Rotating objects in three dimensions

Visualized in *Mathematica* using Platonic Solids

Brian Horn, trycatchhorn@gmail.com

Introduction

The purpose of this notebook is to show how rotation of simple solid geometries can be applied. The focus is not to show the theory behind rotation, but merely to show how one can apply rotation of general objects when implementing e.g. simple rendering algorithms.

There exists a lot of great material describing the mathematics behind rotations, but when one needs to implement basic algorithms that use rotations in an applied context it can be hard to transfer the theory into computer code. My hope is that showing howto apply the mathematics involving rotations in *Mathematica* can help when implementing rotations in 3D libraries like OpenGL.

Mathematica supplies a great arsenal of internal functions for working with rotations of geometries in both 2D and 3D. Since the primary focus in this notebook is not about how to use *Mathematica* when solving problems related to rotations, I have deliberately decided to use very general functions and datatypes available in most programming languages.

Initially, the math and the terminolgy used throughout the notebook is presented. This content is not presented with a particular graphics framework in mind, but should be considered adaptable to most graphics libraries. Finally, the formalism is applied in the context of implementing rotation of the Platonic Solids.

Notation and Terminology

When reading about rotation in mathematical literature one often finds quite a lot of formalism with respect to terminology and notation. In the following the terminology used in this notebook is explained.

Coordinate System

When working with rotations it is very important to understand the orientation of the axes in the coordinate system used. *Mathematica* uses a right-handed coordinate system which is the coordinate system that will be used in this notebook. However, there are no convention in graphics about which coordinate system to use, so please consult the documentation for the specific library or API for information about the coordinate system used, as this has a number of implications for how geometric operations are defined.

Coordinate System

When working with rotations the type of coordinate system used is important. In this notebook a right-handed Cartesian coordinate system is used with the origin O located at the intersection of the axes X, Y, and Z.

Given an orthonormal set of basis vectors representing the coordinate axes, there are multiple ways to orient the axes. If you take your right hand and point it along the positive x-axis with your palm facing the positive y-axis and extend your thumb, your thumb indicates the positive direction of the z-axis.

■ Right-handed coordinate system

Figure 1 shows a right-handed Cartesian coordinate system and the order of rotation about each axes. Notice that positive rotations are performed counterclockwise. Here ϕ denotes rotation about the x-axis, θ denotes rotation about the y-axis, and ψ denotes rotation about the z-axis. The symbols denoting each rotation often depend on the convention -and the topic being adressed. More details about the different conventions -and symbols being used when working with 3D rotations, will be given later in the notebook.

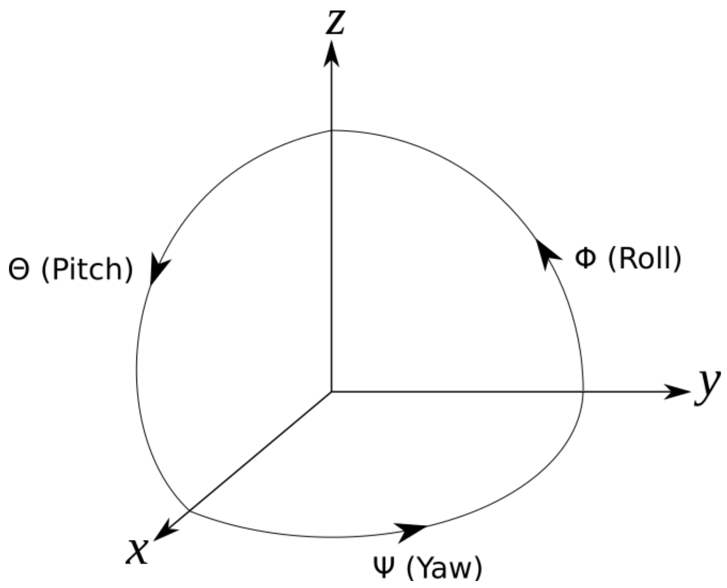
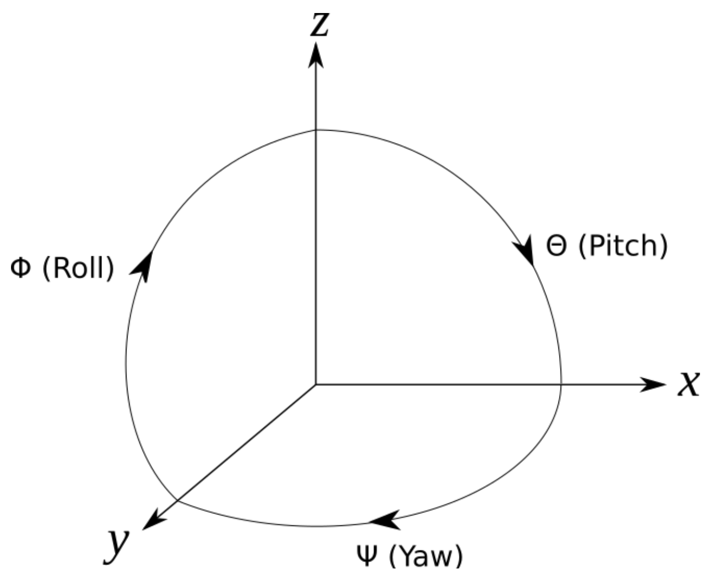


Figure 1

■ Left-handed coordinate system

Figure 2 shows a left-handed Cartesian coordinate system and the order of rotation about each axes. Notice that positive rotations are performed clockwise in this coordinate system. Again, ϕ denotes rotation about the x-axis, θ denotes rotation about the y-axis, and ψ denotes rotation about the z-axis.



Euler Angles

Generally, Euler angles are used to describe the orientation of a given frame or coordinate system with respect to another frame. However, care must be taken when using the term *Euler* angles, since the literature categorizes Euler angles into *types* in order to distinguish whether the specific rotation is performed around a fixed or moving frame.

Euler angles are a set of three angles used to specify the orientation or change in orientation of an object in three dimensional space. Each of the three angles in a Euler angle triplet specifies an elemental rotation around one of the axes in a three-dimensional Cartesian coordinate system as defined above. Unfortunately this is not a complete definition. To completely define a Euler angle system, one must choose from a number of conventions.

The following paragraphs will briefly explain the various terminologies and conventions used when speaking of Euler angles.

■ Elemental Rotation Matrix

Imagine rotating a point P(x, y z) in the sequence:

1) Rotate $P(x, y, z)$ an angle ϕ counterclockwise around the x-axis to a new point $P^{'}(x^{'}, y^{'}, z^{'})$.

2) Rotate $P^{'}(x^{'}, y^{'}, z^{'})$ an angle θ counterclockwise around the y-axis to a new point $P^{''}(x^{''}, y^{''}, z^{''})$.

3) Rotate $P^{''}(x^{''}, y^{''}, z^{''})$ an angle ψ counterclockwise around the z-axis to a new point $P^{'''}(x^{'''}, y^{'''}, z^{'''})$.

It is possible to perform these rotations individually so that when the first rotation is performed the resulting position is used when performing the second rotation and so on. Although, doing this can be quite informative, it is hard to describe a series of rotations into a single rotation - expressed in one equation. In order to achieve this we need to use rotation matrices.

Rotation about the x-axis

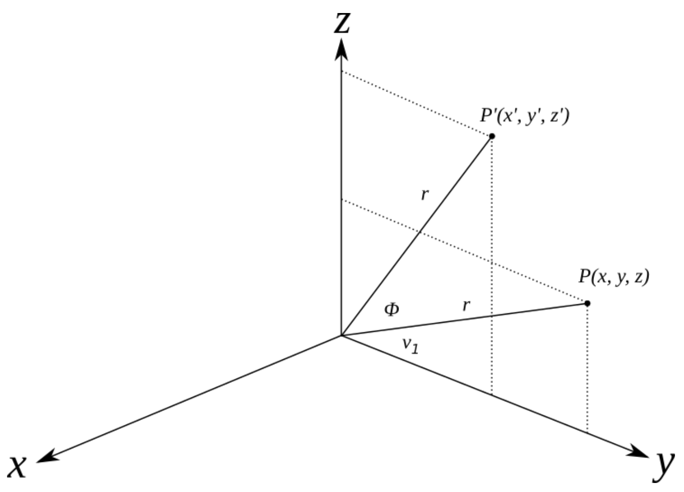


Figure 3 shows how the point $P(x, y, z)$ is rotated about the x-axis into a new position $P^{'}(x^{'}, y^{'}, z^{'})$ preserving the distance from the origo to the point. When rotating the point $P(x, y, z)$ an angle ϕ counterclockwise about the x-axis, the x-coordinate remains constant while the y- and z-coordinates are changed. Algebraically, this is expressed in the equations:

$$x^{'} = x \tag{1}$$

$$y^{'} = r\cos(v_1 + \phi) = r\cos(v_1)\cos(\phi) - r\sin(v_1)\sin(\phi) = r(\cos(v_1)\cos(\phi) - \sin(v_1)\sin(\phi)) \tag{2}$$

$$z^{'} = r\sin(v_1 + \phi) = r\sin(v_1)\cos(\phi) + r\cos(v_1)\sin(\phi) = r(\sin(v_1)\cos(\phi) + \cos(v_1)\sin(\phi)) \tag{3}$$

Where v_1 is the angle between the y-axis and the point $P(x, y, z)$ in the yz-plane. The following relations holds:

$$y = r\cos(v_1) \Longleftrightarrow \cos(v_1) = \frac{y}{r} \tag{4}$$

$$z = r\sin(v_1) \Longleftrightarrow \sin(v_1) = \frac{z}{r} \tag{5}$$

Inserting expressions (4) and (5) into equations (1), (2), and (3) we get

$$x^{'} = x$$

$$y^{'} = r\left(\frac{y}{r}\cos(\phi) - \frac{z}{r}\sin(\phi)\right) = y\cos(\phi) - z\sin(\phi)$$

$$z^{'} = r\left(\frac{z}{r}\cos(\phi) + \frac{y}{r}\sin(\phi)\right) = z\cos(\phi) + y\sin(\phi)$$

