take a look at its input arguments.
3. Finally, show both the original and the resulting image.

Note: The size of the output image depends on the transformation we want to do, but for the following examples you can use the double of the width and height of the original image.

Output example:



In [2]:
```
# Write your code here!
matplotlib.rcParams['figure.figsize'] = (12.0, 6.0)

# Read image
image = cv2.imread(images_path + 'football.jpg')
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Define transformation matrix
M = np.array([[1,0,50],[0,1,100],[0,0,1]], dtype=float)

# Apply homography
n_rows = image.shape[0]*2
n_cols = image.shape[1]*2
transformed = cv2.warpPerspective(image, M, (n_cols,n_rows))

# Show the resulting image
plt.subplot(121)
plt.title("Original")
plt.imshow(image)
plt.subplot(122)
plt.title("Transformed")
plt.imshow(transformed);
```



LEVEL 2: Euclidean transformation

A linear transformation defining an euclidean (rigid) transformation has the shape:

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & t_x \\ sin(\theta) & cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} cos(\theta)x - sin(\theta)y + t_x \\ sin(\theta)x + cos(\theta)y + t_y \\ 1 \end{bmatrix}$$

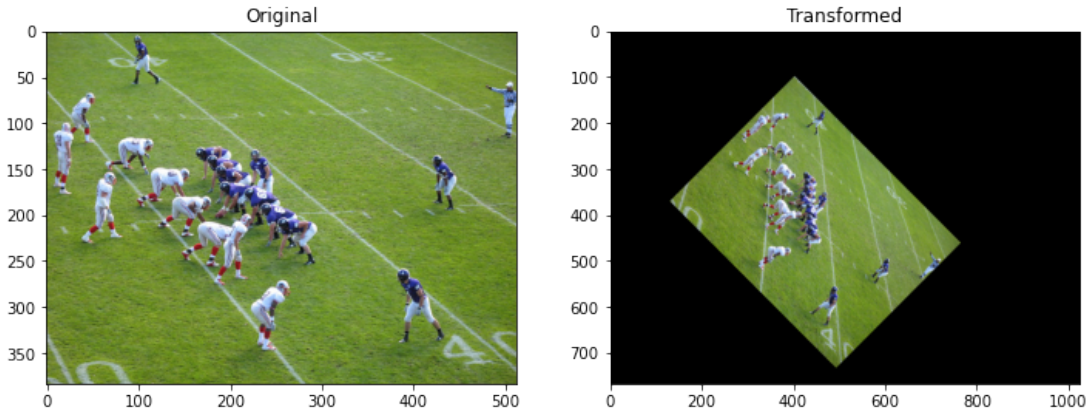
As we saw in the previous notebook, this homography applies **rotation + translation**. *Note that rotation is around the origin of coordinates (0,0), which is the top-left corner of the image.*

ASSIGNMENT 1b: Euclidean transformation comes into play

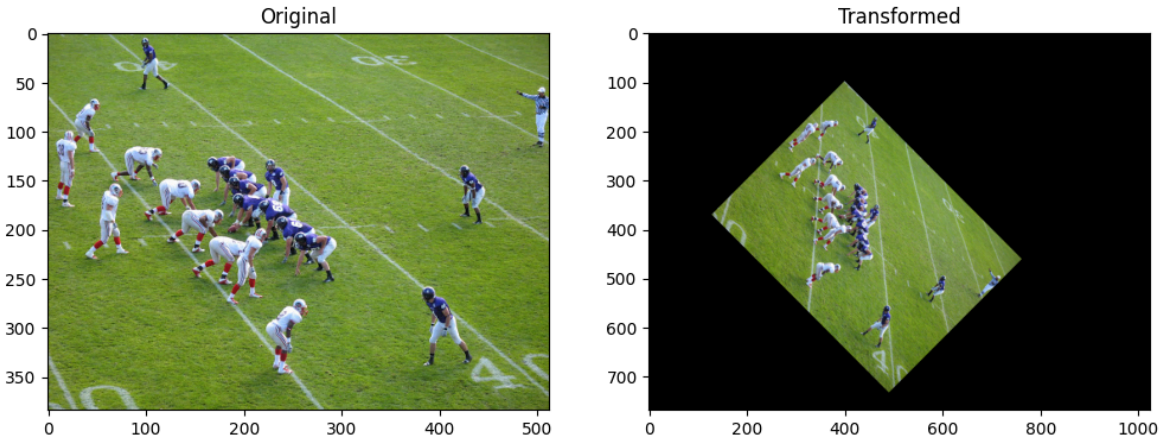
Repeat the previous exercise using an Euclidean transformation, that is:

1. define the transformation matrix `M`,
2. apply it to the image,
3. show the initial and the resulting images.

Output example:



In [3]:
```
# Write your code here!

# Define transformation matrix
theta = pi/4
M = np.array([[cos(theta),-sin(theta),400],[sin(theta),cos(theta),100],[0,0,1]],

# Apply homography
n_rows = image.shape[0]*2
n_cols = image.shape[1]*2
transformed = cv2.warpPerspective(image, M, (n_cols,n_rows))

# Show the resulting image
plt.subplot(121)
plt.title("Original")
plt.imshow(image)
plt.subplot(122)
plt.title("Transformed")
plt.imshow(transformed);
```



LEVEL 3: Similarity

The similarity transformation adds a scale to the previous one, so:

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta) & -sin(\theta) & t_x \\ sin(\theta) & cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s \cdot cos(\theta) & -s \cdot sin(\theta) & s \\ s \cdot sin(\theta) & s \cdot cos(\theta) & s \\ 0 & 0 \end{bmatrix}$$

That is, this homography applies **rotation + translation + scale**.

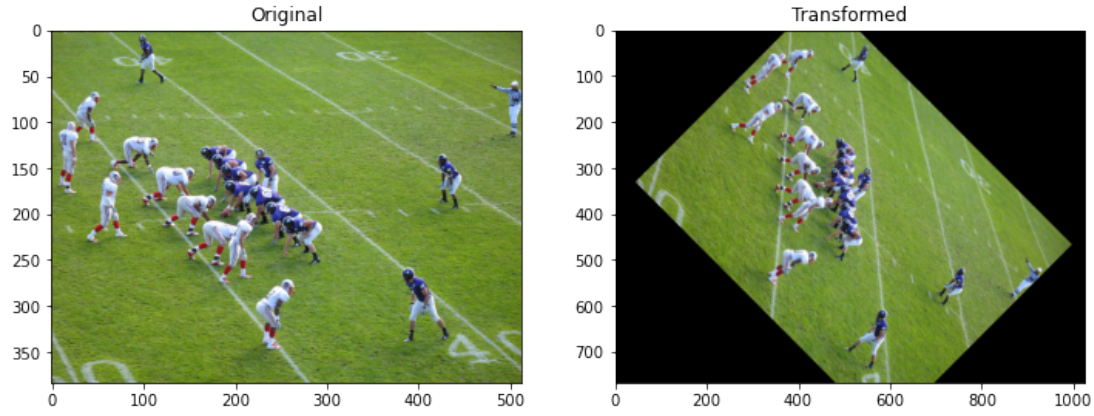
Note that scale have to be equal for both axes x and y.

ASSIGNMENT 1c: Applying a similarity transformation

Repeat the previous exercise using a similarity transformation, that is:

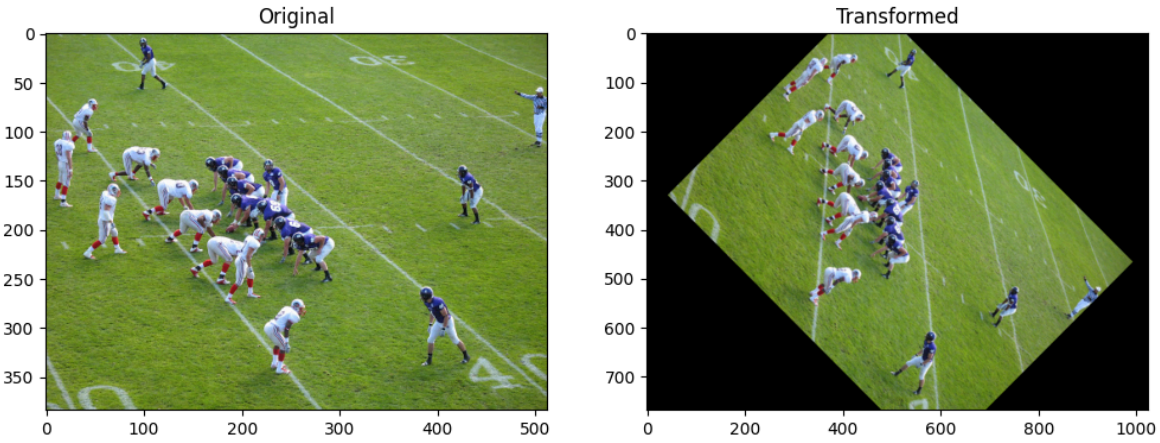
1. define the transformation matrix `M`,
2. apply it to the image,
3. show the initial and the resulting images.

Output example:



In [4]:
```
# Write your code here!

# Define transformation matrix
theta = pi/4
```

```
scale = 1.5
M = np.array([[scale*cos(theta),-scale*sin(theta),scale*300],[scale*sin(theta),s

# Apply homography
n_rows = image.shape[0]*2
n_cols = image.shape[1]*2
transformed = cv2.warpPerspective(image, M, (n_cols,n_rows))

# Show the resulting image
plt.subplot(121)
plt.title("Original")
plt.imshow(image)
plt.subplot(122)
plt.title("Transformed")
plt.imshow(transformed);
```



LEVEL 4: Affine

The affine transformation can be defined as:

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & t_x \\ h_{10} & h_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} h_{00}x + h_{01}y + t_x \\ h_{10}x + h_{11}y + t_y \\ 1 \end{bmatrix}$$

This is an interesting homography because it appears quite often in real life (e.g. when the depth variation within the planar object and a camera is large):

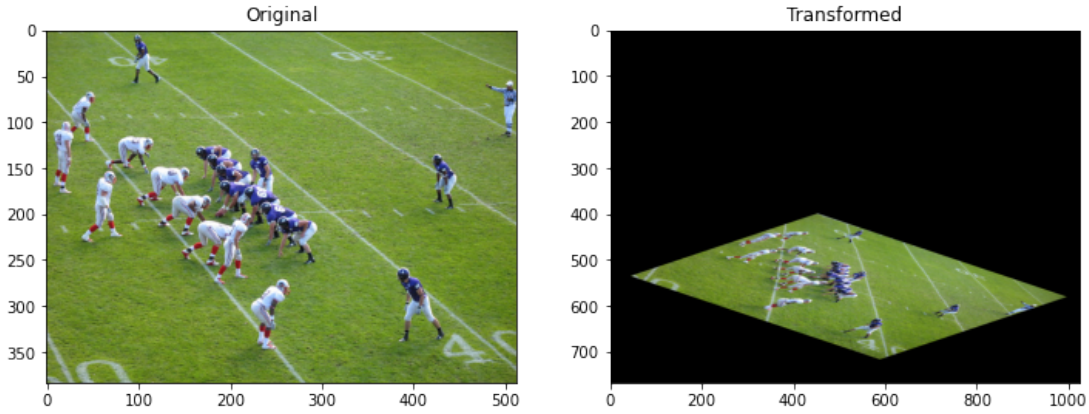
We can deffine it as a **rotation + translation + scale (different for each axis)** transform.

ASSIGNMENT 1d: Affine transformation, I choose you

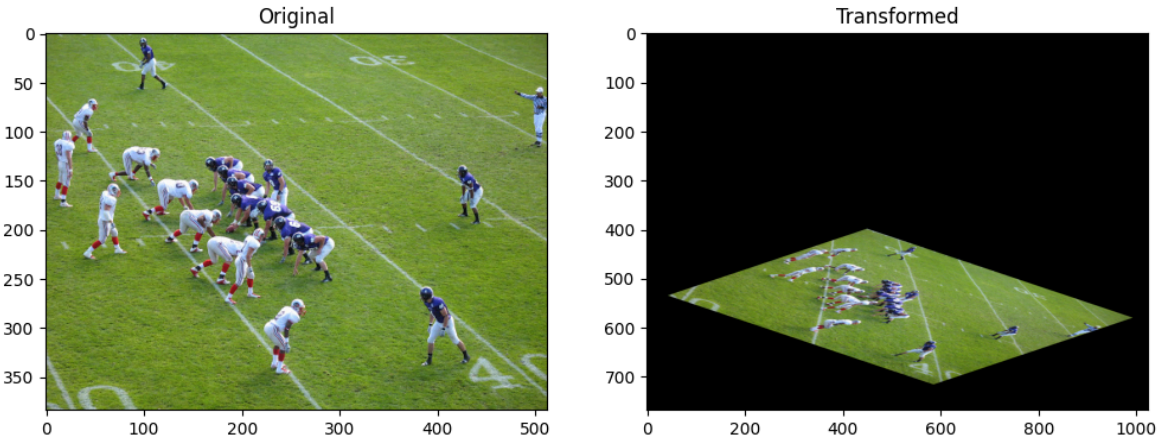
Repeat the previous assignment but using an affine transformation, that is:

1. define the transformation matrix `M`,
2. apply it to the image,
3. show the initial and the resulting images.

Output example:



In [5]:
```
# Write your code here!

# Define transformation matrix
theta = pi/4
s_x = 1.5
s_y = 1/2
M = np.array([[s_x*cos(theta),-s_x*sin(theta),s_x*300],[s_y*sin(theta),s_y*cos(t

# Apply homography
n_rows = image.shape[0]*2
n_cols = image.shape[1]*2
transformed = cv2.warpPerspective(image, M, (n_cols,n_rows))

# Show the resulting image
plt.subplot(121)
plt.title("Original")
plt.imshow(image)
plt.subplot(122)
plt.title("Transformed")
plt.imshow(transformed);
```

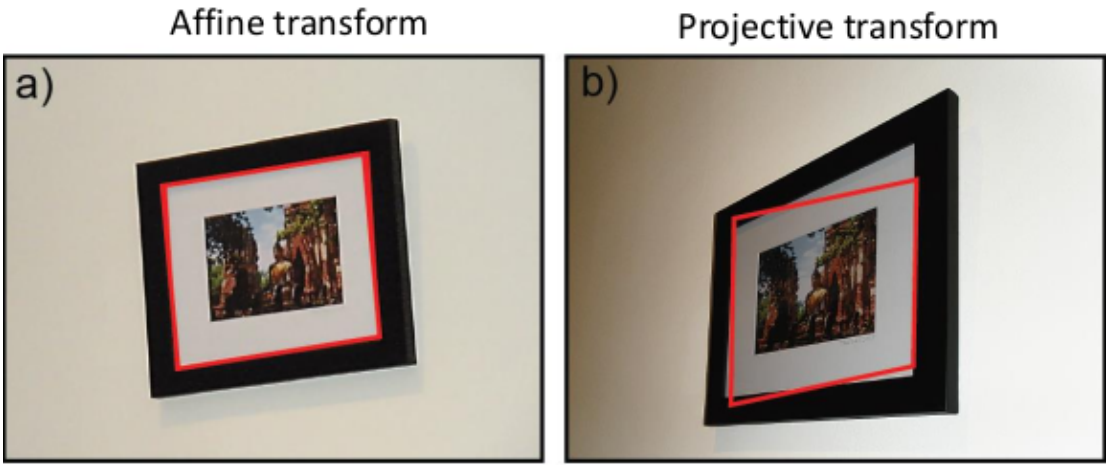


LEVEL 5: Projective

And we have reached the top level, the projective transformation:

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} h_{00}x + h_{01}y + h_{02} \\ h_{10}x + h_{11}y + h_{12} \\ h_{20}x + h_{21}y + 1 \end{bmatrix}$$

This is the general 2D homography since there are no constraints. Unlike the affine transformation, projective transformations appear in real life when variation in depth is comparable to the distance from the camera to an object:

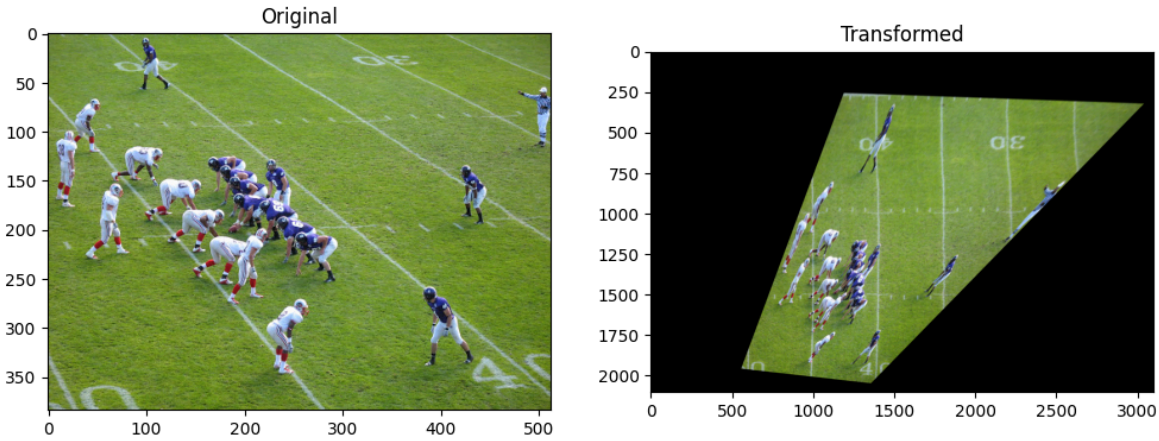


Note that we used $h_{22} = 1$ for fixing the scale factor in linear transformations. This constraint can be replaced with $||\mathbf{h}|| = \mathbf{1}$ (better).