Arranging the terms gives

$$x^{'} = x \tag{6}$$

$$y^{'} = y\cos(\phi) - z\sin(\phi) \tag{7}$$

$$z^{'} = y\sin(\phi) + z\cos(\phi) \tag{8}$$

In matrix form

$$\begin{pmatrix} x^{'} \\ y^{'} \\ z^{'} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{9}$$

Therefore, we can write the position of the new point as

$$P' = R_x(\phi)\,P \tag{10}$$

Where $R_x(\phi)$ is the elemental rotation matrix, rotating the point P an angle ϕ counterclockwise around the x-axis. This operation is often called *Roll*.

$$R_x(\phi) = \text{Roll}(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \tag{11}$$

Rotation about the y-axis

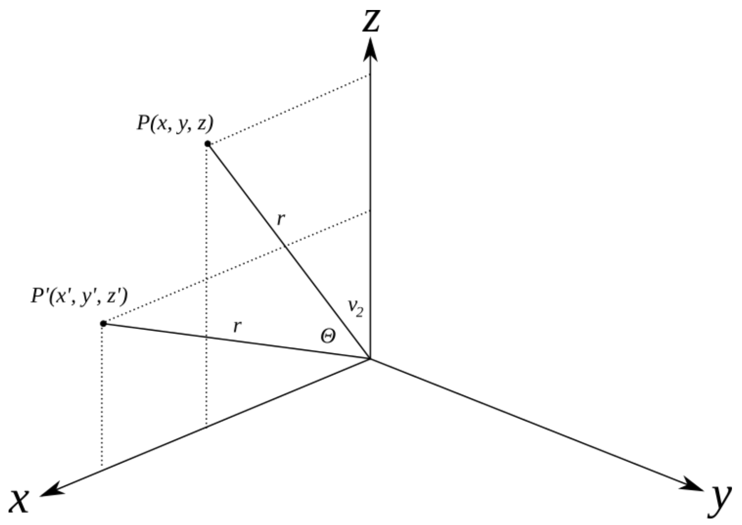


Figure 4

We repeat the exercise of finding the elemental rotation matrix for rotating about the y-axis. Figure 4 shows how the point $P(x,\ y,\ z)$ is rotated about the y-axis into a new position $P'(x',\ y',\ z')$ preserving the distance from the origo to the point. When rotating the point $P(x,\ y,\ z)$ an angle θ counterclockwise about the y-axis, the y-coordinate remains constant while the x- and z-coordinates are changed. This can be exptessed by the equations:

$$x' = r\cos(v) \tag{12}$$

$$y' = y \tag{13}$$

$$z' = r\sin(v) \tag{14}$$

Where the following equation gives the relation between the angles

$$180 = \theta + v_2 + 90 + v \Longleftrightarrow v = 90 - \theta - v_2 \tag{15}$$

Using the trigonometry identities for the complementary angle, this can be written as

$$x' = r\cos(v) = r\cos(90 - \theta - v_2) = r\sin(v_2 + \theta) \tag{16}$$

$$y' = y \tag{17}$$

$$z' = r\sin(v) = r\sin(90 - \theta - v_2) = r\cos(v_2 + \theta) \tag{18}$$

Using the angle sum identities, we get

$$x' = r\sin(v_2+\theta) = r\sin(v_2)\cos(\theta) + r\cos(v_2)\sin(\theta) = r(\sin(v_2)\cos(\theta)+\cos(v_2)\sin(\theta)) \tag{19}$$

$$y' = y \tag{20}$$

$$z' = r\cos(v_2+\theta) = r\cos(v_2)\cos(\theta) - r\sin(v_2)\sin(\theta) = r(\cos(v_2)\cos(\theta)-\sin(v_2)\sin(\theta)) \tag{21}$$

Where v_2 is the angle between the z-axis and the point $P(x, y, z)$ in the xz-plane. The following relations holds:

$$x = r\sin(v_2) \Longleftrightarrow \sin(v_2) = \frac{x}{r} \tag{22}$$

$$z = r\cos(v_2) \Longleftrightarrow \cos(v_2) = \frac{z}{r} \tag{23}$$

Inserting expressions (22) and (23) into equations (19), (20), and (21) we get

$$x' = r\left(\frac{x}{r}\cos(\theta) + \frac{z}{r}\sin(\theta)\right) = x\cos(\theta) + z\sin(\theta)$$

$$y' = y$$

$$z' = r\left(\frac{z}{r}\cos(\theta) - \frac{x}{r}\sin(\theta)\right) = z\cos(\theta) - x\sin(\theta)$$

Arranging the terms gives

$$x' = x\cos(\theta) + z\sin(\theta) \tag{24}$$

$$y' = y \tag{25}$$

$$z' = -x\sin(\theta) + z\cos(\theta) \tag{26}$$

In matrix form

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{27}$$

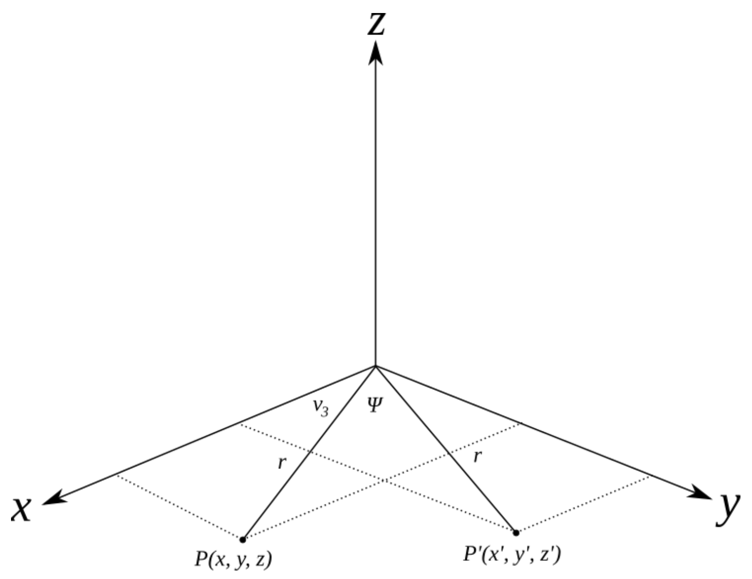
Therefore, we can write the position of the new point as

$$P' = R_y(\theta)\,P \tag{28}$$

Where $R_y(\theta)$ is the elemental rotation matrix, rotating the point P an angle θ counterclockwise around the y-axis. This operation is often called Pitch.

$$R_y(\theta) = \text{Pitch}(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{29}$$

Rotation about the z-axis



Finally, we can derive the elemental rotation matrix for rotating about the z-axis. Figure 5 shows how the point $P(x, y, z)$ is rotated about the z-axis into a new position $P^{'}(x^{'}, y^{'}, z^{'})$ preserving the distance from the origo to the point. When rotating the point $P(x, y, z)$ an angle ψ counterclockwise about the z-axis, the z-coordinate remains constant while the x- and y-coordinates are changed. This can be exptessed by the equations:

$$x^{'} = r\sin(v) \tag{30}$$

$$y^{'} = r\cos(v) \tag{31}$$

$$z^{'} = z \tag{32}$$

Where the following equation gives the relation between the angles

$$180 = \psi + v_3 + 90 + v \Longleftrightarrow v = 90 - \psi - v_3 \tag{33}$$

Using the trigonometry identities for the complementary angle, this can be written as

$$x^{'} = r\sin(v) = r\sin(90 - \psi - v_3) = r\cos(v_3 + \psi) \tag{34}$$

$$y^{'} = r\cos(v) = r\cos(90 - \psi - v_3) = r\sin(v_3 + \psi) \tag{35}$$

$$z^{'} = z \tag{36}$$

Using the angle sum identities, we get

$$x^{'} = r\cos(v_3 + \psi) = r\cos(v_3)\cos(\psi) - r\sin(v_3)\sin(\psi) = r\,(\cos(v_3)\cos(\psi) - \sin(v_3)\sin(\psi)) \tag{37}$$

$$y^{'} = r\sin(v_3 + \psi) = r\sin(v_3)\cos(\psi) + r\cos(v_3)\sin(\psi) = r\,(\sin(v_3)\cos(\psi) + \cos(v_3)\sin(\psi)) \tag{38}$$

$$z^{'} = z \tag{39}$$

Where v_3 is the angle between the x-axis and the point $P(x, y, z)$ in the xy-plane. The following relations holds:

$$x = r\cos(v_3) \Longleftrightarrow \cos(v_3) = \frac{x}{r} \tag{40}$$

$$y = r\sin(v_3) \Longleftrightarrow \sin(v_3) = \frac{y}{r} \tag{41}$$

Inserting expressions (40) and (41) into equations (37), (38), and (39) we get

$x' = r\left(\frac{x}{r}\cos(\psi) - \frac{y}{r}\sin(\psi)\right) = x\cos(\psi) - y\sin(\psi)$

$y' = r\left(\frac{y}{r}\cos(\psi) + \frac{x}{r}\sin(\psi)\right) = y\cos(\psi) + x\sin(\psi)$

$z' = z$

Arranging the terms gives

$$x' = x\cos(\psi) - y\sin(\psi) \tag{42}$$

$$y' = x\sin(\psi) + y\cos(\psi) \tag{43}$$

$$z' = z \tag{44}$$

In matrix form

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{45}$$

Therefore, we can write the position of the new point as

$$P' = R_z(\psi)\, P \tag{46}$$

Where $R_z(\psi)$ is the elemental rotation matrix, rotating the point P an angle ψ counterclockwise around the z-axis. This operation is often called Yaw.

$$R_z(\psi) = \text{Yaw}(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{47}$$

Combined rotation about xyz (roll-pitch-yaw)

Often we would like to perform a sequence of rotations around different axes. Instead of performing intermediate calculations, where we store the result of each elemental rotation and use it to perform the next rotation, we can use matrix multiplication and derive a transformation matrix which expresses a rotation about an arbitrary sequence of coordinate axes.

Suppose an object is rotated an angle ϕ about the x-axis, followed by a rotation an angle θ about the y-axis, ending with a rotation an angle ψ about the z-axis. The resulting rotation matrix can be determined by multiplication of the elemental rotation matrices in the order shown in equation (48).

$$R_{\text{xyz}}(\phi,\ \theta,\ \psi) = R_x(\phi)\, R_y(\theta)\, R_z(\psi) \tag{48}$$

$$R_{\text{xyz}}(\phi,\ \theta,\ \psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{49}$$

The full rotation matrix for the elemental rotation order xyz (roll-pitch-yaw) is given by equation (50).

$$R_{\text{xyz}}(\phi,\ \theta,\ \psi) = \begin{pmatrix} \cos(\theta)\cos(\psi) & -\cos(\phi)\sin(\psi) & \sin(\theta) \\ \cos(\psi)\sin(\theta)\sin(\phi) + \cos(\phi)\sin(\psi) & \cos(\phi)\cos(\psi) - \sin(\theta)\sin(\phi)\sin(\psi) & -\cos(\theta)\sin(\phi) \\ -\cos(\phi)\cos(\psi)\sin(\theta) + \sin(\phi)\sin(\psi) & \cos(\psi)\sin(\phi) + \cos(\phi)\sin(\theta)\sin(\psi) & \cos(\theta)\cos(\phi) \end{pmatrix} \tag{50}$$

Combined rotation about zyx (yaw-pitch-roll)

Suppose an object is rotated an angle ψ about the z-axis, followed by a rotation an angle θ about the y-axis, ending with a rotation an angle ϕ about the x-axis. The resulting rotation matrix can be determined by multiplication of the elemental rotation matrices in the order shown in equation (51).

$$R_{\text{zyx}}(\psi,\ \theta,\ \phi) = R_z(\psi)\, R_y(\theta)\, R_x(\phi) \tag{51}$$

$$R_{\text{zyx}}(\psi,\ \theta,\ \phi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \tag{52}$$

The full rotation matrix for the elemental rotation order zyx (yaw-pitch-roll) is given by equation (53).

$$R_{\text{zyx}}(\psi,\ \theta,\ \phi) = \begin{pmatrix} \cos(\theta)\cos(\psi) & \cos(\psi)\sin(\theta)\sin(\phi) - \cos(\phi)\sin(\psi) & \cos(\phi)\cos(\psi)\sin(\theta) + \sin(\phi)\sin(\psi) \\ \cos(\theta)\sin(\psi) & \cos(\phi)\cos(\psi) + \sin(\theta)\sin(\phi)\sin(\psi) & -\cos(\psi)\sin(\phi) + \cos(\phi)\sin(\theta)\sin(\psi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{pmatrix} \tag{53}$$

■ Proper Euler Angles, Classic Euler Angles, or Fixed Euler Angles

Proper Euler angles, Classic Euler angles, or Fixed Euler angles refer to the same thing. In this notebook we will use the term Proper Euler angles. In the Proper Euler angles convention, the three elemental rotations are performed around only two axes. For example, the first rotation may be around the z axis, the second around the y axis, and the third around the z axis again - resulting in a z-y-z rotation. The Proper Euler angle convention gives a total number of six possible rotations about two axes in a Cartesian coordinate system. In all of them, the first and third rotation axes are the same. Each of these are given below:

* X-Z-X

* X-Y-X

* Y-X-Y

* Y-Z-Y

* Z-Y-Z

* Z-X-Z

■ Tait-Bryan Angles

Tait-Bryan angles are a special type of Euler angles, involving all three axes in the Cartesian coordinate system. In the Tait-Bryan convention, each of the three angles in a Euler angle triplet defines the rotation around a different Cartesian axis. For example, the first angle may specify the rotation around the z axis, the second around the y axis, and the third around the x axis - resulting in a z-y-x rotation. The Tait-Bryan angle convention gives a total number of six possible rotations about three axes in a Cartesian coordinate system. In all of them, each rotation occurs about a new axis. Each of these are given below:

* X-Z-Y

* X-Y-Z

* Y-X-Z

* Y-Z-X

* Z-Y-X

* Z-X-Y

■ Intrinsic Rotations

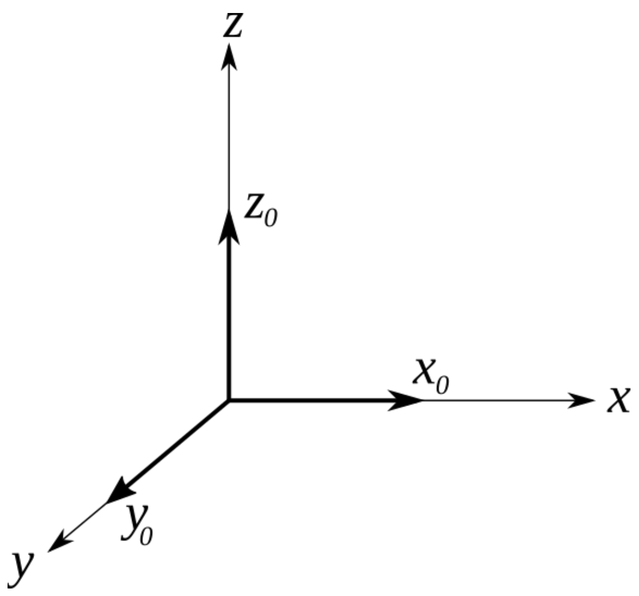
Generally, rotations can be divided into two categories: *intrinsic* rotations and *extrinsic* rotations. Both systems is perfectably able to express any rotation and it is in fact possible to convert a rotation expressed in the intrinsic system into a rotation expressed in the extrinsic system and vice versa. The three elemental rotations may occur either about the axes of the original coordinate system, which remains motionless (extrinsic rotations), or about the axes of the rotating coordinate system, which changes its orientation after each elemental rotation (intrinsic rotations). As such there is no particular reason to favar one system from the other - often it depends on the problem at hand. Definitions of both systems follow below.

Intrinsic rotations are elemental rotations that occur about the axes of a Cartesian XYZ coordinate system attached to a moving body. Therefore, they change their orientation after each elemental rotation. The XYZ body coordinate system rotates, while the XYZ world coordinate system is fixed. Starting with XYZ body coordinate system overlapping the XYZ world coordinate system, a composition of three intrinsic rotations can be used to reach any target orientation for XYZ.

Both Proper Euler angles and Tait-Bryan can be defined by intrinsic rotations. The rotated frame (Body) XYZ may be imagined to be initially aligned with the fixed frame (World) XYZ, before undergoing the three elemental rotations represented by either Proper Euler angles or Tait-Bryan angles. This is shown in figure 6.

For Tait-Bryan angles the successive intrinsic orientations may be denoted as follows:

- X_0-Y_0-Z_0 (initial)
- X_1-Y_1-Z_1 (after first rotation)
- X_2-Y_2-Z_2 (after second rotation)
- X_3-Y_3-Z_3 (after third rotation)



Figures 7, 8, and 9 show successive Tait-Bryan intrinsic rotations applied in the order ZYX. Initially, the world coordinate frame is aligned with the body coordinate frame. The fixed world coordinate axes are denoted XYZ and the axes of the moving body frame are denoted $X_0 Y_0 Z_0$ as shown in figure 6. The first rotation is performed about the Z-axis, where the body frame is rotated an angle ψ counterclockwise. This turns the X -and Y axes of the body frame into a new orientation - these axes are now denoted X_1 and Y_1, as shown in the first frame in figure 7. The second rotation, rotates the body frame an angle θ counterclockwise about the Y_1-axis, changing the orientation of the (old) X_1 -and Z_0 axes, which now are denoted X_2 -and Z_1, as shown in the second frame in figure 8. The final rotation, rotates the body frame an angle ϕ counter-clockwise about the X_2 axis, turning the (old) Y_1 -and Z_1 axes into Y_2 -and Z_2, as show in the last frame in figure 9. It is important to note that the intrinsic Tait-Bryan rotation sequence shown in figures 7, 8, and 9, uses the Tait-Bryan ZYX rotation order.