The projective transformation is a very powerful tool, and it has many applications. **Take a look at the following example:**

In [6]:
```
# Define transformation matrix
M = np.array([[3.1654,0.03225,1191.14702],[0.0799604,10.2911,263.9896],[-0.00014

# Apply homography
transformed = cv2.warpPerspective(image, M, (3100,2100))
```

```
# Show the resulting image
plt.subplot(121)
plt.title("Original")
plt.imshow(image)
plt.subplot(122)
plt.title("Transformed")
plt.imshow(transformed);
```



As you can see, the perspective of the image has been **removed** and the resultant (transformed) image is *like* an image taken from the air.

But, you can see that the coefficient of the employed transformation matrix are not some randomly picked numbers, so how can you obtain the specific matrix `M` that is able to do this? The short answer is from a set of corresponding points (rings any bell?). And going one step further: **How can a matrix `M` be obtained having pairs of correspondences**?

Just as a preview, in the example above, we have used the four corners of the rectangle that the white lines are forming and we have made them corresponded with an actual rectangle without perspective. Do not worry, we will explain this later, but first let's have a look at the maths behind this.

8.3.2 Solving the 2D homography: Direct Lineal Transformation (DLT)

As you know, a general 2D homography that converts a point $p = (x, y)$ to another point $p' = (x', y')$ looks like this (in homogeneous coordinates):

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If we know the coordinates of the corresponding points in both images, we can build an equations' system to solve the coefficients of the matrix. Furthermore, knowing a certain amount of corresponding points that share the same homography will allow us to get a(n) (over)determined equation system to get them.

In this case, from a certain pair of points i, we can form a linear equations system in order to isolate the matrix variables:

$$x'_i = \frac{h_{00}x_i + h_{01}y_i + h_{02}}{h_{20}x_i + h_{21}y_i + 1} \longrightarrow x'_i(h_{20}x_i + h_{21}y_i + 1) = h_{00}x_i + h_{01}y_i + h_{02} \quad (1)$$

$$y'_i = \frac{h_{10}x_i + h_{11}y_i + h_{12}}{h_{20}x_i + h_{21}y_i + 1} \longrightarrow y'_i(h_{20}x_i + h_{21}y_i + 1) = h_{10}x_i + h_{11}y_i + h_{12} \quad (2)$$

This can be rewritten as:

$$A\mathbf{h} = 0 \longrightarrow \begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x'_i x_i & x'_i y_i & x'_i \\ 0 & 0 & 0 & -x_i & -y_i & -1 & y'_i x_i & y'_i y_i & y'_i \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

You can see that with less than 4 independent points ($rank(A) < 8$), there are infinitely many solutions behind $k\mathbf{h}$.

But, if we have **4 or more independent pair of points** we can build the following expression:

$$Ah = 0 \longrightarrow \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x'_1 x_1 & x'_1 y_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & y'_1 x_1 & y'_1 y_1 & y'_1 \\ & & & & \vdots & & & & \\ -x_n & -y_n & -1 & 0 & 0 & 0 & x'_n x_n & x'_n y_n & x'_n \\ 0 & 0 & 0 & -x_n & -y_n & -1 & y'_n x_n & y'_n y_n & y'_n \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

- For $\mathbf{n = 4}$ **independent pair of points** ($rank(A) = 8$), there is a solution $k\mathbf{h}$ with $\mathbf{h} \neq 0, k \neq 0$. This should be sufficient (and strictly it is, as we will test later), but the solution will be highly affected by the noise in the points coordinates, so this is not usually employed in applications with automatically detected points, but for manually chosen ones.

- For $\mathbf{n > 4}$ **independent pair of points** ($rank(A) > 8$), (e.g. those pairs of points detected by SIFT or Harris) there is not a solution (apart from $\mathbf{h} = 0$) as the system

is overdetermined. However, we can get the solution that minimizes the error in the points coordinates (that is, the least-squares solution):

- Arg. $\min_h ||\mathbf{Ah}||^2$ with $||\mathbf{h}|| = 1 \backslash [5pt]$
- Solution $\hat{h}$: eigenvector of the smallest eigenvalue of $A^T A$

Note that if you know the specific type of the transformation, you may need less points to solve this equation system:

- **Translation** (2 unknowns) $\rightarrow$ **1 pair of points needed**
- **Euclidean** (3 unknowns) $\rightarrow$ **2 pair of points needed**
- **Similarity** (4 unknowns) $\rightarrow$ **2 pair of points needed**
- **Affine** (6 unknowns) $\rightarrow$ **3 pair of points needed**
- **Projective** (8 unknowns) $\rightarrow$ **4 pair of points needed**

But, as we said, usually, you need many more points for noise robustness, so this is not relevant at all.

Also, each type of transformation entails certain transformations and invariance, the table below resume this:

	Euclidean	**Similarity**	**Affine**	**Projective**
Transformations:				
Rotation, translation	x	x	x	x
Isotropic scale		x	x	x
Axes scale			x	x
Perspective transformation				x
Invariants:				
Distance	x			
Angles, distance ratios	x	x		
Parallelism, mass center	x	x	x	
Cross-ratio	x	x	x	x

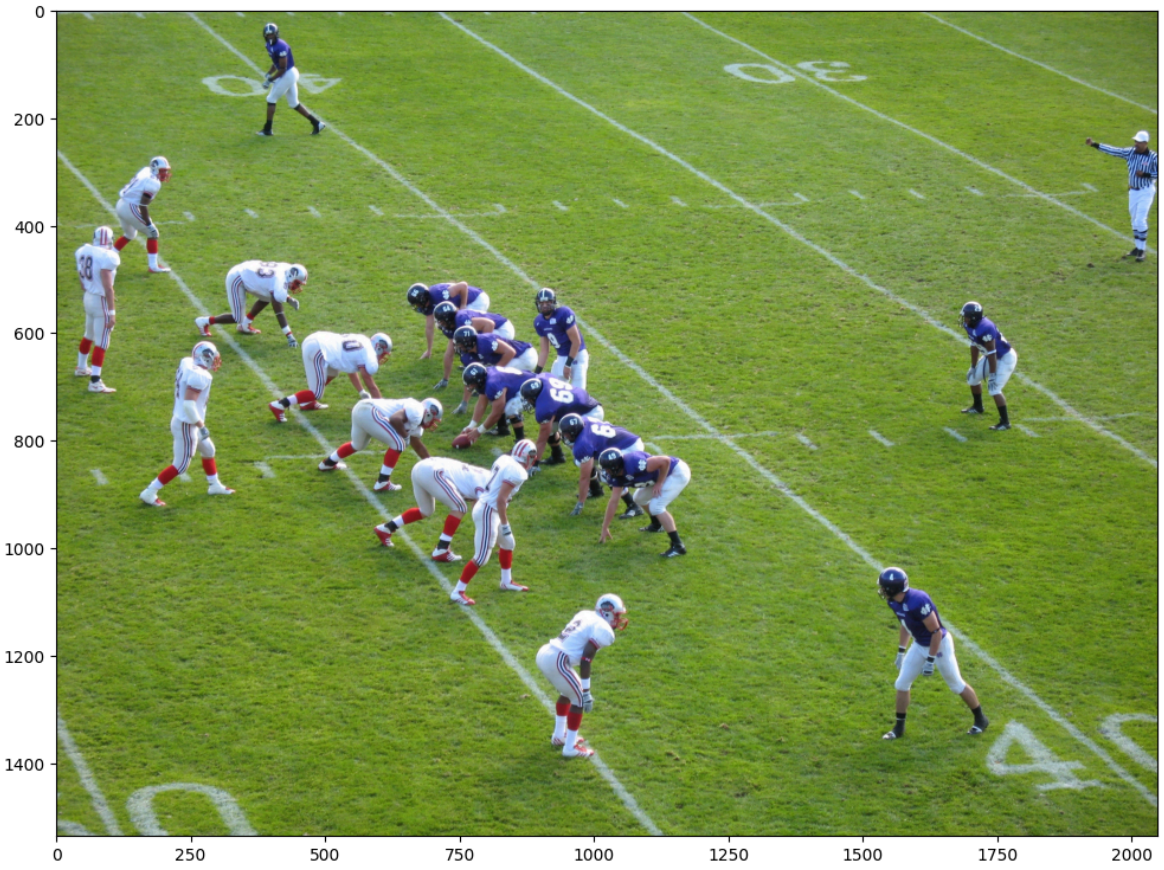
Let's apply your knowledge!

Now that we know how to solve the 2D homography, let's go back to our American football problem.

We are going to work with the image `football-big.jpg`, which is the same as the one used in previous exercises but with a better resolution and quality.

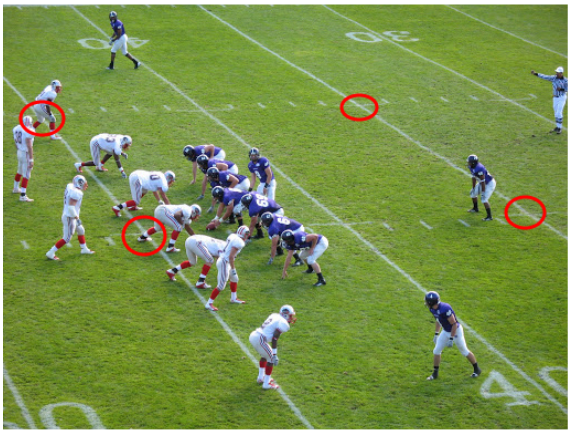
Let's start by reading and showing such color image:

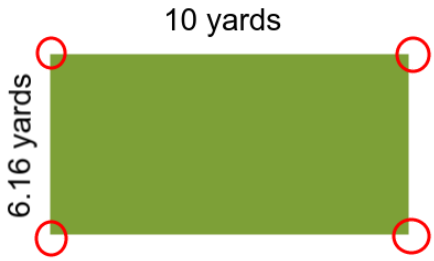
In [7]:
```
# Write your code here!

matplotlib.rcParams['figure.figsize'] = (12.0, 12.0)

# Read american football image.
image = cv2.imread(images_path + 'football-big.jpg')
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Show it
plt.imshow(image);
```



Our first task is to add the image `marker.jpg` to the field.

Recall that this means: 3rd (try) & 3 (yards to go). For adding an image without perspective to an image with perspective, the first step is to **remove the perspective** of the original image, and for that we are going to *map* the points of a rectangle on the field to **an actual rectangle**, but keeping the ratio of the sides' lengths of the rectangle (remember that a projective transformation is only invariant to the cross-ratio distances!):





Now that we have four (manually selected) corresponding points, we are going to compute the transformation matrix of that projective homography. This is easy in openCV, as it provides the `cv2.findHomography()` method, which takes two lists of corresponding points as input:

- the first list contains the input coordinates (x_i, y_i) and
- the second list their correspondences (x_i', y_i') (points without perspective).

Note that those points represent coordinates, their format is (n_col, n_row).

ASSIGNMENT 2a: Obtaining the transformation matrix

Your task here is to retrieve the transformation matrix that transforms points from the image with perspective to the one without it. For that:

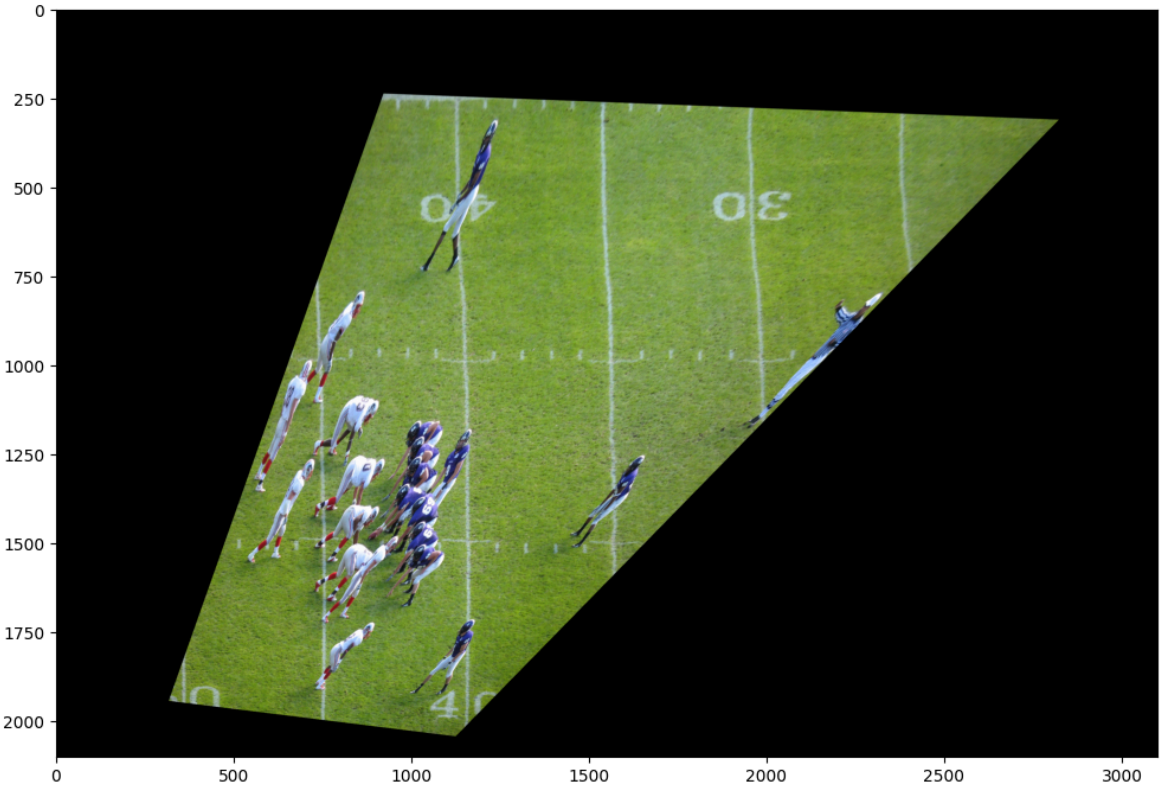
1. Take the **four corners of the specified rectangle** (you can use an external tool to get the coordinates),
2. and obtain the transformation matrix that removes the perspective of the football field. For that you have to define two pairs of correspondences, `pts_src` and `pts_dst`, linking points in the first image with points in the second one.
3. Then **use `cv2.warpPerspective()` with the obtained matrix** to remove the perspective and **show the resultant image**.
4. Finally, save it into your computer, we will need it for the next exercise.

Remember that the output rectangle should maintain the ratio of the original football field $(10 \ / \ 6.16)$ *when you define the list of the correspondent points, and for computation purposes use numbers that are "similar".*

Expected output in the next assignment.

In [8]:
```
# Write your code here!

# Points of a rectangle in the field
pts_src = np.array([[150, 410],[520, 840],[1300, 370],[1900, 760]])

# Points of a rectangle without perspective with r=1.6
pts_dst = np.array([[750, 1000], [750, 1500], [1562, 1000],[1562, 1500]])
```

```
# Calculate Homography
M, status = cv2.findHomography(pts_src, pts_dst)

# Warp source image to destination based on homography
no_perspective = cv2.warpPerspective(image, M, (3100,2100))

# Show image without perspective
plt.imshow(no_perspective)

# Save image
im_out = cv2.cvtColor(no_perspective,cv2.COLOR_RGB2BGR)
cv2.imwrite(images_path + "no_perspective.jpg",im_out);
```



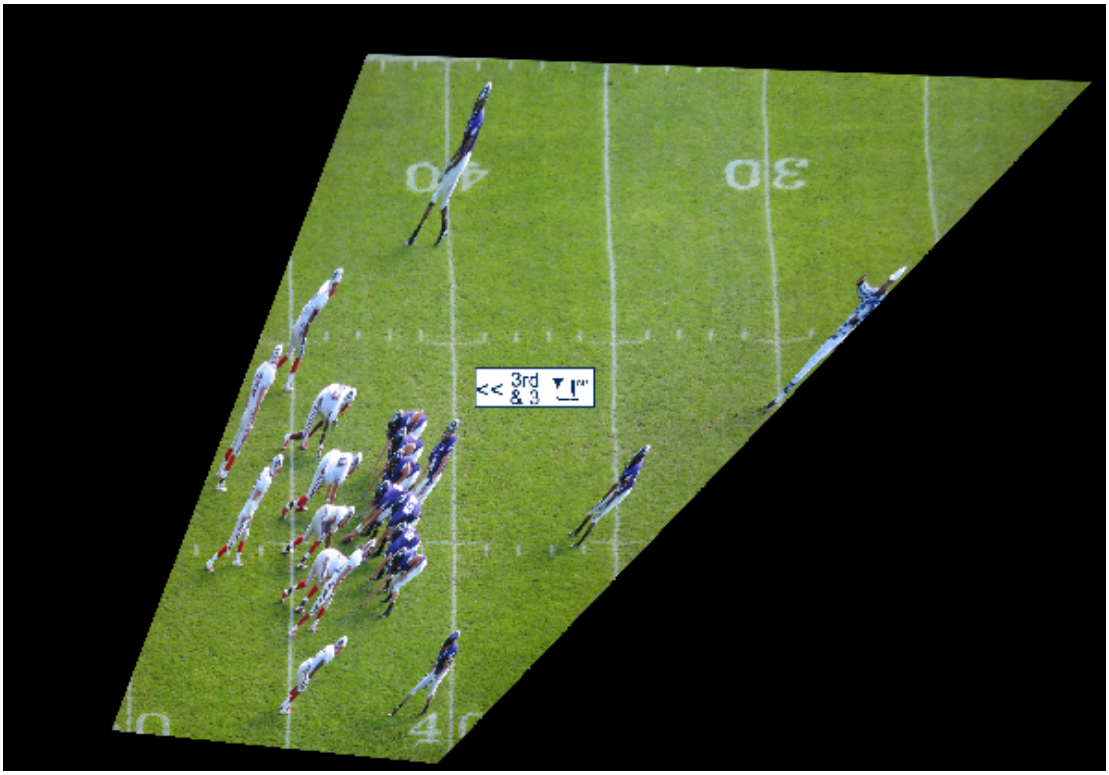
Overlapping an image

Now that we have the field without perspective, we can add the marker to the image in any place.