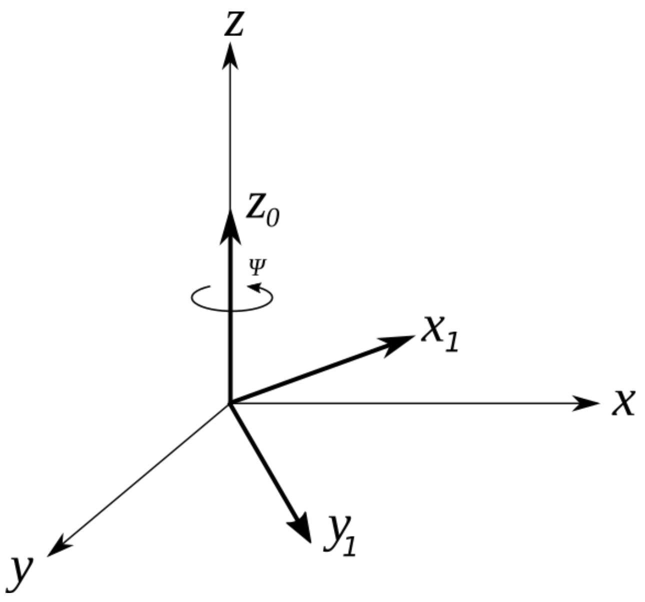


Figure 7

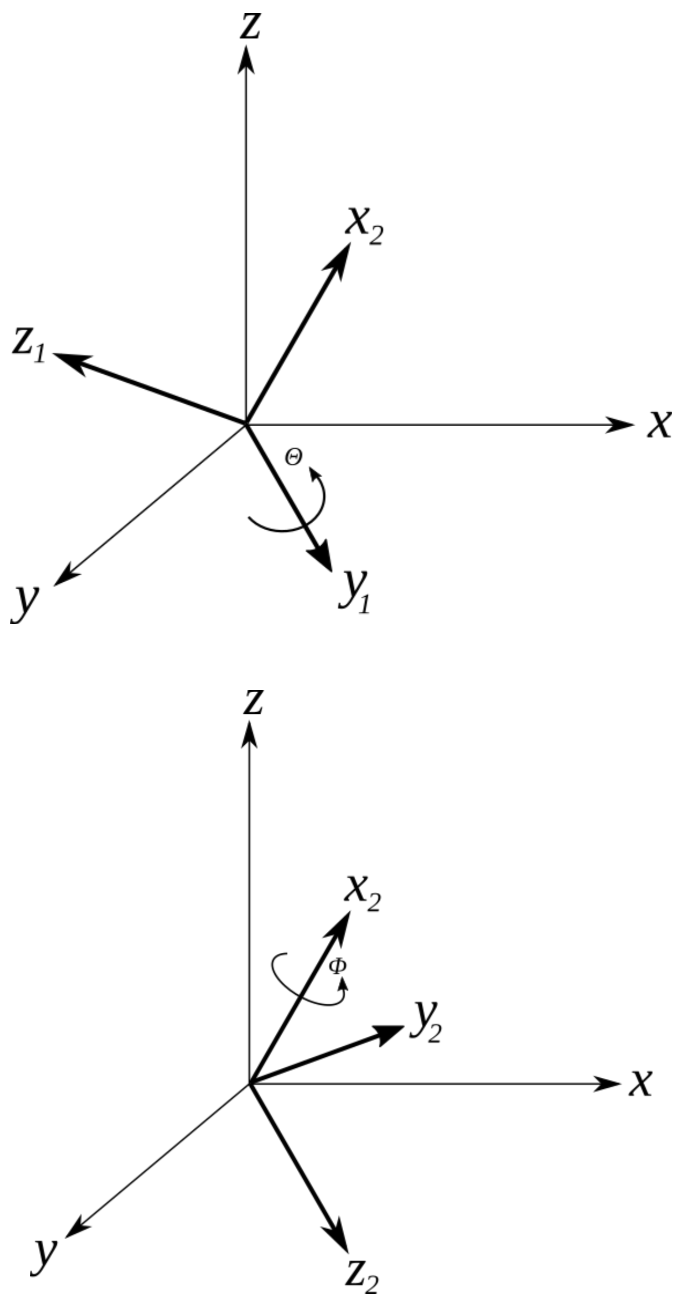


Figure 9

■ Extrinsic Rotations

Extrinsic rotations are elemental rotations that occur about the axes of the fixed world coordinate system, represented by a Cartesian XYZ coordinate system. The XYZ body coordinate system rotates, while the XYZ body coordinate system is fixed. Simlar to the intrinsic system, the world coordinate system and the body coordinate system are initially overlapping, as shown in figure 10. A composition of three extrinsic rotations can be used to reach any target orientation for XYZ. The Proper Euler or Tait-Bryan angles are the amplitudes of these elemental rotations.

For Tait-Bryan angles the successive extrinsic ZYX orientations may be denoted as follows:

- Body -and world coordinate system align (initial).
- First rotation is performed about the fixed Z-axis of the world frame.
- Second rotation is performed about the fixed Y-axis of the world frame.
- Third rotation is performed about the fixed X-axis of the world frame.

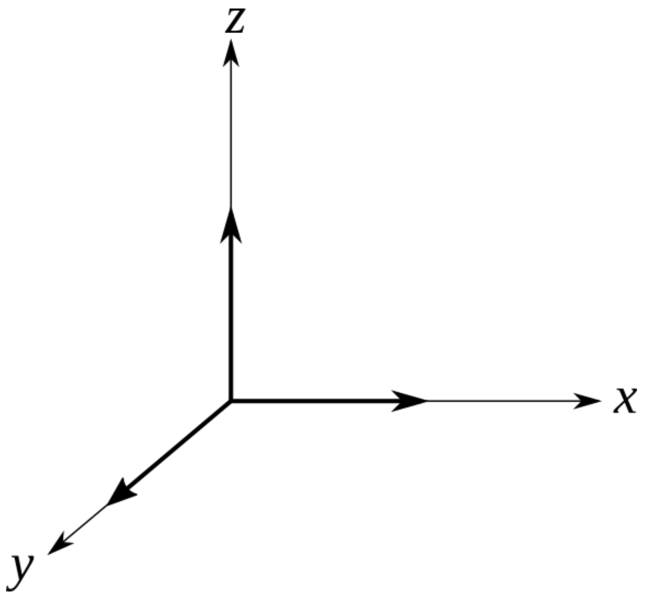


Figure 10

Figures 11, 12, and 13 show successive Tait-Bryan extrinsic rotations applied in the order XYZ. NOTE: here we have reversed the order of rotation, compared to the intrinsic rotations performed above. Initially, the world coordinate frame is aligned with the body coordinate frame. In both coordinate systems, the world coordinate system and the body coordinate system, the axes are denoted XYZ, since all successive rotations are performed around the fixed world frame. The first rotation is performed about the X-axis in the world coordinate system, where the system is rotated an angle ϕ counterclock-wise, as shown in the first frame in figure 11. The second rotation is performed about the Y-axis in the world coordinate system, where the system is rotated an angle θ counterclockwise, as shown in the second frame in figure 12. Finally, the extrinsic rotation sequence finishes by rotation the about the Z-axis in the world coordinate system, where the system is rotated an angle ψ counterclockwise, as shown in the third -and last frame in figure 13.

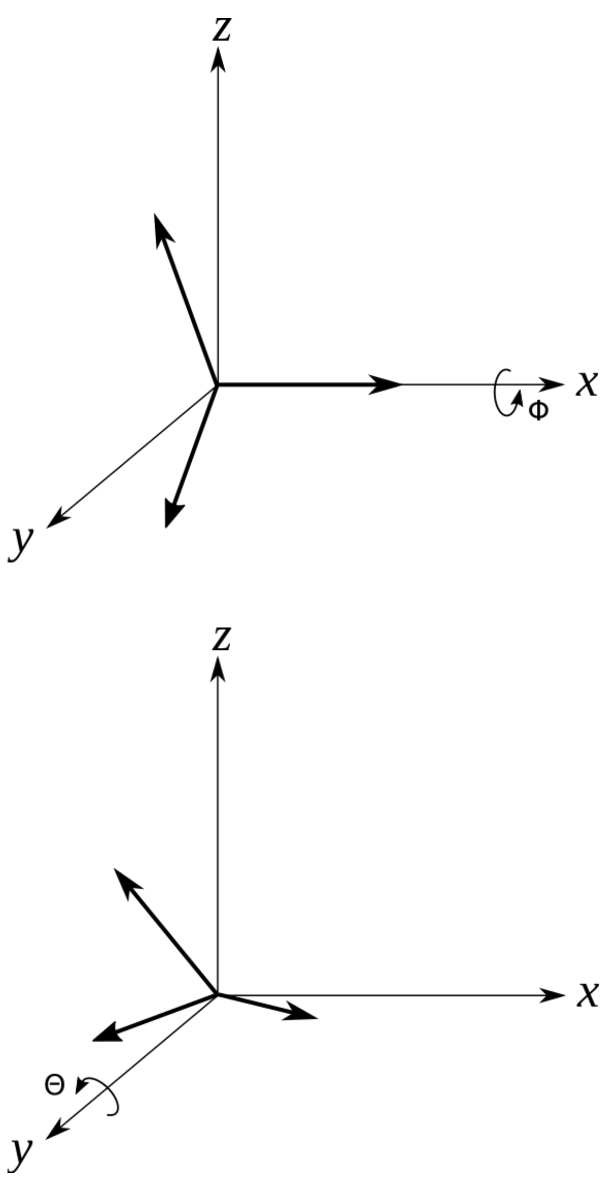
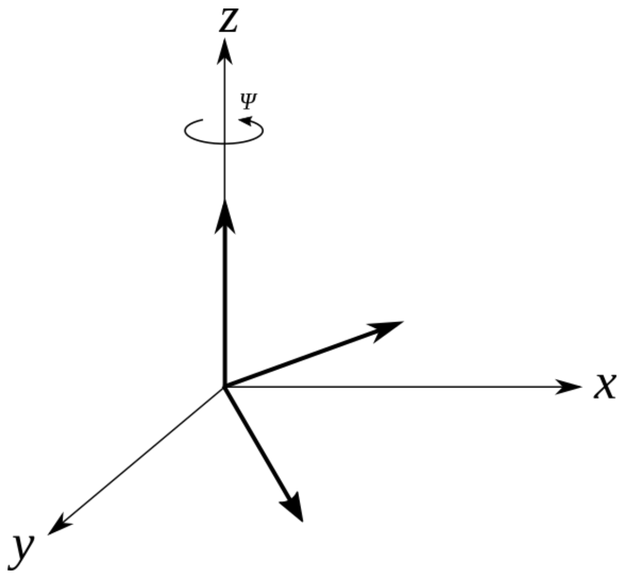


Figure 12



From the above it follows that ány extrinsic rotation is equivalent to an intrinsic rotation by the same angles but with inverted order of elemental rotations, and vice versa. For instance, the intrinsic rotations Z-Y-X by respective angles ψ, θ, ϕ are equivalent to the extrinsic rotations X-Y.Z by angles ϕ, θ, ψ. Both are represented by matrices as shown below:

$$R = R_{\text{zyx}}(\psi,\ \theta,\ \phi) = R_z(\psi)\, R_y(\theta)\, R_x(\phi) \text{ (intrinsic)} \tag{54}$$

$$R = R_{\text{xyz}}(\phi,\ \theta,\ \psi) = R_x(\phi)\, R_y(\theta)\, R_z(\psi) \text{ (extrinsic)} \tag{55}$$

■ Change of Basis

Mathematically, an active change of a vector means an operation that takes a vector and returns a different vector. A passive change leaves the vector alone, but changes the basis that is used to describe that vector.

We seek a releationship between two orthonormal bases with a common origin. The first thing to notice is that the transformation is linear, which can be seen from the definition of a vector basis. For a set of basis vectors (**i**, **j**, **k**), every vector can be expanded as a linear combination:

$$\boldsymbol{v} = v_1\, \boldsymbol{i} + v_2\, \boldsymbol{j} + v_3\, \boldsymbol{k} \tag{56}$$

Since any other basis is comprised of vectors, the vectors of the new basis, denoted ($\boldsymbol{i}'$, $\boldsymbol{j}'$, $\boldsymbol{k}'$), may be expanded in the old basis:

$$\boldsymbol{i}' = a_{11}\, \boldsymbol{i} + a_{12}\, \boldsymbol{j} + a_{13}\, \boldsymbol{k} \tag{57}$$

$$\boldsymbol{j}' = a_{21}\, \boldsymbol{i} + a_{22}\, \boldsymbol{j} + a_{23}\, \boldsymbol{k} \tag{58}$$

$$\boldsymbol{k}' = a_{31}\, \boldsymbol{i} + a_{32}\, \boldsymbol{j} + a_{33}\, \boldsymbol{k} \tag{59}$$

and conversely,

$$\boldsymbol{i} = b_{11}\, \boldsymbol{i}' + b_{12}\, \boldsymbol{j}' + b_{13}\, \boldsymbol{k}' \tag{60}$$

$$\boldsymbol{j} = b_{21}\, \boldsymbol{i}' + b_{22}\, \boldsymbol{j}' + b_{23}\, \boldsymbol{k}' \tag{61}$$

$$\boldsymbol{k} = b_{31}\,\boldsymbol{i}' + b_{32}\,\boldsymbol{j}' + b_{33}\,\boldsymbol{k}' \tag{62}$$

Passive transformation

In a passive transformation any given vector remains fixed while its basis is transformed. Thus, for a passive transformation and an arbitrary vector **v**, we expand in each basis:

$$\begin{aligned}\boldsymbol{v} &= v_1\,\boldsymbol{i} + v_2\,\boldsymbol{j} + v_3\,\boldsymbol{k}\\ &= v_1{}'\,\boldsymbol{i}' + v_2{}'\,\boldsymbol{j}' + v_3{}'\,\boldsymbol{k}'\end{aligned} \tag{63}$$

where the basis vectors are related as above.

Active transformation

In a active transformation the basis remains fixed while all vectors are transformed. Thus, for a active transformation and two arbitrary vectors **v** and $\boldsymbol{v}'$

$$\boldsymbol{v} = v_1\,\boldsymbol{i} + v_2\,\boldsymbol{j} + v_3\,\boldsymbol{k} \tag{64}$$

$$\boldsymbol{v}' = v_1{}'\,\boldsymbol{i} + v_2{}'\,\boldsymbol{j} + v_3{}'\,\boldsymbol{k} \tag{65}$$

where the components are releated by

$$\boldsymbol{v} = v_1\,\boldsymbol{i} + v_2\,\boldsymbol{j} + v_3\,\boldsymbol{k} \tag{66}$$

$$\boldsymbol{v}' = v_1\,\boldsymbol{i}' + v_2\,\boldsymbol{j}' + v_3\,\boldsymbol{k}' \tag{67}$$

■ Einstein Summation Convention

Having looked at changing basis of vectors it is easy to see that this involves a lot of vector algebra. To simplify the calculations a special convention - called *Einstein summation convention* or *repeated index notation* is used. Using index notation the three basis vectors are denoted by $\hat{\boldsymbol{e}}_i$, i = 1, 2, 3, so that

$$\hat{\boldsymbol{e}}_1 = \hat{\boldsymbol{i}} \tag{68}$$

$$\hat{\boldsymbol{e}}_2 = \hat{\boldsymbol{j}} \tag{69}$$

$$\hat{\boldsymbol{e}}_3 = \hat{\boldsymbol{k}} \tag{70}$$

And similarly for $\hat{\boldsymbol{e}}'_i$

$$\hat{\boldsymbol{e}}'_1 = \hat{\boldsymbol{i}}' \tag{71}$$

$$\hat{\boldsymbol{e}}'_2 = \hat{\boldsymbol{j}}' \tag{72}$$

$$\hat{\boldsymbol{e}}'_3 = \hat{\boldsymbol{k}}' \tag{73}$$

Therefore, the basis transformations for both passive -and active transformations can be written as

$$\hat{\boldsymbol{e}}'_i = \sum_{j=1}^{3} a_{ij}\,\hat{\boldsymbol{e}}_j \tag{74}$$

$$\hat{\boldsymbol{e}}_i = \sum_{j=1}^{3} b_{ij}\,\hat{\boldsymbol{e}}'_j \tag{75}$$

For the vector the expansion can be written as

$$\boldsymbol{v} = \sum_{j=1}^{3} v_j \, \hat{\boldsymbol{e}}_j \tag{76}$$

In the above expressions the index *j* is repeated. Repeated indices are always contained within summations, or phrased differently a repeated index implies a summation. Therefore, the summation symbol is typically dropped. The repeated index notation is know as Einstein’s convention. Any repeated index is called a dummy index. Since a repeated index implies a summation over all possible values of the index, one can always relable a dummy index. Therefore, the above expressions can be written as

$$\sum_{j=1}^{3} a_{ij} \, \hat{\boldsymbol{e}}_j \Longrightarrow a_{ij} \, \hat{\boldsymbol{e}}_j \tag{77}$$

$$\sum_{j=1}^{3} b_{ij} \, \hat{\boldsymbol{e}}'_j \Longrightarrow b_{ij} \, \hat{\boldsymbol{e}}'_j \tag{78}$$

$$\sum_{j=1}^{3} v_j \, \hat{\boldsymbol{e}}_j \Longrightarrow v_j \, \hat{\boldsymbol{e}}_j \tag{79}$$

Since the we can choose any symbol for the repeated index, we can write

$$v_j \, \hat{\boldsymbol{e}}_j = v_k \, \hat{\boldsymbol{e}}_k \tag{80}$$

as long as we do not use an index that we have used elsewhere in the same expression. Thus, in the basis change examples above, we cannot use *i* as dummy index because it is used to distinguish three independent equations:

$$\hat{\boldsymbol{e}}'_1 = a_{1\,j} \, \hat{\boldsymbol{e}}_j \tag{81}$$

$$\hat{\boldsymbol{e}}'_2 = a_{2\,j} \, \hat{\boldsymbol{e}}_j \tag{82}$$

$$\hat{\boldsymbol{e}}'_3 = a_{3\,j} \, \hat{\boldsymbol{e}}_j \tag{83}$$

Such an index is called a free index. Free indices must match in every term of an expression. Since the basis is orthonormal, we know that the dot product is given by

$$\hat{\boldsymbol{e}}_i \bullet \hat{\boldsymbol{e}}_j = \delta_{ij} \tag{84}$$

where δ_{ij} is the Kronecker delta, defines as:

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \quad i, \; j = 1, \, 2, \, 3 \tag{85}$$

The Kronecker delta nicely summarizes the rules for computing dot products of orthogonal unit vectors; if the two vectors have the same subscript, meaning they are in the same direction, their dot product is one. If they have different subscripts, meaning they are in different directions, their dot product is zero.

Wen working with the Kronecker delta function and summations it is important to recall that summation notation is built upon a simple protocol: repeated indices indicate a sum over that index from 1 to 3. Therefore, it is important to remember that expressions like δ_{ij} do not imply any summation since there is no repeated index. Below a couple of examples involving the Kronecker delta are presented.

Example 1

What is the value of δ_{ii}? This expression does have a repeated index, and means the sum should be taken over all values of i from 1 to 3. This means that:

$$\delta_{ii} = \delta_{11} + \delta_{22} + \delta_{33} = 1 + 1 + 1 = 3 \tag{86}$$

This is of course exactly the same result you would get from δ_{jj} or δ_{kk}. The choice of index is irrelevant, what matters is that the index is repeated.

Example 2

What is the value of $\delta_{ij}\delta_{jk}$? We realize that the first delta will go to zero unless $i=j$; we can make that substitution in the second delta and contact the two deltas into one as:

$$\delta_{ij}\,\delta_{jk} = \delta_{ik} \tag{87}$$

The logic is straightforward: the first delta will be zero unless $i = j$, and the second delta will be zero unless $j = k$; this is equivalent to saying that the product is zero unless $i = k$, which is the result reflected on the right side.

Example 3

How would we evaluate the expression $x_i\, x_j\, \delta_{ij}$? We now have two repeated indices i and j, and we sum over both of them. Setting "j" as the "inner" variable and summing over that first (only indexing the "i" counter once "j" runs from 1 to 3):

$$\begin{aligned} &x_i\, x_j\, \delta_{ij} = \\ &x_1\, x_1\, \delta_{11} + x_1\, x_2\, \delta_{12} + x_1\, x_3\, \delta_{13} + x_2\, x_1\, \delta_{21} + x_2\, x_2\, \delta_{22} + x_2\, x_3\, \delta_{23} + x_3\, x_1\, \delta_{31} + x_3\, x_2\, \delta_{32} + x_3\, x_3\, \delta_{33} \\ &= x_1^2 + x_2^2 + x_3^2 \end{aligned} \tag{88}$$

It takes a little practice before being completely comfortable working with Einstein summations. However, it is important to remember the it is always possible to write the summations explicitly:

$$\begin{aligned} &\sum_{i=1}^{3}\sum_{j=1}^{3} x_i\, x_j\, \delta_{ij} = \sum_{i=1}^{3}(x_i\, x_1\, \delta_{i1} + x_i\, x_2\, \delta_{i2} + x_i\, x_3\, \delta_{i3}) \\ &= \\ &x_1\, x_1\, \delta_{11} + x_1\, x_2\, \delta_{12} + x_1\, x_3\, \delta_{13} + x_2\, x_1\, \delta_{21} + x_2\, x_2\, \delta_{22} + x_2\, x_3\, \delta_{23} + x_3\, x_1\, \delta_{31} + x_3\, x_2\, \delta_{32} + x_3\, x_3\, \delta_{33} \\ &= x_1^2 + x_2^2 + x_3^2 \end{aligned} \tag{89}$$

Of course, there is no need (nor is it advisable) to do these summations explicitly. The final result could have been deduced immediately by recognizing that the product of $x_i\, x_j\, \delta_{ij}$ will be zero unless $i = j$, and if $i = j$, the expression simplifies to $x_i\, x_j$.