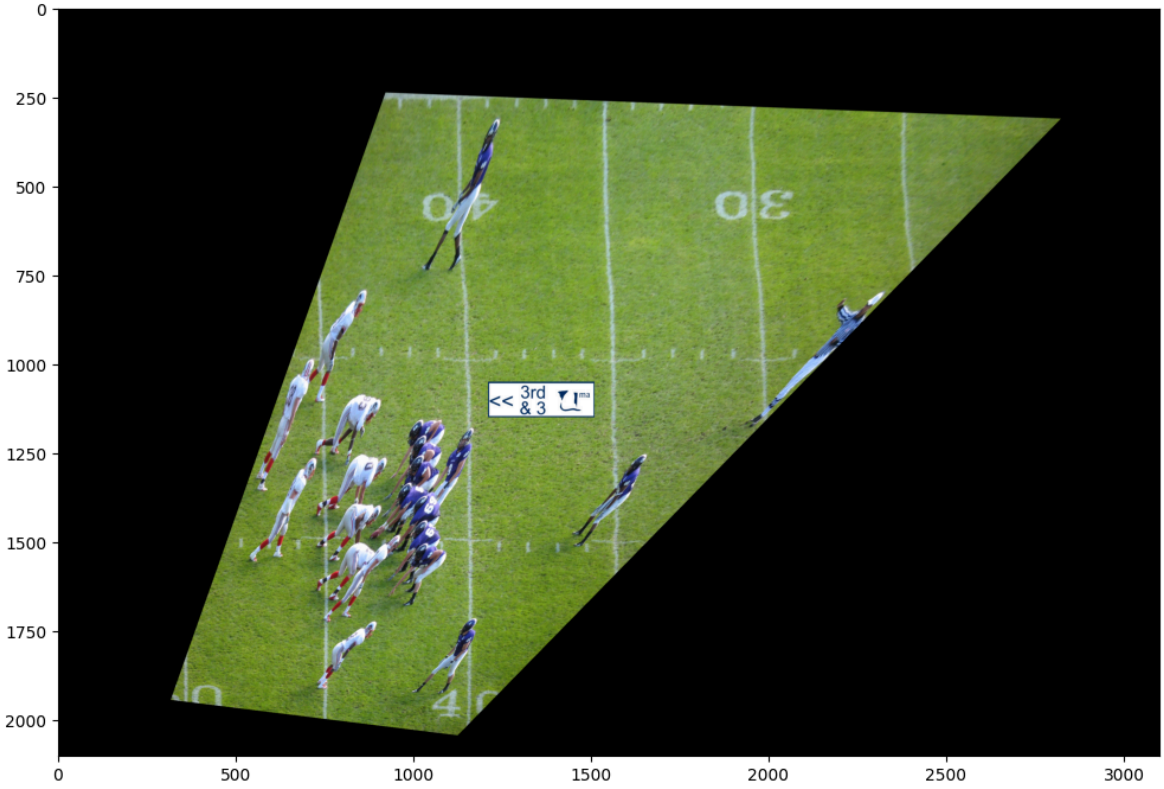
As we want to **add the start and down line in the original image**, we need to **take two points for each line in this image** (note that now the lines are vertical lines, so take points with the same x coordinate for both lines). In the next exercise we will apply inverse homography to those points so it will be possible to draw the line in the original image. **The down line should be at 3 yards of the start line** (each small vertical segment marks one yard, and vertical lines mark five yards).

ASSIGNMENT 2b: Adding the marker

Add the marker to the image without perspective and show it, also take the points mentioned before. The resulting image should look like this:



In [9]:
```
# Write your code here!

# Read the marker image.
marker = cv2.imread(images_path + 'marker.jpg')
marker = cv2.cvtColor(marker,cv2.COLOR_BGR2RGB)

# Get width and height of the marker
h,w = marker.shape[:2]

# Choose position of marker
x, y = 1210, 1050

# Place marker in football field image without perspective
no_perspective[y:y+h, x:x+w] = marker

# Show image
plt.imshow(no_perspective)

# Pick two homogenous points contained in the start line
start_line = np.array([[910,910],[1750,750],[1,1]])

# Pick two homogenous points contained in the down line
down_line = np.array([[675,675],[1800,1500],[1,1]]);
```



Augmented reality

It's time to restore the initial perspective, for that **apply an homography using the inverse of the transformation matrix**:

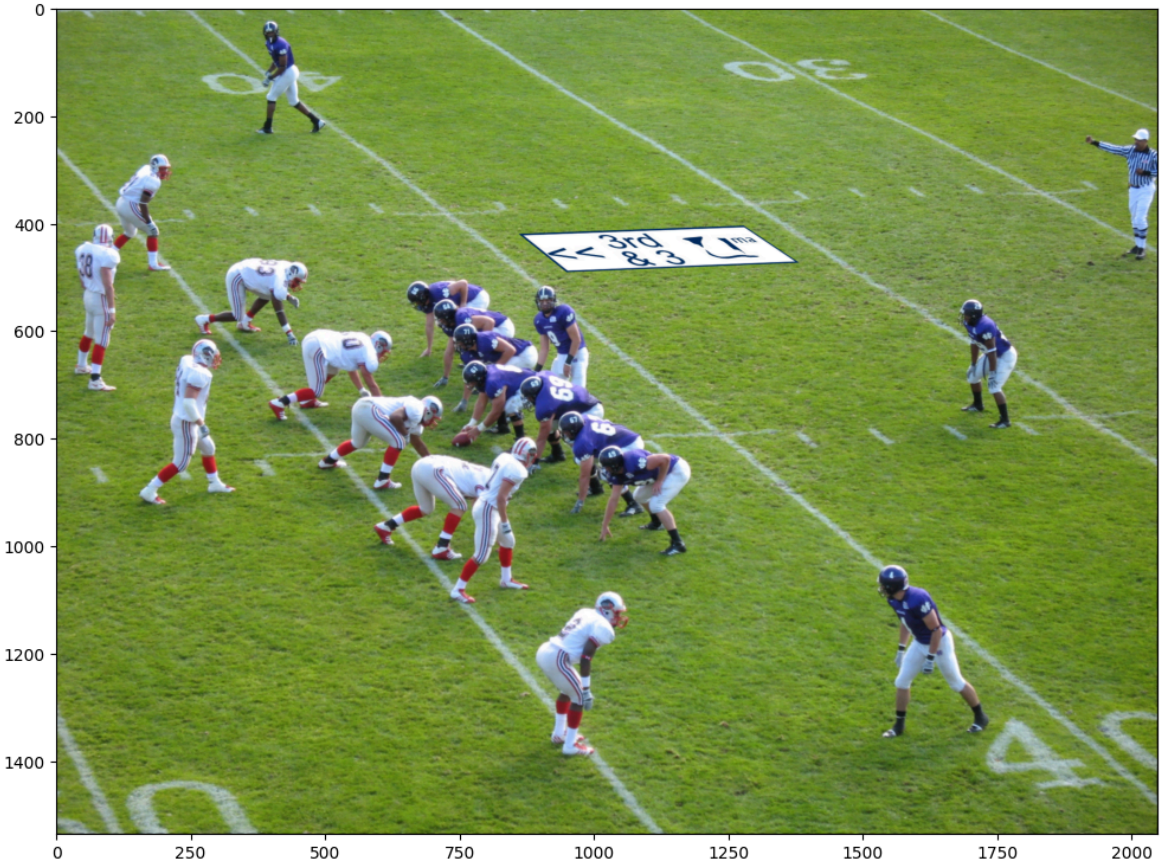
$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{bmatrix}^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

ASSIGNMENT 2c: Going back to the initial perspective, but with the mark!

Compute the inverse of the transformation matrix using `numpy.linalg.inv()`, and bring back the perspective to the image with the marker using the new homography. Then show it.

Now, you should see the marker having the same perspective than the original image.

In [10]:
```
# Write your code here!

# Compute the inverse of the homography
invM = np.linalg.inv(M)

# Get the original perspective again
n_rows = image.shape[0]
n_cols = image.shape[1]
perspective = cv2.warpPerspective(no_perspective, invM, (n_cols,n_rows))

# Show the image
plt.imshow(perspective);
```



Time to include the lines

Finally, its time to **draw the lines**. You should have saved the coordinates of two points for each line in the image without perspective.

For homography of lines, you just need to **apply the homography to two points of the line**. In that way, you will have two points in the original image defining the line.

Note that `cv2.line()` has two points as input, but only draws the segment line. You will have to **extend that segment** using the slope of the line.

$$slope = \frac{y_2 - y_1}{x_2 - x_1}$$

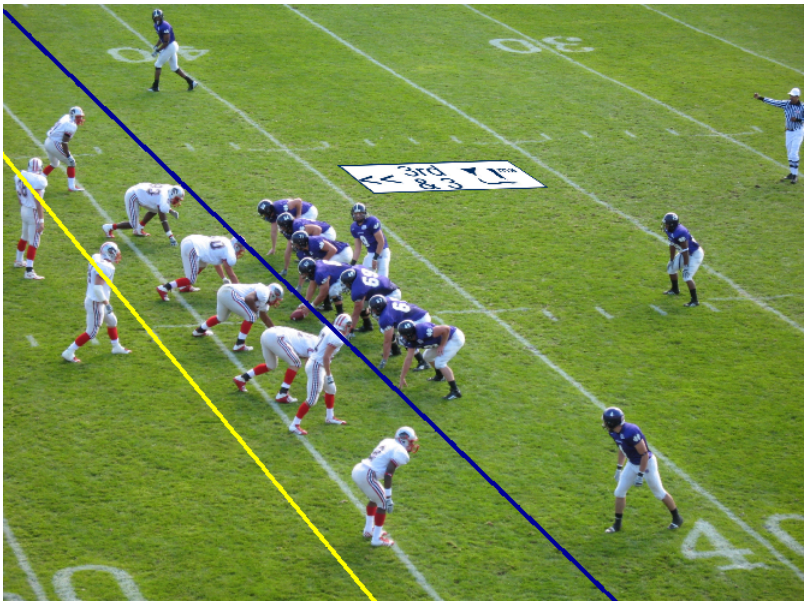
The new points will be:

$$x_1', \ y_1' = x_1 + extension, \ y_1 + extension \times slope \quad (3)$$

$$x_2', \ y_2' = x_2 - extension, \ y_2 - extension \times slope \quad (4)$$

ASSIGNMENT 2d: Drawing lines

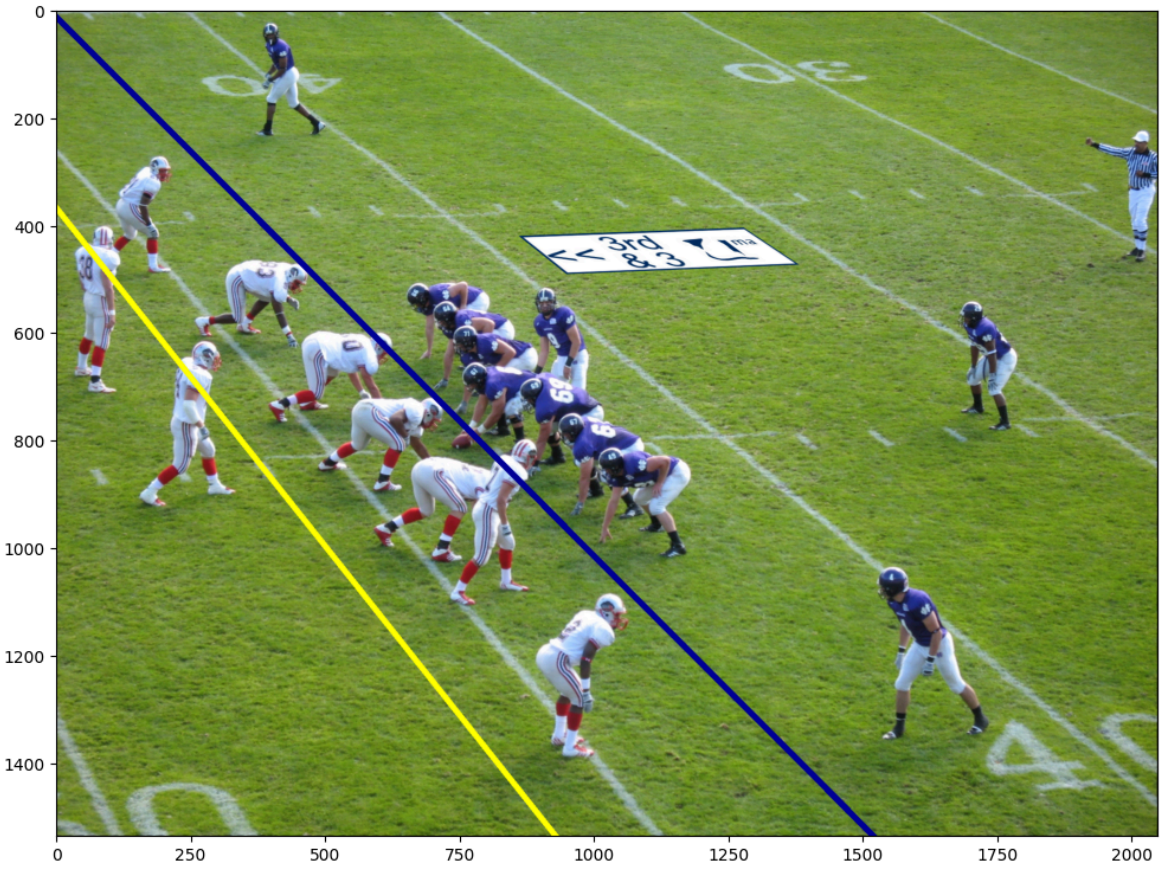
Apply the **inverse homography** to the line points, **extend** the line segment and **draw** the start line (blue) and the down line (yellow). Finally, show the resulting image, which should look like this:



In [11]:

```
# Write your coude here!

# Apply inverse homography to the start line points taken in the image without p
start_line_perspective = invM @ start_line
start_line_perspective = (start_line_perspective / start_line_perspective[2])[0:

x1,y1 = start_line_perspective[0,0], start_line_perspective[1,0]
x2,y2 = start_line_perspective[0,1], start_line_perspective[1,1]

# Compute the slope for extending the line segment
slope = (y2-y1)/(x2-x1)

# Extend the line segment
ext = 3000
x1,y1 = int(x1+ext), int(y1+ext*slope)
x2,y2 = int(x2-ext), int(y2-ext*slope)

# Draw the start line
pers_with_lines = perspective.copy()
pers_with_lines = cv2.line(pers_with_lines, (x1,y1), (x2,y2),thickness=10,color=

# Apply inverse homography to the down line points taken in the image without pe
down_line_perspective = invM @ down_line
down_line_perspective = (down_line_perspective / down_line_perspective[2])[0:2]

x1,y1 = down_line_perspective[0,0], down_line_perspective[1,0]
x2,y2 = down_line_perspective[0,1], down_line_perspective[1,1]

# Compute the slope for extending the line segment
slope = (y2-y1)/(x2-x1)

# Extend the line segment
x1,y1 = int(x1+3000), int(y1+3000*slope)
x2,y2 = int(x2-3000), int(y2-3000*slope)

# Draw the start line
pers_with_lines = cv2.line(pers_with_lines, (x1,y1), (x2,y2),thickness=10,color=
```

```
# And... show the final image!
plt.imshow(pers_with_lines);
```



A second way when dealing with lines: using its equation

An alternative way to apply an homography to a line is by considering its equation:

$$ax + by + c = 0$$

So $l = [a, b, c]^T$. In this way, a vertical line in the image without projection would be represented by a vector in this form $l' = [1, 0, -x]^T$, so we just need the x coordinate (column) to build such line!

Once we have the l' line, the same homography used for transforming points can be used here for obtaining its equivalent in the image with projection:

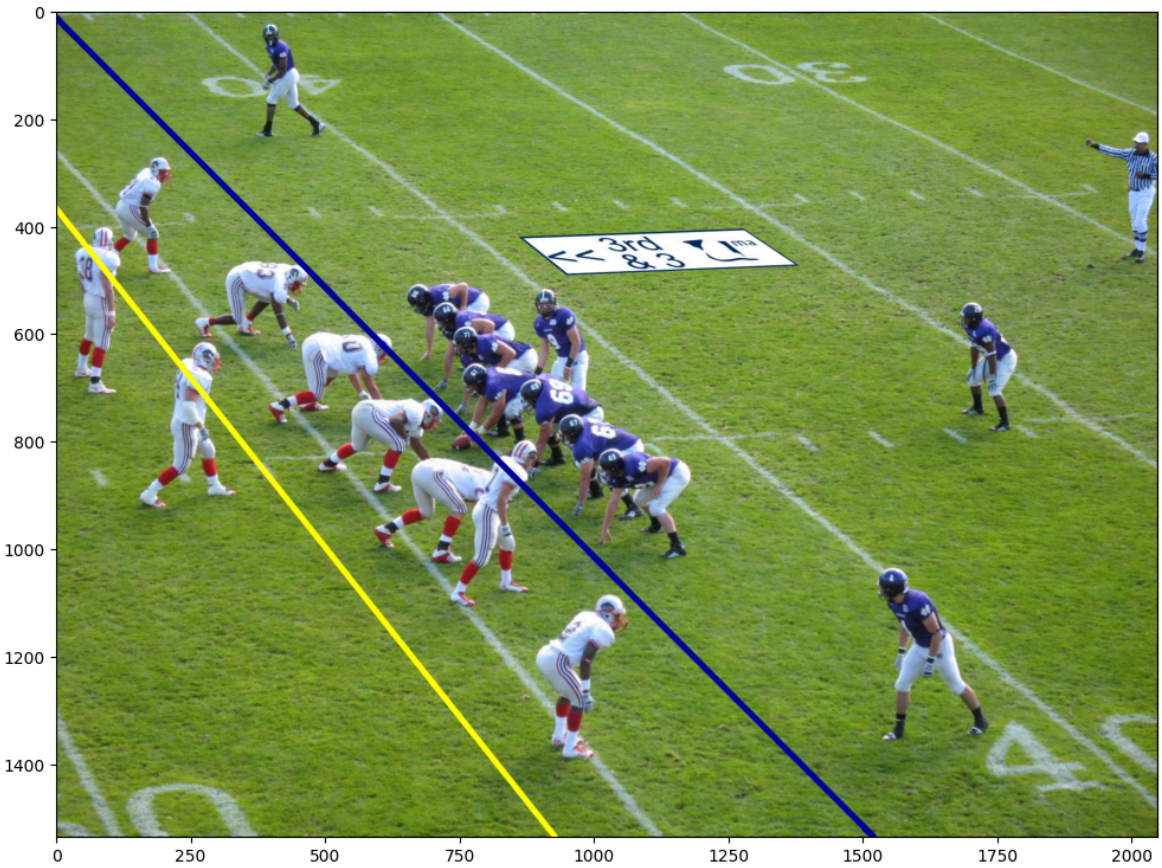
$$l = H^T l'$$

Amazing, isn't? Take a look at the following code which carries out this computations and provides the same results as before.

In [12]:
```
pers_with_lines = perspective.copy()

x1 = start_line[0,0]

l_h = np.vstack(np.array([1,0,-x1]))

l = M.T@l_h
n_y1 = 0
n_y2 = pers_with_lines.shape[0]
n_x1 = (-l[1]*n_y1-l[2])/l[0]
```

```
n_x2 = (-l[1]*n_y2-l[2])/l[0]

pers_with_lines = cv2.line(pers_with_lines, (int(n_x1),int(n_y1)), (int(n_x2),in


x1 = down_line[0,0]

l_h = np.vstack(np.array([1,0,-x1]))

l = M.T@l_h
n_y1 = 0
n_y2 = perspective.shape[0]
n_x1 = (-l[1]*n_y1-l[2])/l[0]
n_x2 = (-l[1]*n_y2-l[2])/l[0]
pers_with_lines = cv2.line(pers_with_lines, (int(n_x1),int(n_y1)), (int(n_x2),in

plt.imshow(pers_with_lines);
```

```
C:\Users\vital\AppData\Local\Temp\ipykernel_6776\500988625.py:13: DeprecationWarn
ing: Conversion of an array with ndim > 0 to a scalar is deprecated, and will err
or in future. Ensure you extract a single element from your array before performi
ng this operation. (Deprecated NumPy 1.25.)
  pers_with_lines = cv2.line(pers_with_lines, (int(n_x1),int(n_y1)), (int(n_x2),i
nt(n_y2)),thickness=10,color=[0,0,155])
C:\Users\vital\AppData\Local\Temp\ipykernel_6776\500988625.py:25: DeprecationWarn
ing: Conversion of an array with ndim > 0 to a scalar is deprecated, and will err
or in future. Ensure you extract a single element from your array before performi
ng this operation. (Deprecated NumPy 1.25.)
  pers_with_lines = cv2.line(pers_with_lines, (int(n_x1),int(n_y1)), (int(n_x2),i
nt(n_y2)),thickness=10,color=[255,255,0])
```



Thinking about it (1)

Now you are in a good position to answer these questions:

- Could you have retrieved the homography M by using 3 correspondences between points in assignment 2? Why?

  No, we could not. This is due to the fact that to achieve a projective transformation we need at least 4 correspondences, as we have 8 unknown values of the transformation matrix and each correspondence gives us 2.

- Could you have retrieved the homography M by using 5 correspondences between points in assignment 2? If yes, how?

  Yes, we could. Even though the new equation system would be overdetermined, we can get the solution by minimising the error in the points coordinates (least-squares solution).

- Why did we need to apply a projective transformation in our American football application in assignment 2? Could it be replaced by any other?

  We needed to apply a projective transformation since it is the only one which applies perspective transformation, which is what the task was about.

- What kind of transformation does the following matrix define?

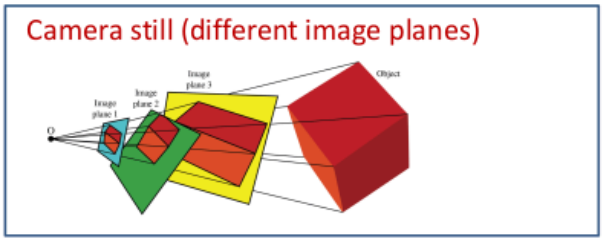
```
[[  1.06066017  -1.06066017 450.        ]
[  1.06066017   1.06066017 -75.        ]
[  0.           0.           1.        ]]
```

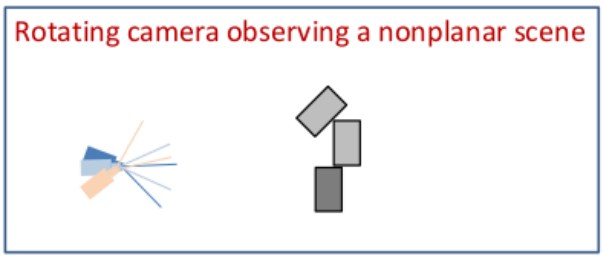
  This transformation matrix defines a similarity transformation, as it is the result of multiplying the rotation (and translation, but you can not distinguish this from euclidean transformation) by a scale.

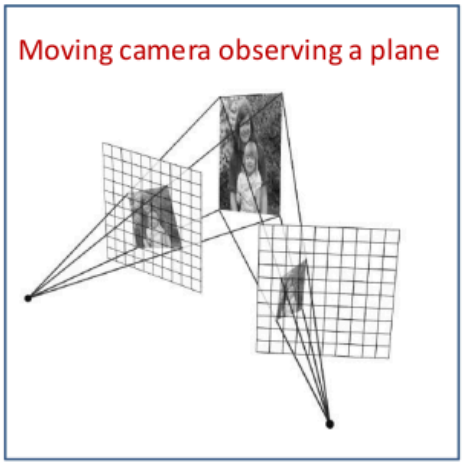
8.3.3 When can a homography be applied?

We have applied a homography to this football field because it is a transformation **between planes**, but if you look to the top player in the *aerial* image, you can see that the transformation is not good at all. So, we need to know when can we apply a homography.

It can be said that there is a homography $H_{3\times3}$ between points in these 3 cases:





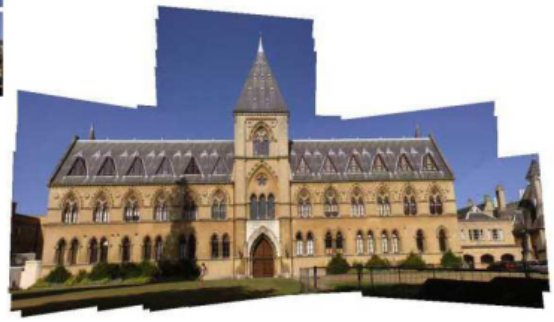


More detail next

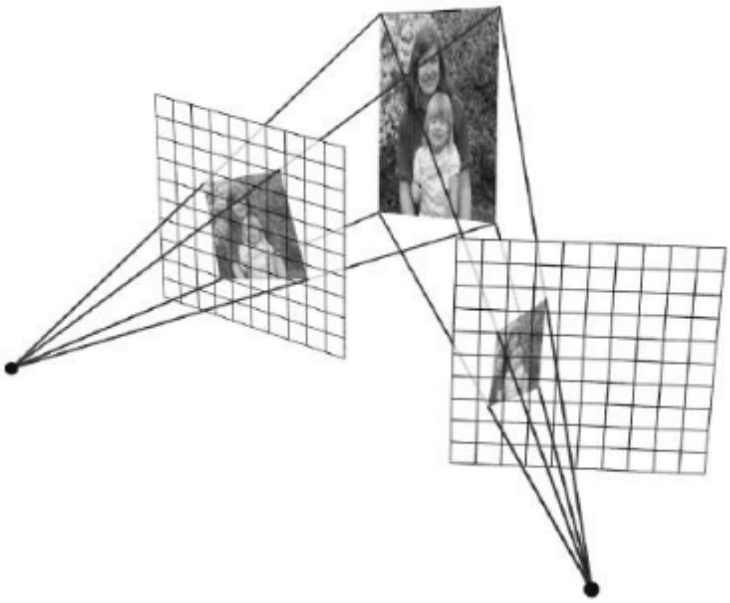
In this exercise we have emulated a **still camera** and we have transformed an image plane to another. Since the players are not in the same plane that the football field, the homography is not valid for those points.

On the other hand, there is also a homography **for a rotating camera observing a non-planar scene**. Actually, this would be the next step for **image stitching** after computing matches with keypoints in the example application in Chapter 4. We have already the correspondences either using Harris or SIFT, so the homography should be easy to obtain.

It doesn't mind if the scene is not planar, as long as the focal point remains still (or negligible movement in comparison to the scene distance)



Finally, there are a homography for a **moving camera observing a plane**. This is the typical case of 2 images of a painting taken from different positions:



Conclusion

Fantastic work! Homographies are a very important tool in computer vision. In this notebook you have learned:

- what a homography is, how to solve it and how to apply it,
- when a homography can be used,
- how to apply homographies to lines,
- some homography applications that are currently being used in American football (and other sports too! think about placing advertising on a tennis or football courts).

8.4 The Camera model

Until now we have first focused in stating the mathematical tools required to understand general vector/coordinates transformations and then to apply them to perform different image transformations (homographies). In fact, remember that in previous notebooks, we have transformed `WORLD` coordinates to `CAMERA` coordinates and the other way around, but that is just a **3D to 3D** transformation. However, we have not properly addressed yet the process of transforming `WORLD` **coordinates (3D)** to `IMAGE` **coordinates (2D)** (and vice versa when possible), that is, how images are formed from real 3D objects. Well, this is the time, let's go for it!

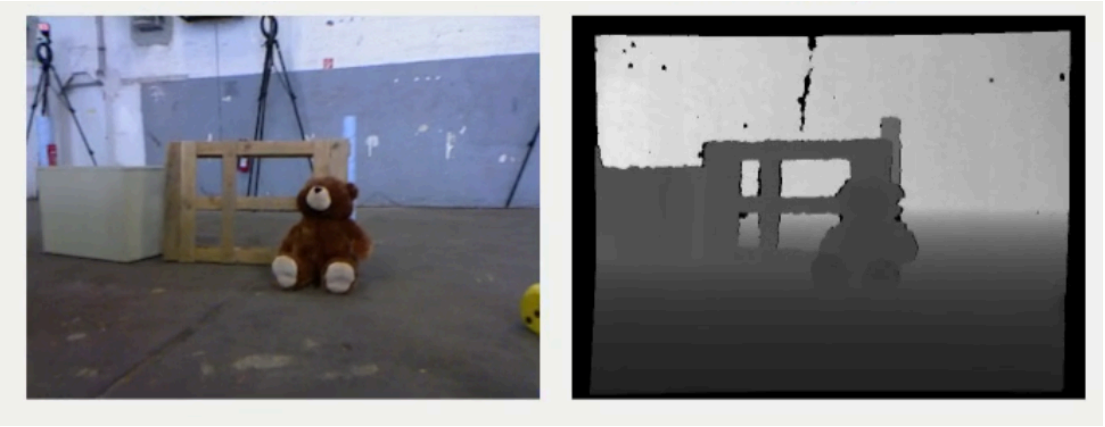
In this notebook we will learn:

- the **big picture** of the problem that we are addressing (section 8.4.1),
- the **Pinhole model** (section 8.4.2), and
- the **Camera model** (section 8.4.3).

Finally, we will put the learned concepts to work with a practical example (section 8.4.4).

Problem context - RGB-D images

RGB-depth (RGB-D) images are just like standard RGB images (where each pixel has information about each basic color: red, green and blue) but adding another information to each pixel: **the range/depth from the camera to the 3D point that is projected on it**. That is, these images contain not only photometric information but also geometric information.

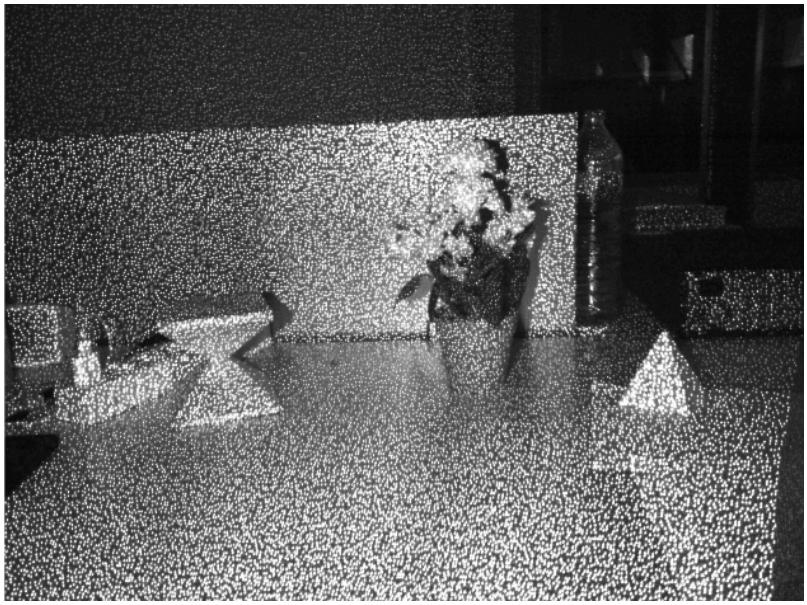


In this figure, the left image is the standard RGB image, while the right one is the *depth* band shown as a greyscale image (darker as the object is closer).

Although this kind of images have been used for some decades now, in recent years they have become popular due to the development of inexpensive sensors, such as the Microsoft's Kinect camera, that are able to directly provide them. In the entertainment

bussiness, these images have been used for segmenting the people using the system from the background of the image and inspecting their movements.

Roughly speaking, the working principle of these devices (called structured light) involves projecting an infrared pattern on the scene (see image below) and inspecting the deformations that such pattern undergoes due to the irregular surfaces where the light bounces. These deformations are directly related to the distance and position of the object.



As RGB-D images actually provide depth information, they are a **good example for learning how image formation models work**, because we can turn images to 3D maps (using the depth) and the other way around (3D scene to 2D image plane).

8.4.1 The big picture

The process of converting **world coordinates** to **image coordinates** involves several steps that are applied by means of the concatenation of transformations expressed with homogeneous matrices/vectors. We address all these transformations in this notebook but, first, let's present the whole process in a single image, just to keep an eye on **the big picture** and then we will unravel its mysteries.