Relationship between basis matrices A and B

Having look at examples of Einstein's summation convention and the Kroneker delta function we return to the matrices a_{ij} and b_{ij} presented in equations (57-59) and equations (60-62) respectively. The relationship between the matrices a_{ij} and b_{ij}, can be found by substituting one basis change into the other:

$$\begin{aligned} \hat{\boldsymbol{e}}'_i &= a_{ij}\, \hat{\boldsymbol{e}}_j \\ &= a_{ij}\left(b_{jk}\, \hat{\boldsymbol{e}}'_k\right) \\ &= a_{ij}\, b_{jk}\, \hat{\boldsymbol{e}}'_k \end{aligned} \tag{90}$$

Taking the dot product with $\hat{\boldsymbol{e}}'_m$ on each side of the equation (notice that we cannot use *i*, *j*, or *k*), we have

$$\hat{\boldsymbol{e}}'_i = a_{ij}\, b_{jk}\, \hat{\boldsymbol{e}}'_k \tag{91}$$

$$\hat{\boldsymbol{e}}'_m\, \hat{\boldsymbol{e}}'_i = \hat{\boldsymbol{e}}'_m\left(a_{ij}\, b_{jk}\, \hat{\boldsymbol{e}}'_k\right) \tag{92}$$

$$\delta_{mi} = a_{ij}\, b_{jk}\, \hat{\boldsymbol{e}}'_m\, \hat{\boldsymbol{e}}'_k \tag{93}$$

$$= a_{ij}\, b_{jk}\, \delta_{mk} \tag{94}$$

$$= a_{ij}\, b_{jm} \tag{95}$$

Since $\delta_{mi} = \delta_{im}$ is the identify matrix *I*, it follows that

$$I = a_{ij}\, b_{jm} = AB \tag{96}$$

Therefore, the matrix *B*, with components b_{ij}, is the inverse to matrix A:

$$B = A^{-1} \tag{97}$$

■ Passive Rotations

Passive rotation, also known as *alias* rotation, is when a coordinate system rotates with respect to the point. The active and the passive conventions produce opposite rotations.

Consider a passive transformation from $\hat{\boldsymbol{e}}_j$ to $\hat{\boldsymbol{e}}'_i$. Substituting for the relationship between the basis vectors, this can be written

$$\begin{aligned} v'_i\, \hat{\boldsymbol{e}}'_i &= v_i\, \hat{\boldsymbol{e}}_i \\ &= v_i\left(b_{ij}\, \hat{\boldsymbol{e}}'_j\right) \\ &= \left(v_i\, b_{ij}\right) \hat{\boldsymbol{e}}'_j \end{aligned} \tag{98}$$

Continuing with the passive transformation, taking the dot product of both sides of the equation with each of the three basis vectors, $\hat{\boldsymbol{e}}'_k$, we get:

$$v'_i\, \hat{\boldsymbol{e}}'_i\, \hat{\boldsymbol{e}}'_k = \left(v_i\, b_{ij}\right) \hat{\boldsymbol{e}}'_j\, \hat{\boldsymbol{e}}'_k \tag{99}$$

$$v'_i\, \delta_{ik} = v_i\, b_{ij}\, \delta_{jk} \tag{100}$$

$$v'_k = v_i\, b_{ik} \tag{101}$$

■ Active Rotations

Active rotation, also known as *alibi* rotation, is when a point is rotated relative to the reference coordinate system and the reference coordinate system remains fixed.

Consider an active rotation of a vector **v** to a new vector $\boldsymbol{v}'$. To see what is happening, first suppose we have two basis vectors, $\hat{\boldsymbol{e}}_i$ and $\hat{\boldsymbol{e}}'_i$, which differ only by a counterclockwise rotation around the z-axis through and angle ψ. Therefore, we can write:

$$\boldsymbol{i}' = \boldsymbol{i}\cos(\psi) - \boldsymbol{j}\sin(\psi) \tag{102}$$

$$\boldsymbol{j}^{'} = \boldsymbol{i}\sin(\psi) + \boldsymbol{j}\cos(\psi) \tag{103}$$

$$\boldsymbol{k}^{'} = \boldsymbol{k} \tag{104}$$

So that

$$a_{ij} = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{105}$$

Now, suppose we want **v** to be pointing in the x-direction and we want $v^{'}$ to be rotated by an angle ψ. Then we have

$$\boldsymbol{v} = v\boldsymbol{i} \tag{106}$$

$$\boldsymbol{v}^{'} = \boldsymbol{v}\boldsymbol{i}^{'} \tag{107}$$

Therefore,

$$\boldsymbol{v}^{'} = v\,(\boldsymbol{i}\cos(\psi) - \boldsymbol{j}\sin(\psi)) \tag{108}$$

$$\boldsymbol{v}^{'} = \boldsymbol{i}\,v\cos(\psi) - \boldsymbol{j}\,v\sin(\psi) \tag{109}$$

So the components of $\boldsymbol{v}^{'}$ in the unprimed basis are

$$v_i^{'} = [v\cos(\psi),\ -v\sin(\psi),\ 0]^T \tag{110}$$

or, in terms of a_{ij},

$$v_i^{'} = v_j\,a_{ji} \tag{111}$$

This is the general relationship between **v** and $\boldsymbol{v}^{'}$ since in general, if

$$\hat{\boldsymbol{e}}_i^{'} = a_{ij}\,\hat{\boldsymbol{e}}_j \tag{112}$$

and we require

$$\boldsymbol{v} = v_i\,\hat{\boldsymbol{e}}_i \tag{113}$$

$$\boldsymbol{v}^{'} = v_i\,\hat{\boldsymbol{e}}^{'}_i = v_i^{'}\,\hat{\boldsymbol{e}}_i \tag{114}$$

it follows that

$$v_i^{'}\,\hat{\boldsymbol{e}}_i = v_i\,\hat{\boldsymbol{e}}^{'}_i \tag{115}$$

$$v_i^{'}\,\hat{\boldsymbol{e}}_i = v_i\,a_{ij}\,\hat{\boldsymbol{e}}_j \tag{116}$$

$$v_k^{'} = v_i\,a_{ik} \tag{117}$$

Notice that this transformation is exactly the inverse to the transformation of the basis.

■ Rotation Order

When performing elemental rotations around each of the three axes, the order in which the rotations are executed matters. That is, performing elemental rotations around the axes in the order z, then y, then x will produce different results than performing the same rotations in any of the other five possible orders.

Interpolating the orientation of a rigid body is subtle. In fact, even specifying the orientation is not easy. If we specify orientations by amounts of rotation about the three principal axes, then the order of specification is important. For example, if a book with its spine facing left is rotated by 90° about the x-axis and the -90° about the y-axis, its spine will face you, whereas if the rotations are done in the opposite order, its spine will face down.

When working with rotation of objects, there often exist multiple ways to achieve a desired orientation. As an example, consider a book laying on a table face up in front of you. Define the x-axis as to the right, the y-axis as away from you, and the z-axis up. A rotation of π radians about the y-axis will turn the book so that the back cover is now facing up. Another way to achieve the same orientation would be to rotate the book π radians about the x-axis, and then π radians about the z-axis. Thus, there is more than one way to achieve a desired rotation.

■ Direction of Rotations

When working with rotations and rotation matrices it is important to be consistent with the direction of rotation - that is whether the elemental rotations are performed clockwise or counterclockwise about the coordinate axes.

Counterclockwise rotation

As stated previously an elemental rotation is a rotation about one of the axes of a Coordinate system. The following three basic rotation matrices rotate vectors **counterclockwise** by angles ϕ, θ, and ψ about the x, y, and z axis respectively in three dimensions using the right-hand-rule.

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \tag{118}$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{119}$$

$$R_z(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{120}$$

Clockwise rotation

Suppose that the rotations are performed about the same axes as shown for the counterclockwise rotations above, but this time the rotations are performed **clockwise** by angles ϕ, θ, and ψ about the x, y, and z axis respectively using the right-hand-rule.

Inverting the direction of rotation corresponds to inserting $-\phi$, $-\theta$, and $-\psi$ in the elemental rotation matrices in equations (118), (119), and (120) respectively.

$$R_x(-\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(-\phi) & -\sin(-\phi) \\ 0 & \sin(-\phi) & \cos(-\phi) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix} \tag{121}$$

$$R_y(-\theta) = \begin{pmatrix} \cos(-\theta) & 0 & \sin(-\theta) \\ 0 & 1 & 0 \\ -\sin(-\theta) & 0 & \cos(-\theta) \end{pmatrix} = \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{122}$$

$$R_z(-\psi) = \begin{pmatrix} \cos(-\psi) & -\sin(-\psi) & 0 \\ \sin(-\psi) & \cos(-\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{123}$$

Generally, the relationship between clockwise -and counterclockwise rotations in a right-handed coordinate system can be written as:

$$R_x(-\phi) = [R_x\,(\phi)]^{-1} = [R_x(\phi)]^T \tag{124}$$

$$R_y(-\theta) = \left[R_y\,(\theta)\right]^{-1} = \left[R_y\,(\theta)\right]^T \tag{125}$$

$$R_z(-\psi) = [R_z\,(\psi)]^{-1} = [R_z(\psi)]^T \tag{126}$$

■ Angle Notation

When studying rotations there seems to be different notations for the axis angles depending on the topic being adressed. The following shows a list of different angle notations often found in the literature:

Definition: Angle Notation

In a regular right-handed Cartesian coordinate system the symbols denoting the angle rotation around the principal axes are often given the following labels:

Rotation around the x-axis: α, ϕ, r, roll, bank
Rotation around the y-axis: β, θ, p, pitch, elevation
Rotation around the z-axis: γ, ψ, y, yaw, heading

Proper Euler angles: the symbols α, β, γ are often used in this convention.
Tait-Bryan angles: the symbols, ϕ, θ, ψ are often used in this convention.