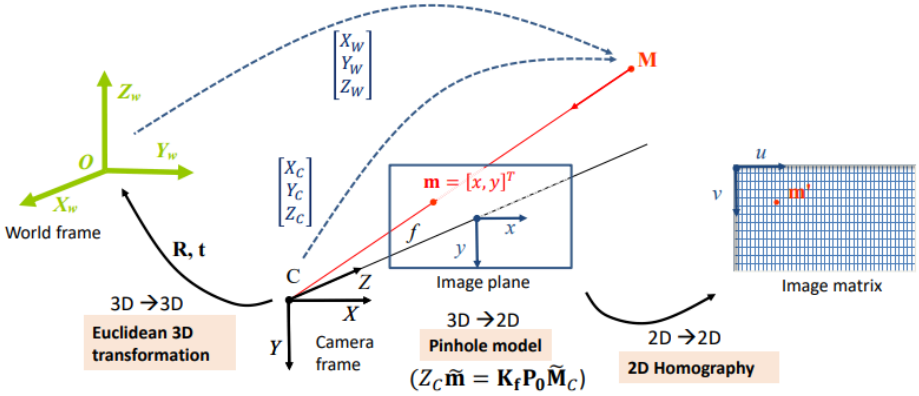


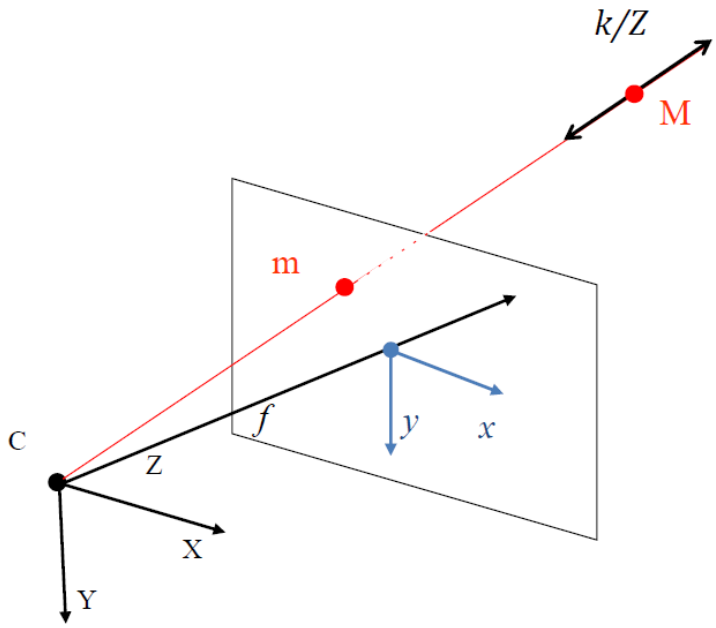
Fig. 1 - The big picture of the transformation involved in the conversion of a 3D point M from its world coordinates to the image ones m.

Let's do this step by step!

In [1]:
```
%matplotlib inline

import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
import scipy.stats as stats

from mpl_toolkits.mplot3d import Axes3D
from math import sin, cos, radians, floor
images_path = './images/'
matplotlib.rcParams['figure.figsize'] = (12.0, 12.0)

import sys
sys.path.append("..")
from utils.plot3DScene import plot3DScene
```

Homogeneous coordinates (again)

As we already know, an interesting property of them is that the **homogeneous coordinates of a point in the plane ($\mathbb{R}^2$) also represents a line passing through the origin in a reference frame parallel to the image plane**:



That is, for a certain Cartesian point $\mathbf{p} = (x, y)$, its homogeneous version $\tilde{\mathbf{p}} = (kx, ky, k), \forall k$ represents a line in 3D starting at the origin of coordinates. This is called a **projection line**.

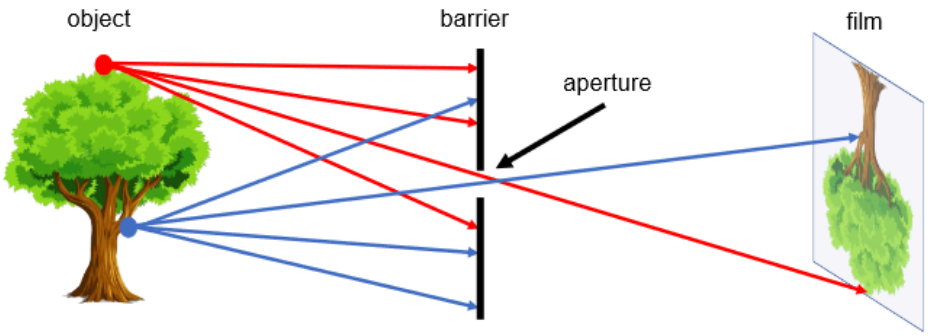
The **projective plane**, called $\mathbb{P}^2$, is the set of 3-tuples of real numbers such that $\begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \equiv k \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$, $k \neq 0$ (equivalent 3-tuples since they represent the same point in $\mathbb{R}^2$), that is the set of all projective lines at a certain distance k from the origin of coordinates.

So, in summary:

- A point in $\mathbb{P}^2$ (3-tuple) is represented in ($\mathbb{R}^3$) as a line passing through the origin.
- The component k can be understood as the *depth*, as it indicates a specific point along the line.

8.4.2 The Pinhole model

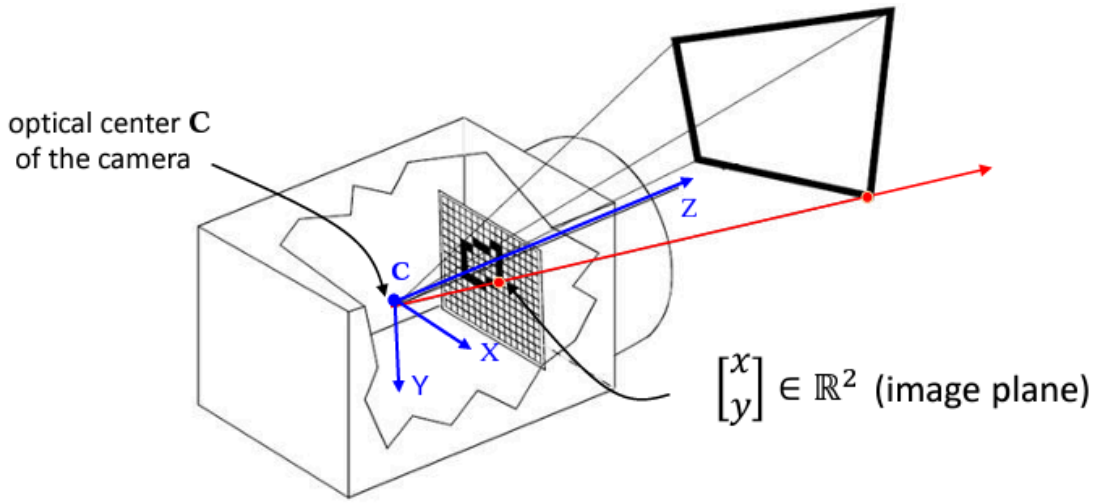
The Pinhole camera model is the simplest model we can think to understand how image formation works [1]. Let's image that we want to design a simple camera system that can record an image of an object in the 3D world. This camera system can be designed by placing a solid barrier with a small (pin-size) aperture between the 3D object and a photographic film or sensor.



As this figure shows, each point on the 3D object bounces the light from the source and emits multiple rays of such light outwards. Without a barrier in place, every point on the film will be influenced by light rays emitted from every point on the 3D object. However, due to the barrier, only one (or a few) of these rays of light passes through the aperture and hits the film. Therefore, we can establish a one-to-one mapping between points on the 3D object and the film. The result is that the film gets exposed by an *image* of the 3D object by means of this mapping. This simple model is known as **the pinhole camera model**.

In the Pinhole model we want to project the 3D world (a set of $\mathbf{p}_i = [X_i, Y_i, Z_i]^T$ points) in a plane called **image plane**. For that, we have a camera placed at $\mathbf{C} = [0, 0, 0]^{\mathrm{T}}$ in the `WORLD` reference system (i.e. both the `WORLD` and the `CAMERA` systems are coincident). This camera has a fixed property f called **focal length**, which indicates the distance between the the optical center and the **camera sensor** (placed in the interior of

the camera). The camera sensor is the plane **where the scene of the real world is projected**:



To operate with this image formation model, we need to know two processes:

- to project a 3D world point to the image plane
- to back-project a 2D point on the image plane to the 3D world.

For this, we are going to use the previously mentioned property of homogenous coordinates. Let's take a look at these processes!

From 2D to 3D

Given a point $\mathbf{p} = [x, y]^\intercal \in \mathbb{R}^2$ in the image, its **projection line** (in red in the figure above) in the camera system is very simple to compute by means of the line passing through the point $[x, y, f]^\intercal \in \mathbb{R}^3$ (we have added the third component f because the image plane is placed at a distance f in the Z-axis of the camera reference system, recall that this is the **focal length**!). In homogeneous coordinates, this line follows the expression:

$$k \begin{bmatrix} x \\ y \\ f \end{bmatrix} \in \mathbb{P}^2$$

where k indicates a specific 3D point along the projection line, and P is the so called **projective plane**.

For instance, if we set $k = 2$ we have the point:

$$\begin{bmatrix} 2x \\ 2y \\ 2f \end{bmatrix} \in \mathbb{R}^3$$

Note that, in this expression, k does not correspond directly to the depth of the 3D point, but we can fix this dividing by the focal length:

$$\underbrace{kf}_{Z}\begin{bmatrix} x/f \\ y/f \\ 1 \end{bmatrix} \in \mathbb{P}^2$$

Now that the third coordinate is 1, $k' = kf$ indicates the depth of the 3D point (that is, it's coordinate along the Z axis), so that if we know that the depth of a pixel is 5, we only have to set $k' = 5$ and we will have the 3D coordinates of the point.

From 3D to 2D

Let's have a look now at the process of projecting 3D points to 2D. Given any 3D point $\mathbf{M} = [X, Y, Z]^T \in \mathbb{R}^3$, we know that it has a projection point in the image plane $\mathbf{m} = [x, y]^T \in \mathbb{R}^2$.

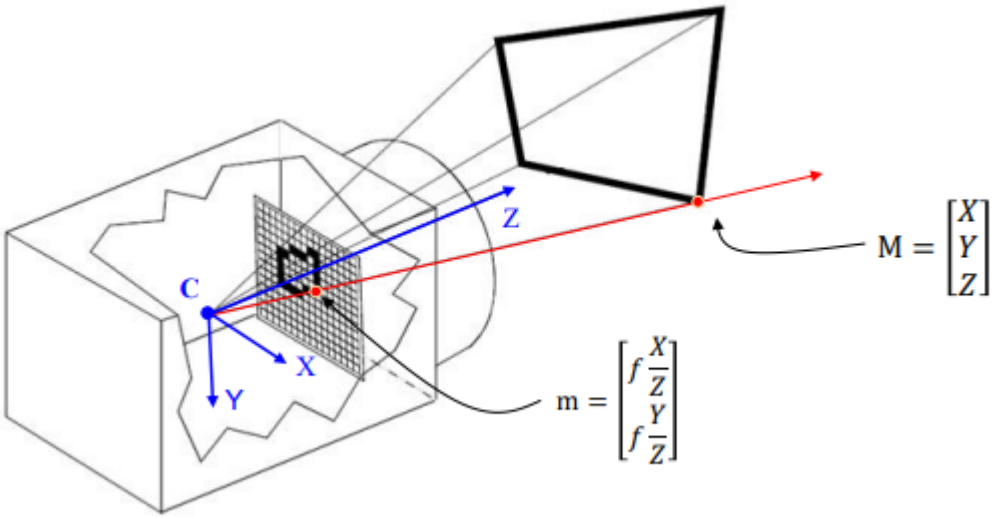
As seen before, the 3D point in the projection line of $\mathbf{m}$ with depth Z is:

$$Z\begin{bmatrix} x/f \\ y/f \\ 1 \end{bmatrix} = \begin{bmatrix} Zx/f \\ Zy/f \\ Z \end{bmatrix} \in \mathbb{R}^3$$

So that we can find the 2D image coordinates of the projected point through:

$$M = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} Zx/f \\ Zy/f \\ Z \end{bmatrix} \longrightarrow \quad \begin{aligned} X = \frac{Zx}{f} \;&\rightarrow\; x = \frac{fX}{Z} \\ Y = \frac{Zy}{f} \;&\rightarrow\; y = \frac{fY}{Z} \end{aligned}$$

This way, the relationship between the 3D point and its projected 2D point is like this:



Note that this transformation is not linear, but **it becomes linear if we use homogenous coordinates!**:

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} \xrightarrow{\text{homogenous to Cartesian}} \begin{bmatrix} fX / Z \\ fY / Z \end{bmatrix}$$

Summary

So, these equations relate 3D and 2D coordinates of a certain point in the real world and its projection in the image:

$$Z \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

3D - 2D: given $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \longrightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f\frac{X}{Z} \\ f\frac{Y}{Z} \end{bmatrix} \in \mathbb{R}^2$

2D - 3D: given $\begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{Z}{f} \begin{bmatrix} x \\ y \\ f \end{bmatrix} \in P^2$

Note that it is impossible to get the complete 3D position of a certain point from its coordinates in a single image because we need to know the Z coordinate beforehand. That is because the set of 3D points that falls in the line from the optical center to the image point **all of them project at the exact same point in the image**! This is also the reason why **we cannot determine the scale of the objects from a single image**, which is called the **scale indetermination** in monocular vision.

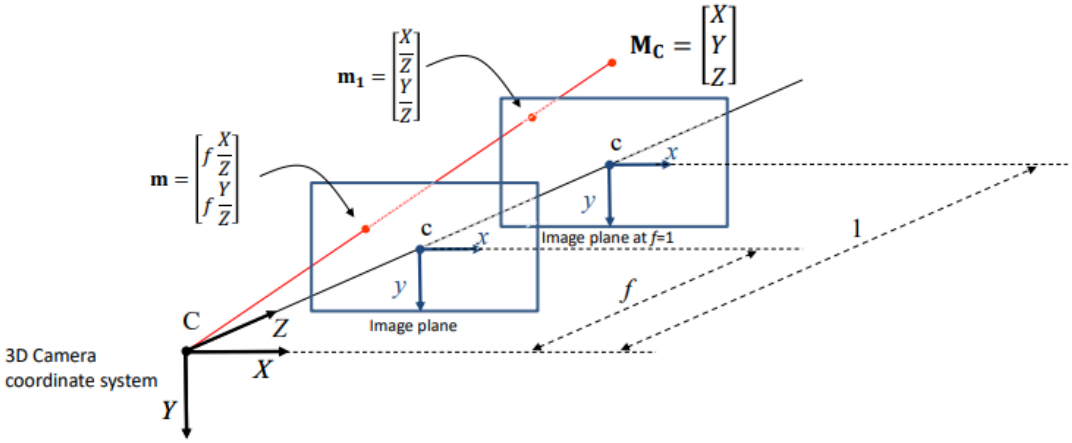
Usually, this transformation is made more general by decomposing it as

$$Z\tilde{\mathbf{m}} = \mathbf{K}_f \underbrace{\mathbf{P}_0 \tilde{\mathbf{M}}_C}_{Z\tilde{\mathbf{m}}_1}$$

, that is:

$$Z\underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{\tilde{\mathbf{m}}} = \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}_f} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{P}_0=[\mathbf{I}|\mathbf{0}]} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

In this expression, called **perspective projection equation**, $\tilde{\mathbf{m}}$ is the homogeneous 2D point, $\mathbf{K}_f$ is called the **calibration matrix** and $\mathbf{P_0}$ is a normalized projection ($f = 1$). The image below further illustrate these relations:



ASSIGNMENT 1: From world to image coordinates using the Pinhole model

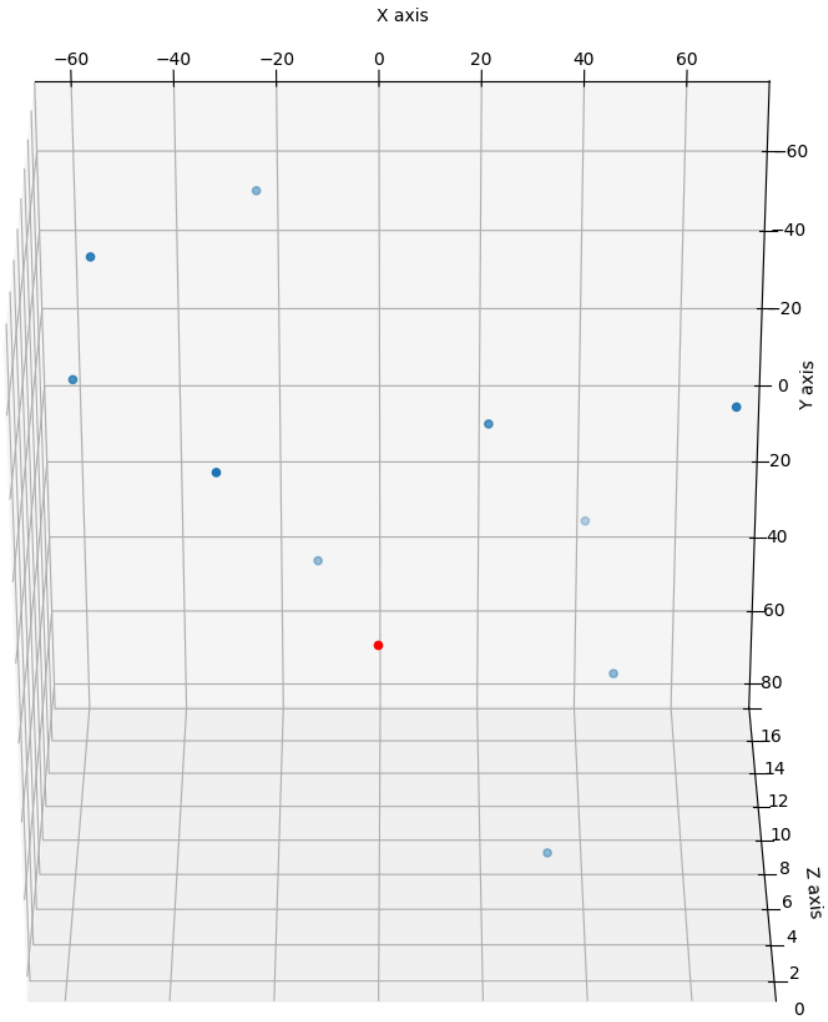
Your first task is to transform a number of points expressed in `WORLD` coordinates into image ones. You have to consider that:

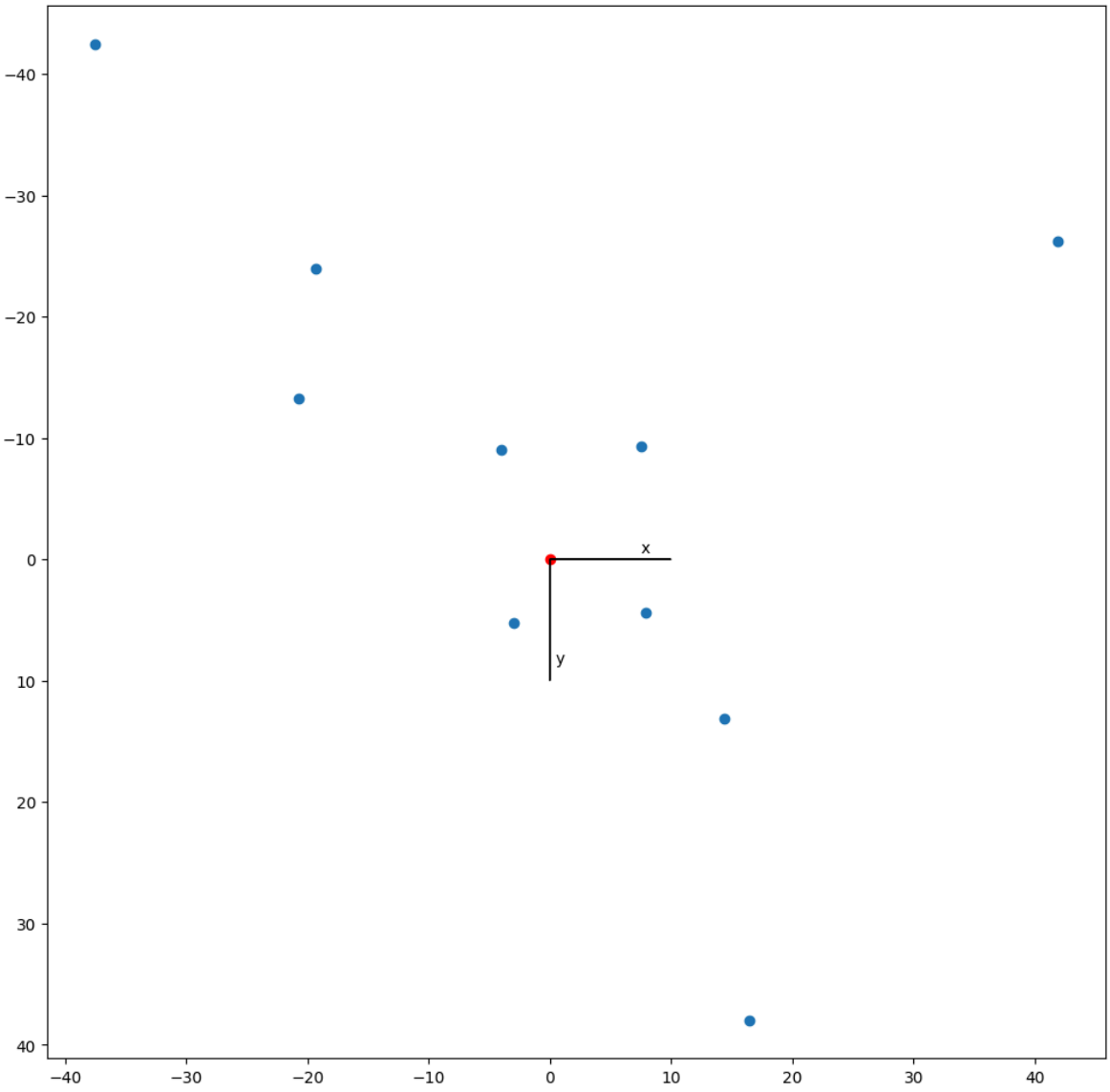
- The world points are given in the `world` matrix,
- both the `WORLD` and the `CAMERA` systems are coincident,
- the focal length of the camera is $f = 2.5$, and
- you should use **only** linear transformations and express the result in **Cartesian coordinates**.

In [2]:
```
# %matplotlib notebook
matplotlib.rcParams['figure.figsize'] = (12.0, 12.0)

# ASSIGNMENT 1 --
# World coordinates
world = np.array([[-12, 21,-30,41, 67,-54,33,-24,46,-58],  # X
                  [ 21,-26,-34,23,-42,-67,76,-54,42,-37],  # Y
                  [ 10,  7,  2,13,  4,  7, 5, 15, 8,  7]]) # Z

# world = np.array([[12,21,30,41,67,54,33,24,46,58],[21,26,34,23,42,67,76,54,42,
# world = np.array([[-12,-21,30,41,-67,54,-33,24,46,-58],[21,-26,34,-23,42,-67,7
```

```
# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = plt.axes(projection='3d')
# print(ax.dist)
# ax.view_init(azim=-60,elev=-30)

# Name axes
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

# Plot points
ax.scatter(world[0,:], world[1,:], world[2,:])
ax.scatter(0, 0, 0, color="red")
ax.view_init(elev=-60, azim=-90)

# Focal length
f = 2.5

# Convert world coordinates to homogenous
h_world = np.append(world, np.ones((1,world.shape[1])), axis= 0)

# Define transformation matrices
K_f = np.array([[f,0,0],[0,f,0],[0,0,1]])

P_0 = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])

# Get sensor homogenous coordinates (apply transformation matrices)
h_sensor = K_f @ P_0 @ h_world

# Transform them to cartesian
sensor = h_sensor[:2,:] / h_sensor[2,:]

# For checking if the result is right
print(sensor.round(3))

# Create figure
fig = plt.figure()

# Prepare figure
ax = fig.gca()

# Plot points
ax.scatter(sensor[0,:], sensor[1,:])
ax.scatter(0, 0, color="red")
ax.arrow(0,0,10,0)
ax.arrow(0,0,0,10)
ax.axis('equal')
ax.text(0.485, 0.38, 'y', horizontalalignment='center', verticalalignment='cente
ax.text(0.565, 0.485, 'x', horizontalalignment='center', verticalalignment='cent
ax.invert_yaxis() # to make the plot show the 'y' axis downwards
```

```
[[ -3.      7.5   -37.5     7.885  41.875 -19.286  16.5    -4.     14.375
  -20.714]
 [  5.25   -9.286 -42.5     4.423 -26.25  -23.929  38.     -9.     13.125
  -13.214]]
```


X axis
−60
−40
−20
0
20
40
60
Y axis
60
40
20
0
20
40
60
80
Z axis
16
14
12
10
8
6
4
2
0



Check if your results are correct:

Expected output:

```
[[ -3.      7.5   -37.5     7.885  41.875 -19.286  16.5    -4.
14.375  -20.714]
 [  5.25   -9.286 -42.5     4.423 -26.25  -23.929  38.     -9.
13.125  -13.214]]
```

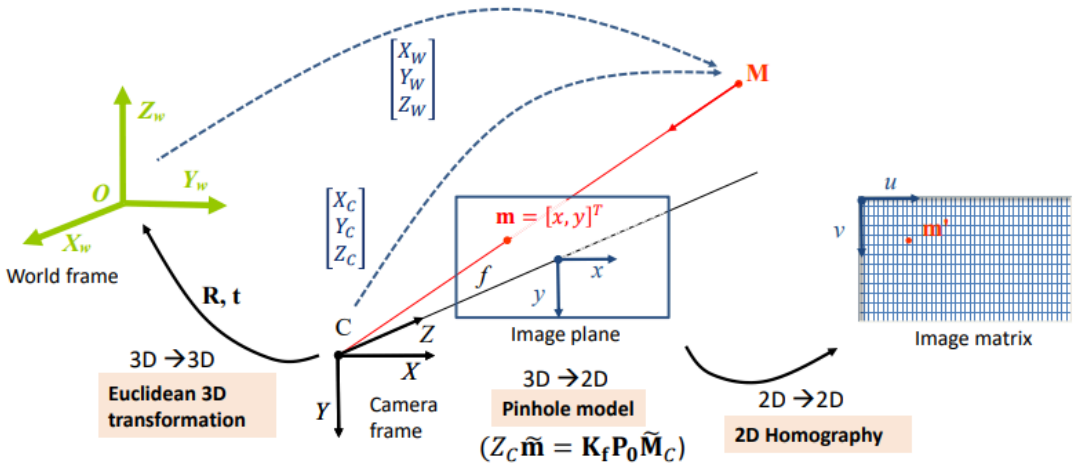
8.4.3 The Camera model

The Pinhole model is a useful starting point for understanding the processes of image formation, but it still has several limitations because:

- the camera frame point m has its coordinates expressed **in meters** within the camera sensor, but we want to know them in **pixels**.
- it assumes that the `CAMERA` and the `WORLD` reference systems **are coincident**, which, of course, is not generally the case.

This is where we define the **camera model**, which includes some linear transformations for solving these limitations and allows us to address the complete problem of image

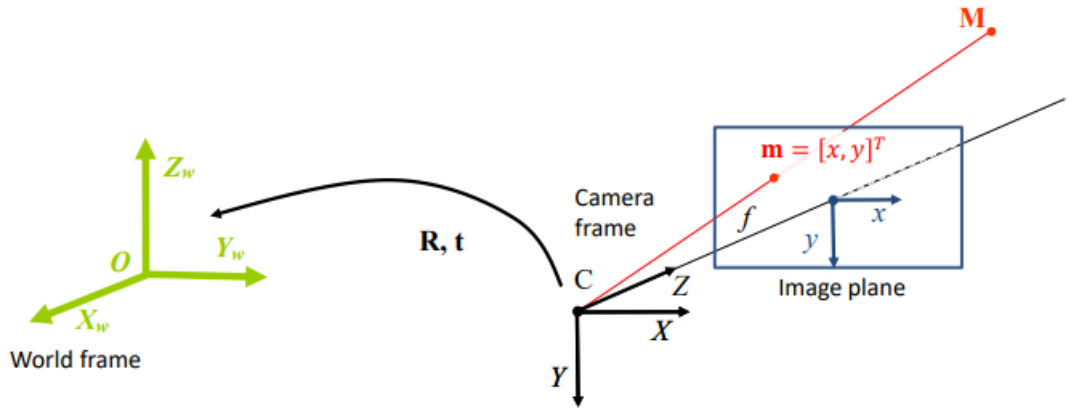
formation as depicted in the big picture shown before:



Let's go step by step!

Step 1: From the `WORLD` frame to the `CAMERA` frame

This transformation implies finding a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$ that **relates the position and orientation of the camera w.r.t. the world**. This is a 3D to 3D transformation and we have used it in the previous notebooks.



As said before, usually the camera is not at the world center, mainly because the camera will move within the world, or even because will be more than one camera in our perception system.

Remember that in the second notebook of this chapter, we learned how to apply a rotation and a translation to a set of 3D points using homogeneous transformations:

$$\mathbf{M}_C = \mathbf{R}\mathbf{M}_W + \mathbf{t} \xrightarrow{\text{In homogenous}} \tilde{\mathbf{M}}_C = \mathbf{D}\tilde{\mathbf{M}}_W$$

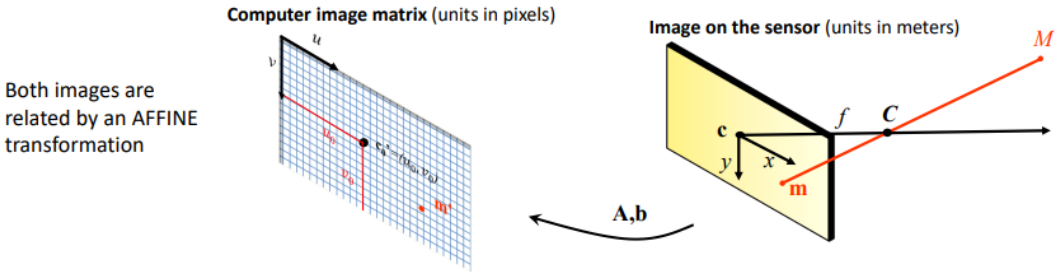
If we add this transformation to the Pinhole model, the new **perspective projection equation** takes the expression:

$$\lambda\tilde{\mathbf{m}} = \mathbf{K}_f\mathbf{P}_0\tilde{\mathbf{M}}_C = \mathbf{K}_f\mathbf{P}_0\mathbf{D}\tilde{\mathbf{M}}_W$$

which allow us to transform points in 3D within the `WORLD` reference system (in meters) to points in 2D in the image sensor (**also in meters!**).

Step 2: From the sensor image to a computer image

Once we have obtained the sensor image points, we want to get the **image matrix coordinates** of such points **in pixels** (that is, *row and column* within the image matrix), which are the units we use in the computer in all the artificial vision processes we have studied in this book.



As you can see in the figure, both images are related by an **affine** transformation (i.e. a 2D homography as seen in the previous notebook). As we know, affine transformations allow **rotation + translation + scale** (different for each axis). In this case we don't need any rotation, but axes **do scale**, and such scale represents the size in meters in the sensor of each pixel in the image:

$$\mathbf{m}' = \mathbf{A}\mathbf{m} + \mathbf{b} = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix} \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{\text{meters}} + \underbrace{\begin{bmatrix} u_0 \\ v_0 \end{bmatrix}}_{\text{pixels}} \xrightarrow{\text{In homogenous}} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} k_x & 0 & u_0 \\ 0 & k_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Note that in this expression, k_x and k_y determine the scale and have units of **pixels/meter**, while u_0 and v_0 are the coordinates of the **principal point**, which is the projection of the Z axis in the image plane. These elements are part of the **intrinsic parameters** of the camera, which means that they are always **constant** for the same camera.

ASSIGNMENT 2: From image coordinates to pixels

It is time to transform the sensor coordinates (in meters) obtained in the previous assignment to proper image coordinates (in pixels). For this, you will need the intrinsic parameters of the camera, which for this example are:

- `k_x = 2`
- `k_y = 3`
- `u_0 = 300`
- `v_0 = 200`

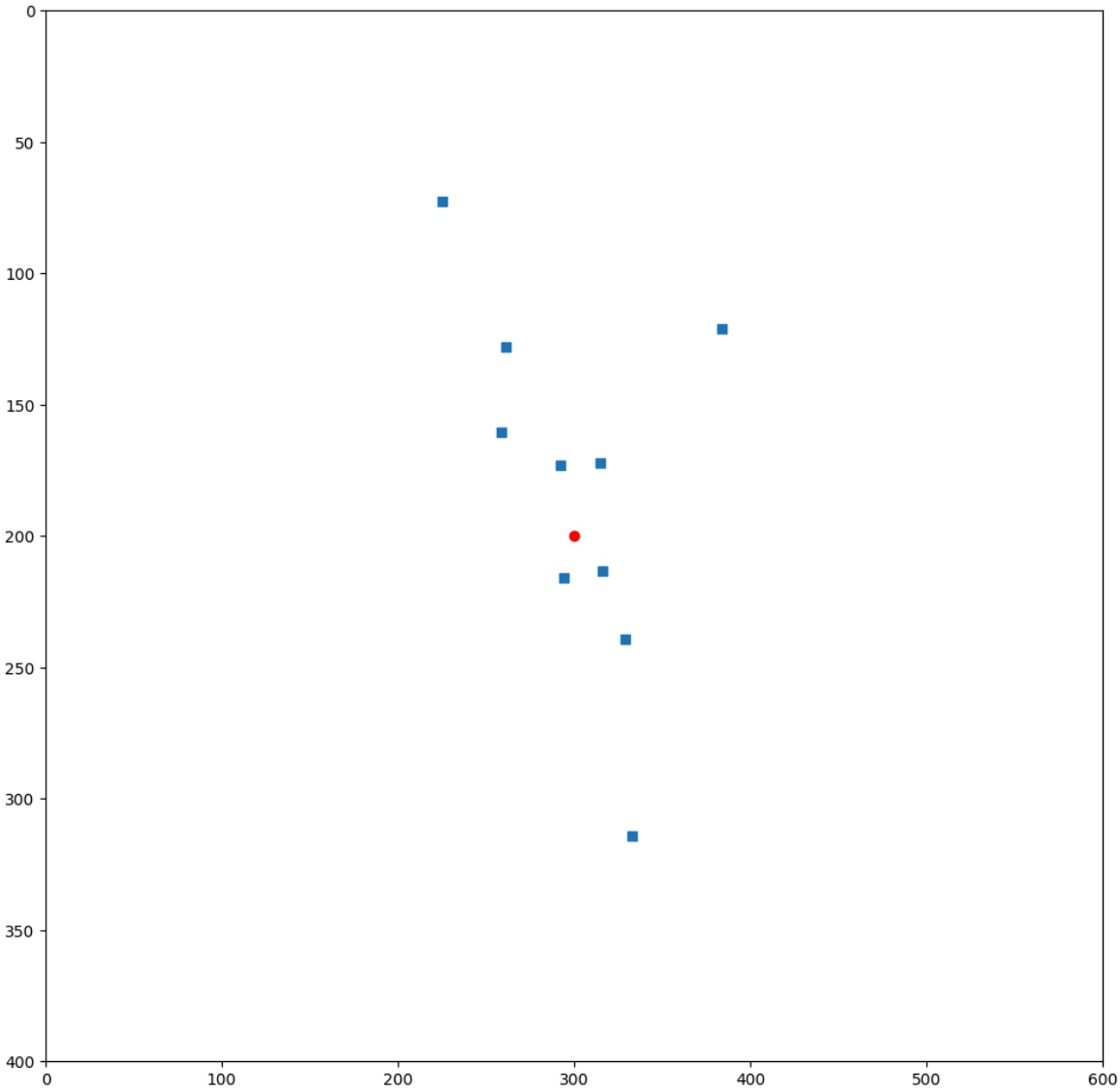
Don't worry, these values are normally provided by the manufacturer of the camera, although not everything is lost otherwise. We could also calculate them through a process called **camera calibration**, which we will address in the next chapter. Exciting!