■ Coordinate System Convention

Left-handed or right-handed coordinate system? As mentioned ealier there is no standard convention specifying which coordinate system to use when working with 3D graphics. Therefore, it is up to the clients using the graphics library or API to adapt to the convention used.

Right-handed coordinate system

The 3D coordinate system shown in figure 1 is right-handed. By convention, positive rotations in a right-handed system are such that, when looking from a positive axis toward the origin, a 90° counterclockwise rotation will transform one positive axis into the other.

Left-handed coordinate system

The 3D coordinate system shown in figure 2 is left-handed. By convention, positive rotations in a left-handed system are such that, when looking from a positive axis toward the origin, a 90° clockwise rotation will transform one positive axis into the other.

Conversion from right to left and left to right

Sometimes conversion between left -and right-handed coordinates are needed. The matrix that converts from points represented in one to points represented in the other is its own inverse, and is given by equation (127). Notice that the effect of multiplying a 3D vector with this matrix is that the x-component in the resulting vector is inverted. This transforma-tion matrix corresponds to converting between the coordinate systems presented in figure 1 and figure 2.

$$M_{R\leftarrow L} = M_{L\leftarrow R} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{127}$$

Using equation (127) on a vector defined in a right-handed coordinate system is equivalent to viewing the first coordinate system in a mirror. Such transformations are known as reflective transformations and will take a right-handed coordinate system into a left-handed coordinate system. The length of any vectors will remain unchanged. The x-component of these vectors will simply be replaced by its negative in the new coordinate system.

Rotation in 2D

Although, the primary focus in this notebook is to study rotations of geometries in three dimensions, we will briefly look at an example of rotating a square in two dimensions.

Example

A square is composed of the four points a = (1, 5), b = (5, 5), c = (1, 1), and d = (5, 1). Determine the coordinates of the four points if the square is rotated by an angle of 15 degree counterclockwise.

Below different solutions to this problem is presented. Initially, the problem is solved manually - that is without using *Mathematica*. Then different solutions using *Mathematica* is presented.

■ Solution - manual

When a point (x, y) is rotated about the origin (0, 0) counterclockwise by an angle θ, the coordinates of the new point are

$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

Thus, when a point (x, y) is rotated about another point (p, q) counterclockwise by an angle θ, we can compute the coordinates of the new point, denoted (x', y'), using the following steps:

1) Translating the entire plane so that (p, q) goes to the origin

2) Perform the rotation

3) Translate the entire plane back

To translate (p, q) to the origin, we subtract p from the x-coordinate and q from the y-coordinate and to compensate for this operation we add p and q instead of subtracting them after having performed the rotation.

$$x' = (x - p)\cos(\theta) - (y - q)\sin(\theta) + p$$
$$y' = (x - p)\sin(\theta) + (y - q)\cos(\theta) + q$$

Using a rotation matrix this can be written as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta \end{pmatrix} \begin{pmatrix} x-p \\ y-q \end{pmatrix} + \begin{pmatrix} p \\ q \end{pmatrix}$$

Using the above equation we can calculate the position of the four points a, b, c, and d after rotating these 15 degrees counterclockwise from the center point (3, 3).

$$\boldsymbol{a} = \begin{pmatrix} \cos(15) & -\sin(15) \\ \sin(15) & \cos(15) \end{pmatrix} \begin{pmatrix} 1-3 \\ 5-3 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} \frac{1+\sqrt{3}}{2\sqrt{2}} & \frac{1-\sqrt{3}}{2\sqrt{2}} \\ \frac{-1+\sqrt{3}}{2\sqrt{2}} & \frac{1+\sqrt{3}}{2\sqrt{2}} \end{pmatrix} \begin{pmatrix} -2 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 3-\sqrt{6} \\ 3+\sqrt{2} \end{pmatrix} \approx \begin{pmatrix} 0.55051 \\ 4.41421 \end{pmatrix}$$

$$\boldsymbol{b} = \begin{pmatrix} \cos(15) & -\sin(15) \\ \sin(15) & \cos(15) \end{pmatrix} \begin{pmatrix} 5-3 \\ 5-3 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} \frac{1+\sqrt{3}}{2\sqrt{2}} & \frac{1-\sqrt{3}}{2\sqrt{2}} \\ \frac{-1+\sqrt{3}}{2\sqrt{2}} & \frac{1+\sqrt{3}}{2\sqrt{2}} \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 3+\sqrt{2} \\ 3+\sqrt{6} \end{pmatrix} \approx \begin{pmatrix} 4.41421 \\ 5.44949 \end{pmatrix}$$

$$\boldsymbol{c} = \begin{pmatrix} \cos(15) & -\sin(15) \\ \sin(15) & \cos(15) \end{pmatrix} \begin{pmatrix} 1-3 \\ 1-3 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} \frac{1+\sqrt{3}}{2\sqrt{2}} & \frac{1-\sqrt{3}}{2\sqrt{2}} \\ \frac{-1+\sqrt{3}}{2\sqrt{2}} & \frac{1+\sqrt{3}}{2\sqrt{2}} \end{pmatrix} \begin{pmatrix} -2 \\ -2 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 3-\sqrt{2} \\ 3-\sqrt{6} \end{pmatrix} \approx \begin{pmatrix} 1.58579 \\ 0.55051 \end{pmatrix}$$

$$\boldsymbol{d} = \begin{pmatrix} \cos(15) & -\sin(15) \\ \sin(15) & \cos(15) \end{pmatrix} \begin{pmatrix} 5-3 \\ 1-3 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} \frac{1+\sqrt{3}}{2\sqrt{2}} & \frac{1-\sqrt{3}}{2\sqrt{2}} \\ \frac{-1+\sqrt{3}}{2\sqrt{2}} & \frac{1+\sqrt{3}}{2\sqrt{2}} \end{pmatrix} \begin{pmatrix} 2 \\ -2 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 3+\sqrt{6} \\ 3-\sqrt{2} \end{pmatrix} \approx \begin{pmatrix} 5.44949 \\ 1.58579 \end{pmatrix}$$

■ Solution - using *Mathematica* and RotationTransform

Let us define a function that allows us to create a square given the length of the sides and the center point of the square.

In[1]:=
```
createSquare[edgeLength_?NumericQ,
  center : {xCenter_?NumericQ, yCenter_?NumericQ}] := Module[
       (* ALLOCATE LOCAL VARIABLES *)
       {points, p0, p1, p2, p3},

       (* DEFINE POINTS *)

  p0 = {xCenter - (1 / 2) * edgeLength, yCenter + (1 / 2) * edgeLength};

  p1 = {xCenter + (1 / 2) * edgeLength, yCenter + (1 / 2) * edgeLength};

  p2 = {xCenter - (1 / 2) * edgeLength, yCenter - (1 / 2) * edgeLength};

  p3 = {xCenter + (1 / 2) * edgeLength, yCenter - (1 / 2) * edgeLength};
       points = {p0, p1, p2, p3}
    ]
```

To rotate the square around a given point, we define a function that takes the square points as input and rotates these an angle counterclockwise around a specified point. Note that we use *Mathematica's* internal function *RotationTransform* to implement the rotation.

In[2]:=
```
rotateSquare[points_, angle_?NumericQ,
   pivot : {xCenter_?NumericQ, yCenter_?NumericQ}] :=
  Map[RotationTransform[angle Degree, pivot], points];
```

In order to visualize the square before and after the rotation we define a function that allows us to display the square given a list of points representing the vertices of the square.

In[3]:=
```
drawSquare[points_] /; (Length[points] == 4) := Module[
      (* ALLOCATE LOCAL VARIABLES *)
      { p0, p1, p2, p3, pCenter},

      (* CREATE VERTICES *)
      vertex0 = Point[points[[1]]];
      vertex1 = Point[points[[2]]];
      vertex2 = Point[points[[3]]];
      vertex3 = Point[points[[4]]];
      vertexCenter = Point[(points[[1]] + points[[4]]) / 2];

      (* CREATE EDGES *)
      edge01 = Line[{points[[1]], points[[2]]}];
      edge02 = Line[{points[[1]], points[[3]]}];
      edge13 = Line[{points[[2]], points[[4]]}];
      edge23 = Line[{points[[3]], points[[4]]}];

      (* CREATE GRAPHICS FOR VERTICES *)
      p0 = Graphics[{RGBColor[1, 0, 0], PointSize[0.1], vertex0}];
      p1 = Graphics[{RGBColor[0, 1, 0], PointSize[0.1], vertex1}];
      p2 = Graphics[{RGBColor[0, 0, 1], PointSize[0.1], vertex2}];
      p3 = Graphics[{RGBColor[1, 1, 0], PointSize[0.1], vertex3}];

  pCenter = Graphics[{RGBColor[0, 0, 0], PointSize[0.03], vertexCenter}];

      (* CREATE GRAPHICS FOR EDGES *)
      e01 = Graphics[{RGBColor[0, 0, 0], Thickness[0.01], edge01}];
      e02 = Graphics[{RGBColor[0, 0, 0], Thickness[0.01], edge02}];
      e13 = Graphics[{RGBColor[0, 0, 0], Thickness[0.01], edge13}];
      e23 = Graphics[{RGBColor[0, 0, 0], Thickness[0.01], edge23}];

      Show[{p0, p1, p2, p3, pCenter, e01, e02, e13, e23},
   AxesLabel → {"x", "y"}, Axes → True, AspectRatio → 1,
   AxesOrigin → {0, 0}, ImageSize → Automatic]
 ]
```

Let us test our functions. First we create a square with a length of 4 centered at (3, 3).