Concretely, **you have to**:

- define the homography $\mathbf{K}_s$ (`H` in the code) relating meters with pixels,
- apply it to the sensor coordinates in `h_sensor` as retrieved in the previous assignment,
- transform the resulting coordinates to Cartesian ones, and
- show them!

In [3]:
```
# ASSIGNMENT 2 --
# Define intrinsic parameters
kx = 2
ky = 3
u0 = 300
v0 = 200

# Write your code here!

# Transformation matrix
H = np.array([[kx,0,u0],[0,ky,v0],[0,0,1]])

# Transform sensor coordinates to image coordinates
h_image_pixels = H @ h_sensor
center = H @ np.array([0,0,1])

# Transform to cartesian
image_pixels = h_image_pixels[:2,:] / h_image_pixels[2,:]

# For checking if the result is right
print(image_pixels.astype(int))

# Create figure
fig = plt.figure()

# Prepare figure for 3D data
ax = fig.gca()

# Plot points
ax.scatter(image_pixels[0,:], image_pixels[1,:], marker='s')
ax.scatter(center[0], center[1], color="red")

# Set axes limits
ax.set_xlim(0, 600)
ax.set_ylim(0, 400)
#ax.axis('equal')
ax.invert_yaxis() # to make the plot show the 'y' axis downwards
```

```
[[294 315 225 315 383 261 333 292 328 258]
 [215 172  72 213 121 128 314 173 239 160]]
```



You can **check if your results are correct**:

Expected output:

```
[[294 315 225 315 383 261 333 292 328 258]
 [215 172  72 213 121 128 314 173 239 160]]
```

General form of the Perspective matrix

Let's see how the final perspective transformation would be **from 3D world frame to computer image**:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} k_x & 0 & u_0 \\ 0 & k_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{from sensor to image}} \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{from camera frame to sensor}} \underbrace{\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0_3^T} & 1 \end{bmatrix}}_{\text{from world to camera frame}} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

Again, we can merge some transformations:

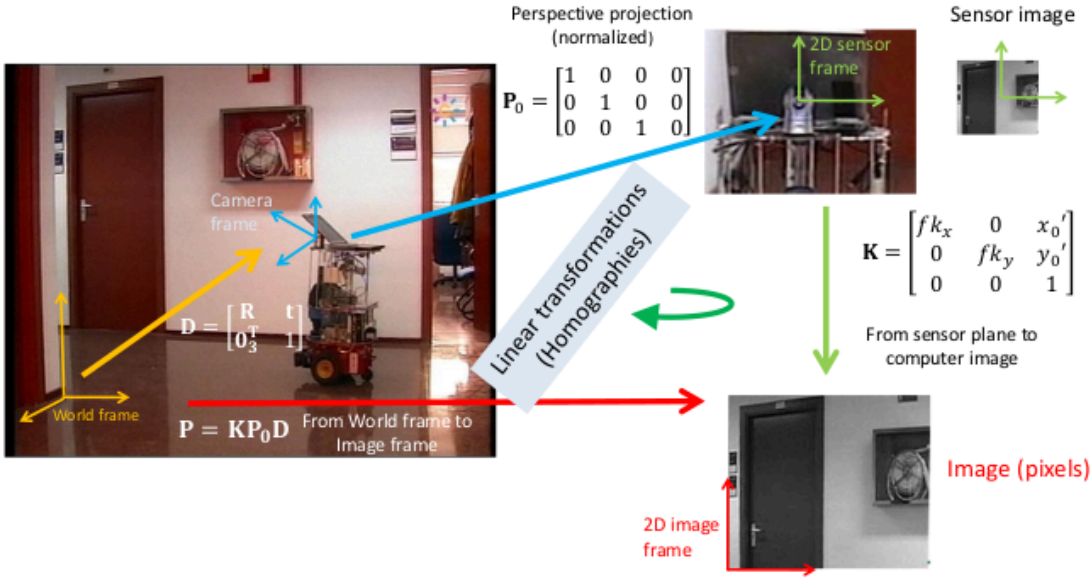
$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} fk_x & 0 & u_0 \\ 0 & fk_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix} \qquad \lambda \tilde{\mathbf{m}}' = \mathbf{K} \begin{bmatrix} \mathbf{R} \, \mathbf{t} \end{bmatrix} \tilde{\mathbf{M}}_{\mathbf{W}}$$

where

- $\mathbf{R}$ and $\mathbf{t}$ are **extrinsic** paremeters that depend on the camera position.
- f, k_x, k_y, u_0 and v_0 are **intrinsic** parameters (constant) that depend on the camera that is being used.

Note: you will usually find that f is directly expressed in pixels, so actually it will be the result of fk_x and fk_y, and furthermore, this is sometimes also called $fk_x = s_x$ and $fk_y = s_y$

Here you have the full camera model in action:



For a visual explanation of this model, you can check this interactive application.

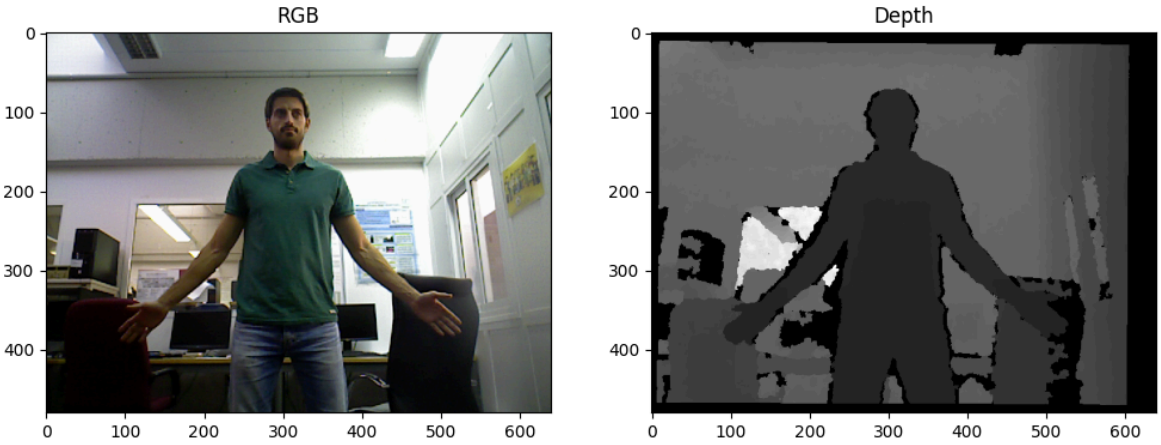
8.4.4 Putting things to work: the RGB-D image

As a practical exercise, we are going to apply the pinhole camera model to get a 3D set of points (also known as a **pointcloud**) from an RGB-D image. Remember that we said that it is not possible to determine the 3D position of a point from just its coordinates in a single image? Well, in this case we have **the depth information** included in the image and therefore we can now accomplish that! So, let's transform 2D points in the RGB image to 3D points in the world frame.

ASSIGNMENT 3a: Visualizing our RGB-D image

First of all, show the RGB-D image in a 2x1 subplot with the RGB part in the left (`person_rgb.png`) and the depth part in the right (`person_depth.png`) and see what they look like.

In [4]:
```
matplotlib.rcParams['figure.figsize'] = (12.0, 12.0)

# ASSIGNMENT 3a --
# Write your code here!

# Read images
image = cv2.imread(images_path + 'person_rgb.png',-1)
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
depth = cv2.imread(images_path + 'person_depth.png',0)

# Show RGB
plt.subplot(121)
plt.title("RGB")
plt.imshow(image)

# Show depth
plt.subplot(122)
plt.title("Depth")
plt.imshow(depth, cmap="gray");
```



From the RGB-D image to 3D coordinates

As you can see, the depth image represents the depth with levels of grey color, the darker it is a pixel, the closer to the camera it is. So, for each pixel **there is a mapping between intensity** $\rightarrow$ **depth**. Since we have the actual depth of the points, we just need to get the projection lines of the pixels in the image and then select the Z of the point according to the depth.

First, we need to transform the points in **image coordinates** to **sensor coordinates**. For this, we reverse the sensor to image method seen previously, either by using the pseudoinverse matrix or by **isolating the variables**.

We know that *(assuming that the scale is 1)*:

$$\textbf{2D} \rightarrow \textbf{2D} \qquad \underbrace{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}_{\textbf{pixels}} = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{\textbf{meters}}$$

If we isolate x and y, we get the coordinates in the sensor (note that since we are already dividing by f, this coordinates are expressed in a projective plane at $Z = 1$):

$$x = \frac{u - u_0}{f}, \; y = \frac{v - v_0}{f} \tag{3}$$

Finally, we add the depth component Z, which is available in the depth image:

$$\textbf{2D} \rightarrow \textbf{3D} \qquad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{Z} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

so $X = Zx$ and $Y = Zy$.

ASSIGNMENT 3b: Building a point cloud from an RGB-D image

Generate a $[3 \times N]$ matrix with the 3D camera frame coordinates of all pixels in the image `person_rgb.png` using these intrinsic parameters:

- `f = 525`
- `k_x = 1`
- `k_y = 1`
- `u_0 = 319.5`
- `v_0 = 239.5`

Notice that we are using a linear scale that discretizes the camera operating range, from 0 up to 5 meters, in order to encode such distances in the 256 pixel possible values of a greyscale image. In this way, `scale=0.02` (approx. $5m/256$) so, for example, if the value of a pixel in the depth image is 20, it means that its corresponding 3D point is `20*scale=0.4` meters away.

In [5]:
```
# Read images
image = cv2.imread(images_path + "person_rgb.png",-1)
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
depth = cv2.imread(images_path + "person_depth.png",0)

# Import intrinsic parameters
f  = 525
u0 = 319.5
v0 = 239.5

# Matrix for storing (homogeneous) sensor coordinates
h_sensor = np.zeros((3,image.shape[0]*image.shape[1]))

# Get sensor coordinates in meters
point = 0
```

```
for v in range(image.shape[0]): # rows
    for u in range(image.shape[1]): # columns
        # Transform to sensor coordinates
        h_sensor[0,point] = (u-u0)/f
        h_sensor[1,point] = (v-v0)/f
        h_sensor[2,point] = 1 # Homogeneous coordinates
        point += 1

scale = 0.02 # ~5m/256
image_d_fila = np.reshape(depth,(1,depth.shape[0]*depth.shape[1]))
image_d_fila = image_d_fila*scale

map_3d = h_sensor * image_d_fila # element-wise multiplication

# check coordinates at points 100060, 100061 and 100062
print(map_3d[:,100060:100063])
```

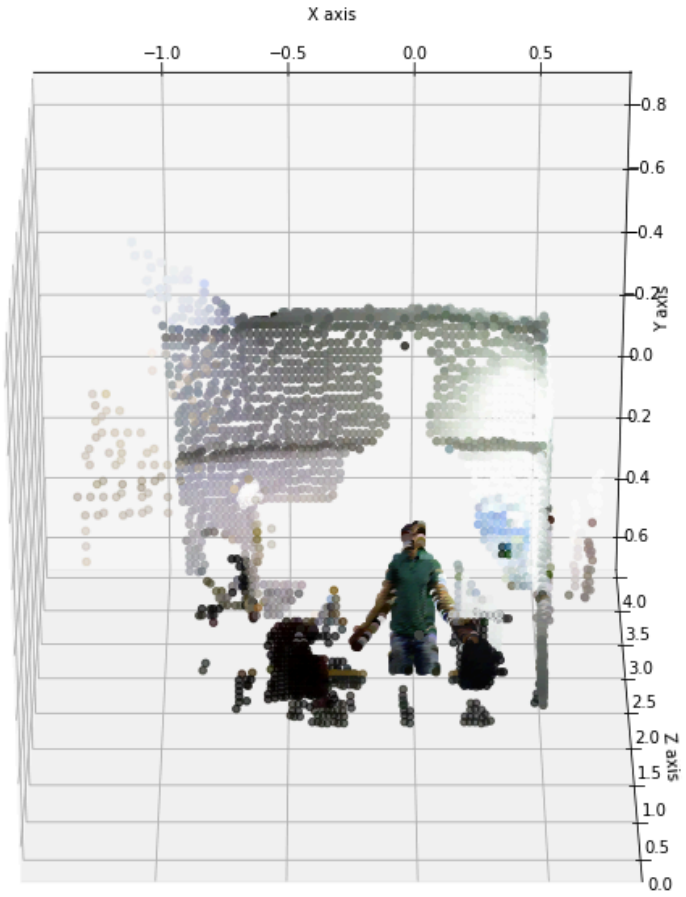
```
[[-0.30702857 -0.30394286 -0.30085714]
 [-0.25765714 -0.25765714 -0.25765714]
 [ 1.62        1.62        1.62      ]]
```

You can **check if your results are correct** (Positions of the points nº 100060, 100061 and 100062 in the camera frame):

Expected output:

```
[[-0.30702857 -0.30394286 -0.30085714]
 [-0.25765714 -0.25765714 -0.25765714]
 [ 1.62        1.62        1.62      ]]
```

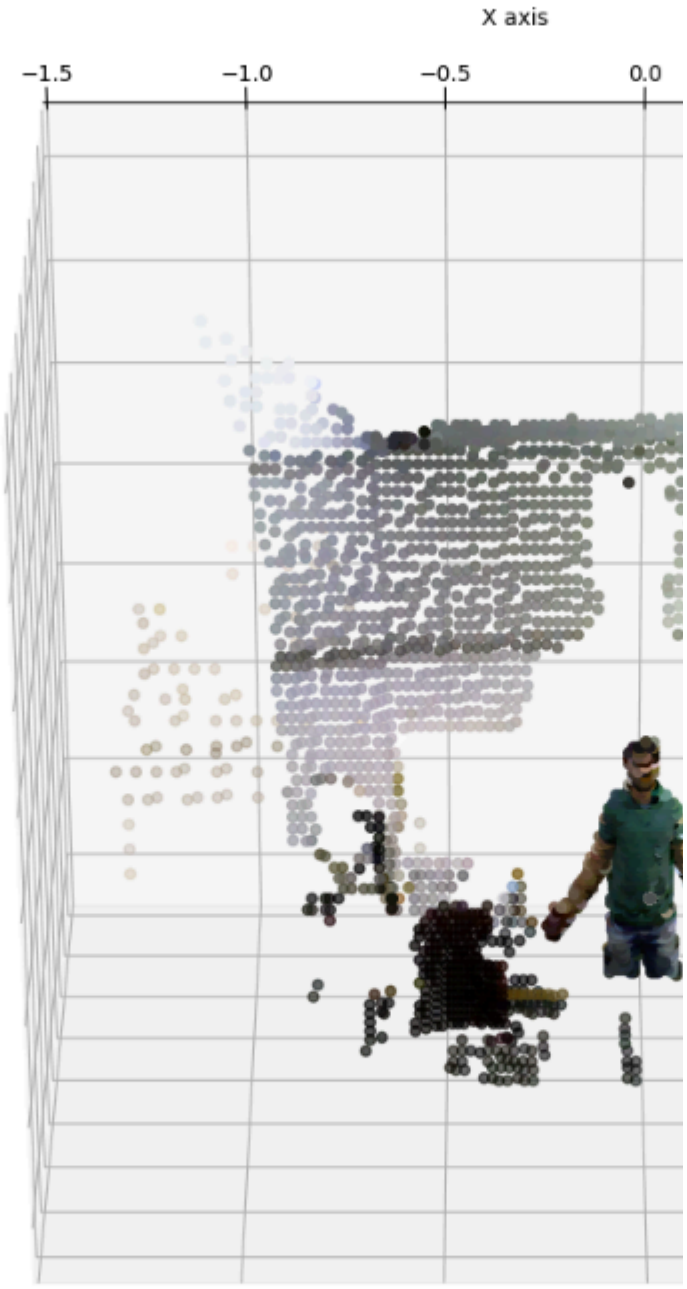
This is what you should get:



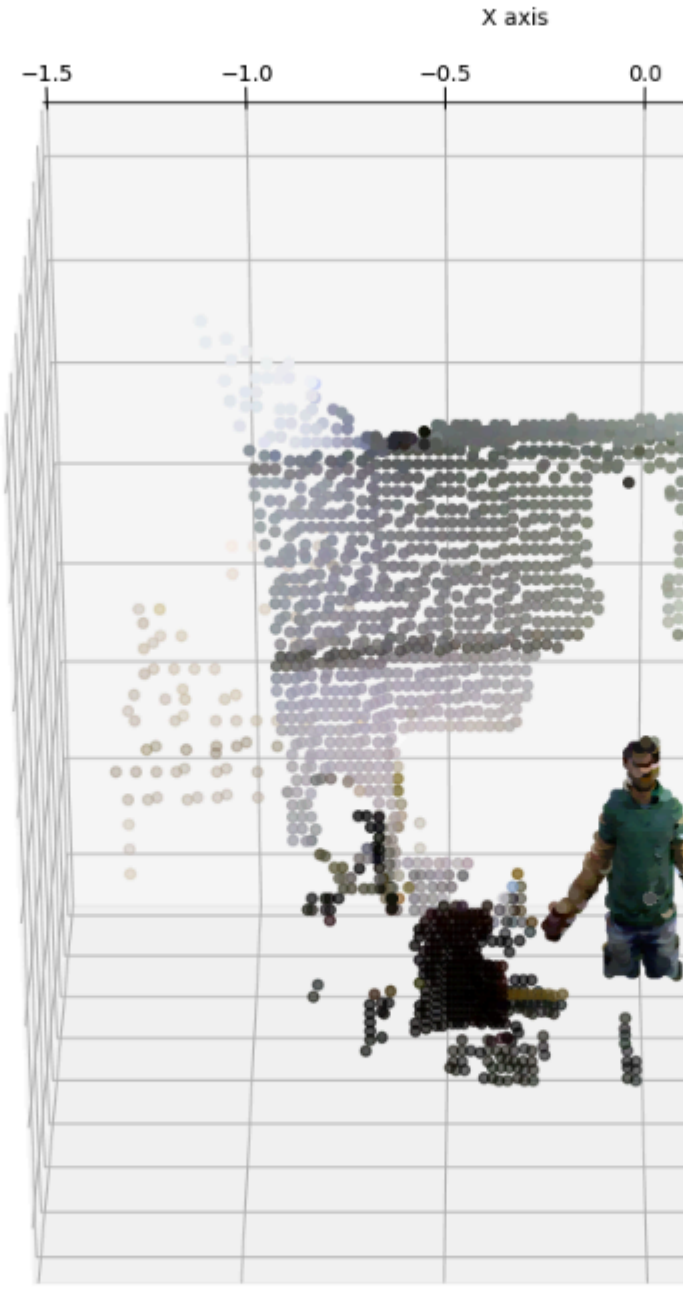
You can see how all the pixels in the image have been back-projected to 3D and we have a fancy pointcloud showing our scene in 3D. Now all those points are in the `WORLD` reference system (we assumed that `WORLD` and `CAMERA` reference systems were coincident) and we can move our camera in order to get an image from a different perspective. Let's go for it.

In [6]:
```
matplotlib.rcParams['figure.figsize'] = (12.0, 12.0)

# Let's draw the pointcloud!
%matplotlib widget
plot3DScene(map_3d,image)
```

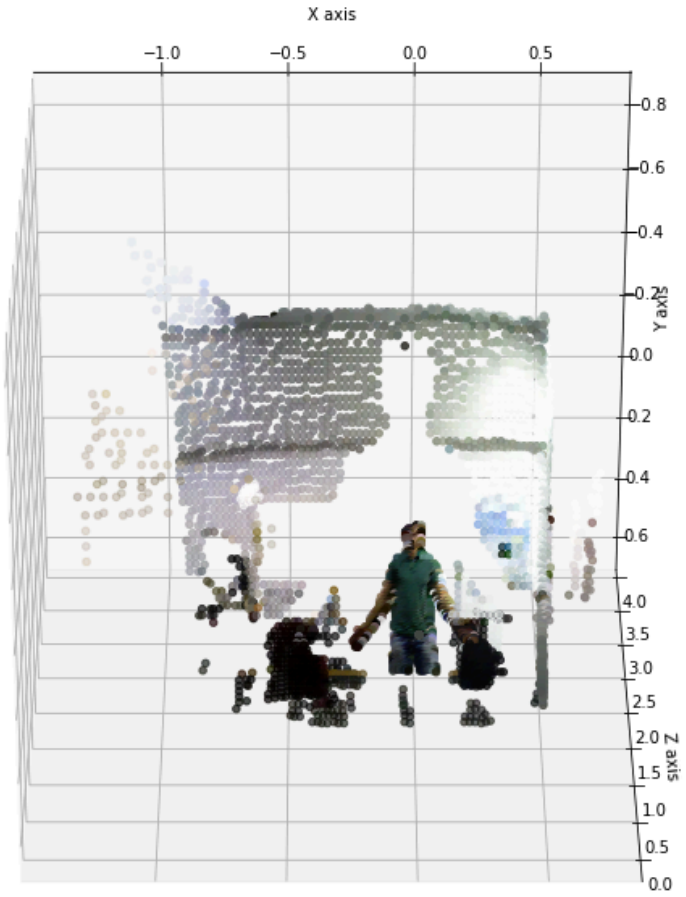
Figure



Figure



This is what you should get:



You can see how all the pixels in the image have been back-projected to 3D and we have a fancy pointcloud showing our scene in 3D. Now all those points are in the `WORLD` reference system and we can move our camera in order to get an image from a different perspective. Let's go for it.

Moving the camera and observing the scene

So, in this last part of this practical exercise we are going to move the camera position and project the pointcloud again to form a new 2D image of it. This implies:

- there is a rotation and translation between the `WORLD` and `CAMERA` reference systems, so we need to compute the point coordinates in the latter,
- then, we need to project those points to the sensor plane,
- then, we need to scale and translate such points in the sensor plane to an *computer image plane*,
- and finally, we need to adjust their coordinates to get the image plane with origin at the top-left.

Therefore, you should use the full projective transformation from the world reference frame to the image:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ \end{bmatrix} \qquad \lambda \tilde{\mathbf{m}}' = \mathbf{K} \begin{bmatrix} \mathbf{R} \, \mathbf{t} \end{bmatrix} \tilde{\mathbf{M}}_{\mathbf{W}}$$

ASSIGNMENT 3c: Taking a picture

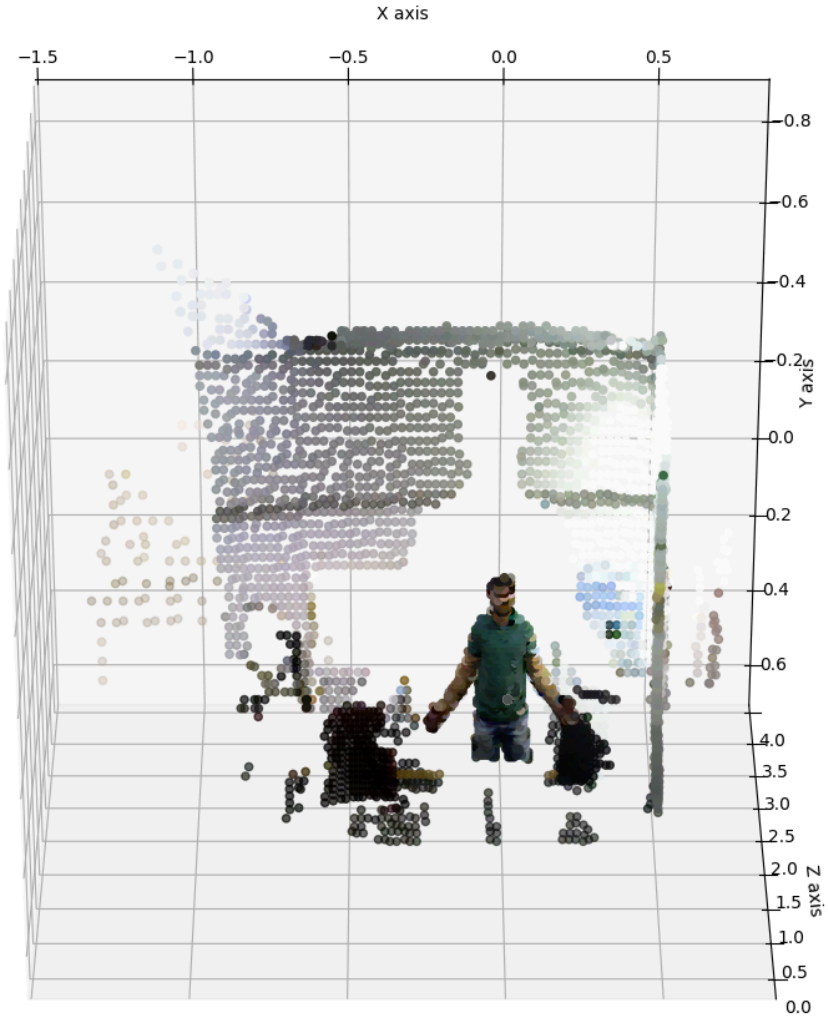
Let's change the pose `CAMERA` in the `WORLD` reference system and take a new picture of the 3D scene. For that:

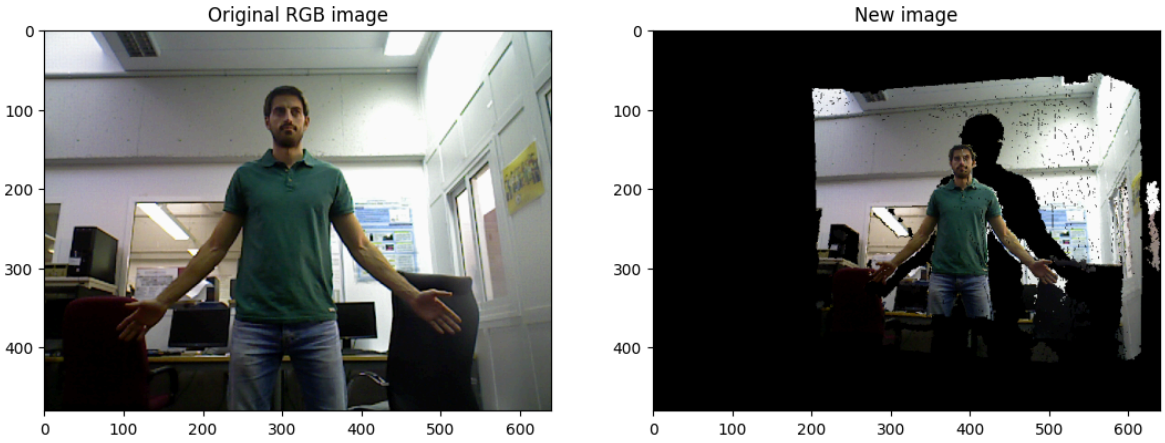
- Define a rotation of 15^{o} in the Y axis and a translation of $0.5m$ in the Z axis to move the camera.
- Then, apply the complete camera model to the previous computed 3D pointcloud `h_map3D`.
- Iterate over the projected points for checking if they must appear in the image:
  - their u and v coordinates are within the image dimensions, and
  - since multiple points can be projected into the same pixel, keep the point that is closer to the camera.
- Finally, show the original image and the new one!

In [7]:
```
%matplotlib inline

# intrinsic parameters
f  = 525
u0 = 319.5
v0 = 239.5

# Define transformation matrices
angle = np.radians(15)
R = np.array([[cos(angle),0,sin(angle)],[0,1,0],[-sin(angle),0,cos(angle)]]) # r

T = np.zeros((3,4))
T[0:3,0:3] = R
T[0:3,3] = np.array([0,0,0.5]) # translation

K = np.array([[f,0,u0],[0,f,v0],[0,0,1]])

# Transform map to homogenous coordinates
h_map3D = np.append(map_3d, np.ones((1,map_3d.shape[1])), axis=0)

# Apply transformation
h_new_image = K @ T @ h_map3D

# Transform to cartesian (they may appear zeroes in z!)
proj = np.divide(h_new_image[:2,:], h_new_image[2,:], where=h_new_image[2,:]!=0)
proj[np.isnan(proj)] = 0 # Fix division by 0
proj = proj.astype(int)
```

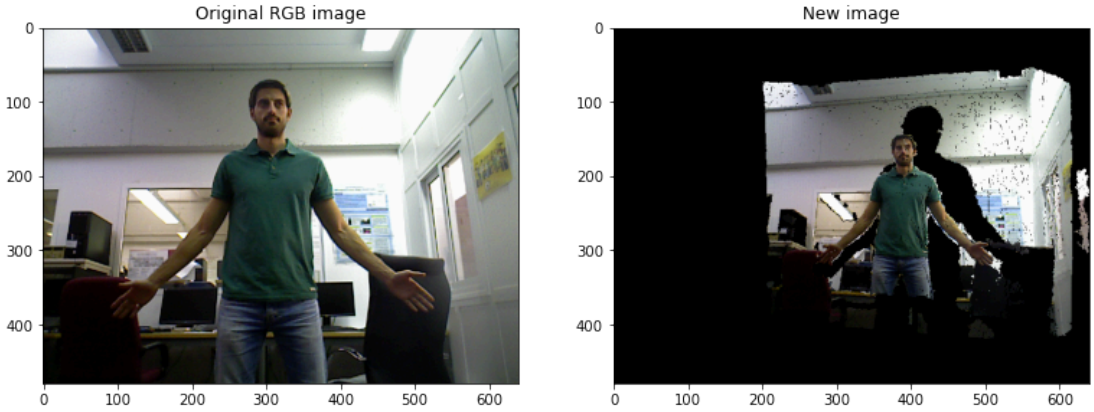
In [8]:
```
# Construct new image
new_image = np.zeros_like(image)
new_depth = np.full(depth.shape,np.inf)

image_vector = image.reshape(image.shape[0]*image.shape[1],3) # this is a Nx3 ve
```

```
# iterate over the projected points and check if they must appear in the image
for p in range(proj.shape[1]):
        u,v = proj[:,p]
        z = h_map3D[2,p] # z-coordinate
        # Should this pixel appear in the image?
        if (u>=0) and (u<new_image.shape[1]) and (v>=0) and (v<new_image.shape[0
            if (new_depth[v,u] > z): # Check if pixel is closer than other in th
                new_depth[v,u] = z
                new_image[v,u,:] = image_vector[p,:] # Get the color

# Show original image
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(13, 13))
ax1.imshow(image)
ax1.set_title("Original RGB image")
ax2.imshow(new_image)
ax2.set_title("New image");
```





This is the resulting image you should have obtained:



You can see how now we have a new perspective of the point cloud! Cool, right?

Thinking about it (1)

Now you are in a good position to answer these questions:

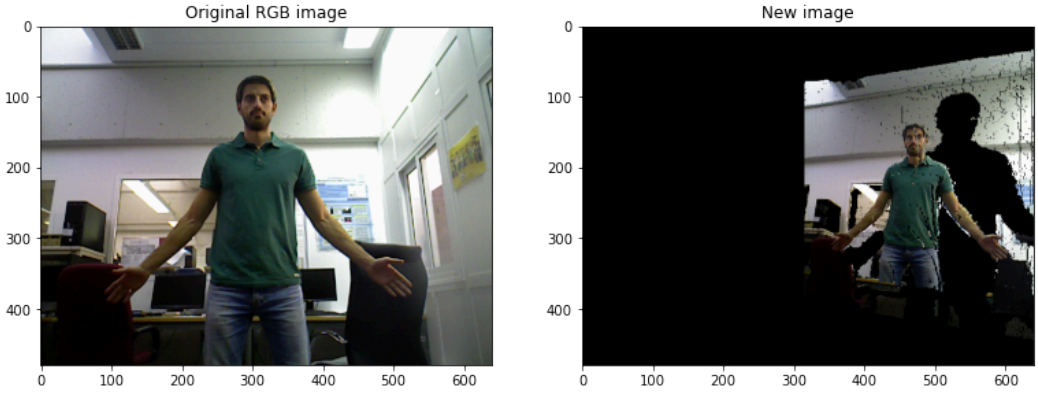
- If we have a RGB-D camera with an operating range from 0 up to 10 meters, what would be the scale used to codify distances in a greyscale image?

  The scale used to codify distances in a greyscale image for a RGB-D camera with that range should be 10/256 = 0,039m.

- If the scale is of 0,01, which would be the maximum operating range? Assume that the operating range starts at 0.

  *If the scale is 0,01, then the maximum operating range would be 0,01*256 = 2,56m.*

- The following image have been taken with the camera rotated in the Y axis. Which has been the rotated angle? $30º$ or $-30ª$?



The shown image has been rotated with an angle of 30º.

Conclusion

Brilliant! Although this has been a dense chapter, it has been satisfying once you get it. In this notebook, you have learned:

- the principle of RGB-D images (widely used in computer vision and robotics fields nowadays),
- how the pinhole camera model works,
- how a more complete camera model works,
- how to project a 3D set of points to a 2D image,
- how to back-project an image with depth information to the 3D space to get a pointcloud.