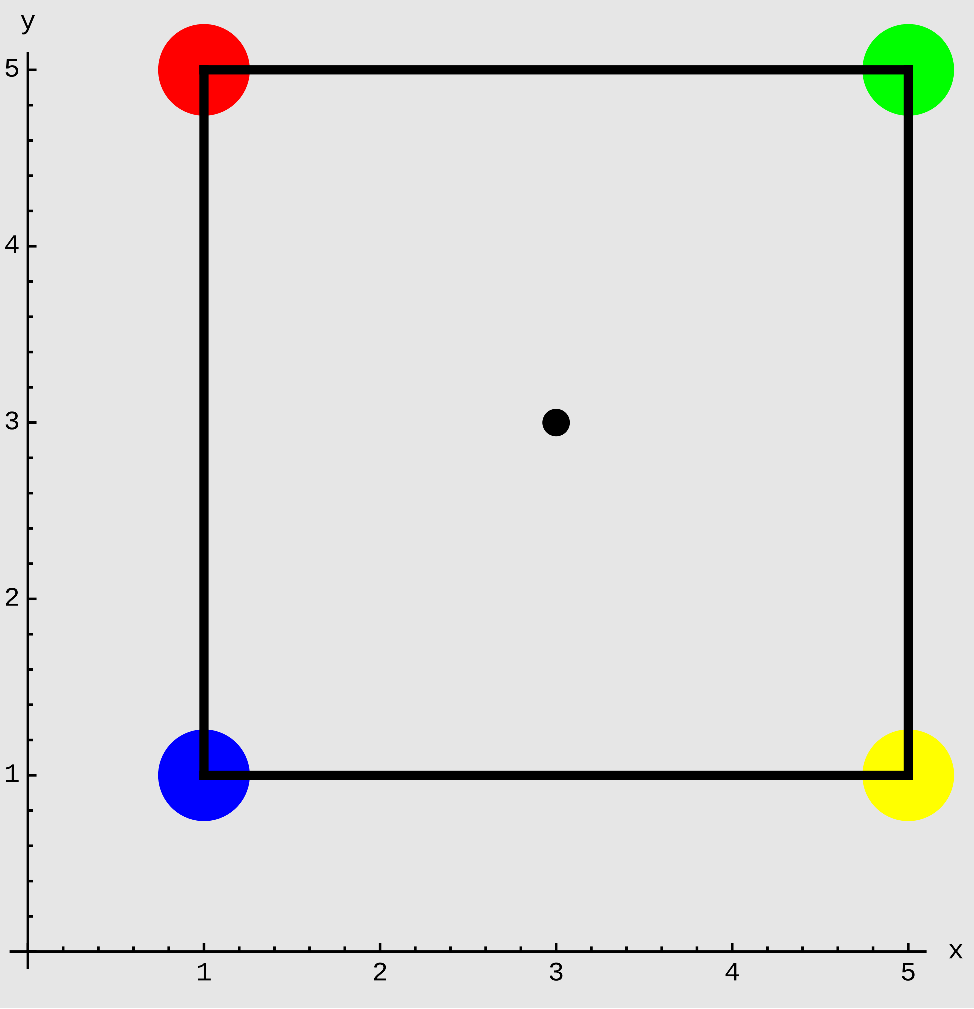
In[4]:=
```
squarePoints = createSquare[4, {3, 3}]
```

Out[4]=
```
{{1, 5}, {5, 5}, {1, 1}, {5, 1}}
```

To visualize our square we call our *drawSquare* function.

In[5]:=
```
drawSquare[squarePoints]
```

Out[5]=



We would like to rotate the square 15 degrees counterclockwise from the center point (3, 3). To accomplish this we call our *rotateSquare* function.

In[6]:=

```
newPoints = rotateSquare[squarePoints, 15, {3, 3}]
```

Out[6]=

$$\left\{\left\{-\frac{3}{2}\left(-2+\sqrt{2}\right)-\frac{5\left(-1+\sqrt{3}\right)}{2\sqrt{2}}+\frac{1+\sqrt{3}}{2\sqrt{2}},\ \frac{-1+\sqrt{3}}{2\sqrt{2}}+\frac{5\left(1+\sqrt{3}\right)}{2\sqrt{2}}-\frac{3}{2}\left(-2+\sqrt{6}\right)\right\},\right.$$

$$\left\{-\frac{3}{2}\left(-2+\sqrt{2}\right)-\frac{5\left(-1+\sqrt{3}\right)}{2\sqrt{2}}+\frac{5\left(1+\sqrt{3}\right)}{2\sqrt{2}},\right.$$

$$\left.\frac{5\left(-1+\sqrt{3}\right)}{2\sqrt{2}}+\frac{5\left(1+\sqrt{3}\right)}{2\sqrt{2}}-\frac{3}{2}\left(-2+\sqrt{6}\right)\right\},$$

$$\left\{-\frac{3}{2}\left(-2+\sqrt{2}\right)-\frac{-1+\sqrt{3}}{2\sqrt{2}}+\frac{1+\sqrt{3}}{2\sqrt{2}},\ \frac{-1+\sqrt{3}}{2\sqrt{2}}+\frac{1+\sqrt{3}}{2\sqrt{2}}-\frac{3}{2}\left(-2+\sqrt{6}\right)\right\},$$

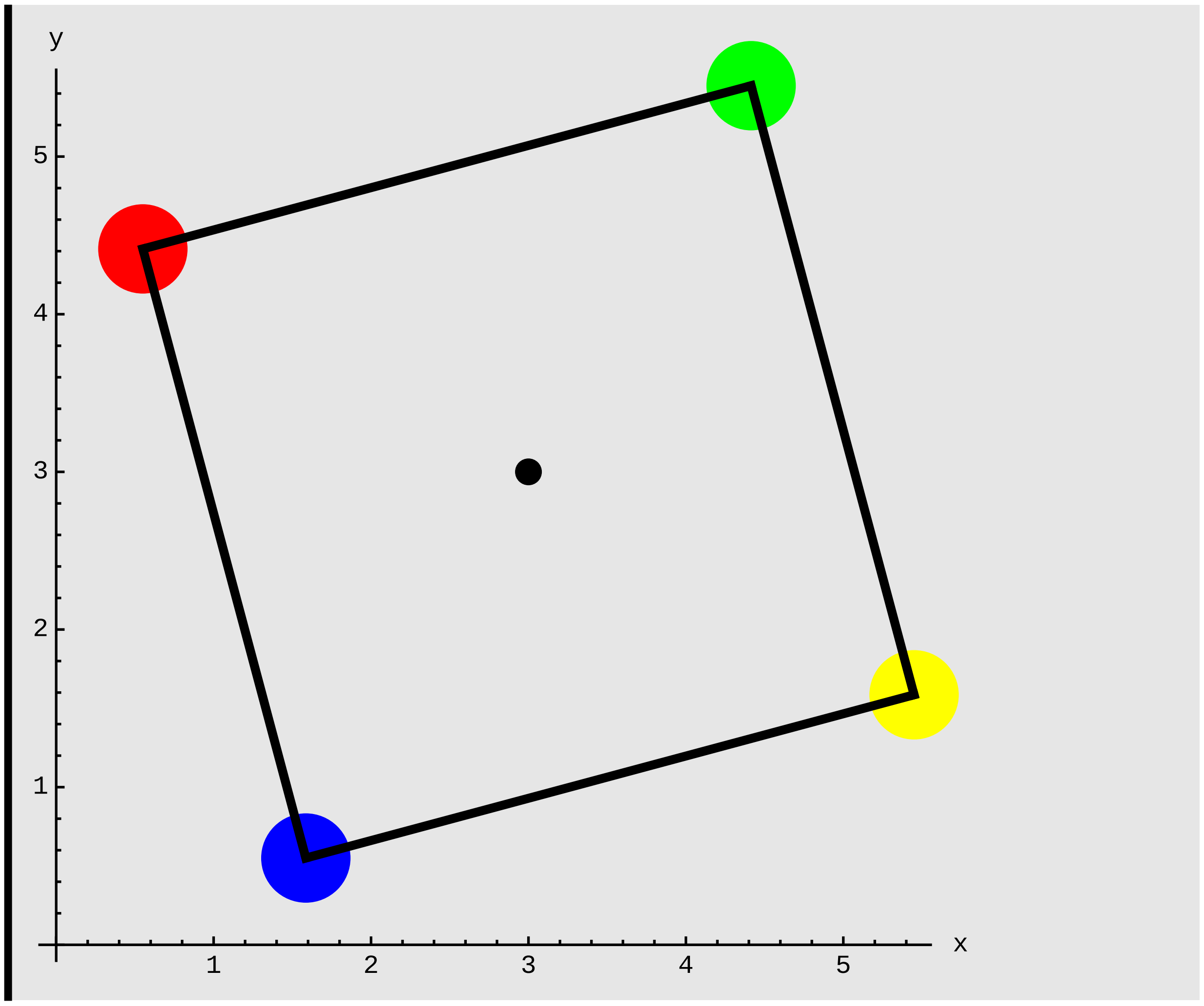
$$\left.\left\{-\frac{3}{2}\left(-2+\sqrt{2}\right)-\frac{-1+\sqrt{3}}{2\sqrt{2}}+\frac{5\left(1+\sqrt{3}\right)}{2\sqrt{2}},\ \frac{5\left(-1+\sqrt{3}\right)}{2\sqrt{2}}+\frac{1+\sqrt{3}}{2\sqrt{2}}-\frac{3}{2}\left(-2+\sqrt{6}\right)\right\}\right\}$$

Let us visualize the result of the rotation.

In[7]:=

```
drawSquare[newPoints]
```

Out[7]=



■ Solution - using *Mathematica* without using RotationTransform

In the solution presented above we use the internal function *RotationTransform* in order to perform the rotation of the square. Below we present an alternative solution which accomplishes the same result, but without using *RotationTransform* instead we use the rotation matrix given below:

$$R = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta \end{pmatrix}$$

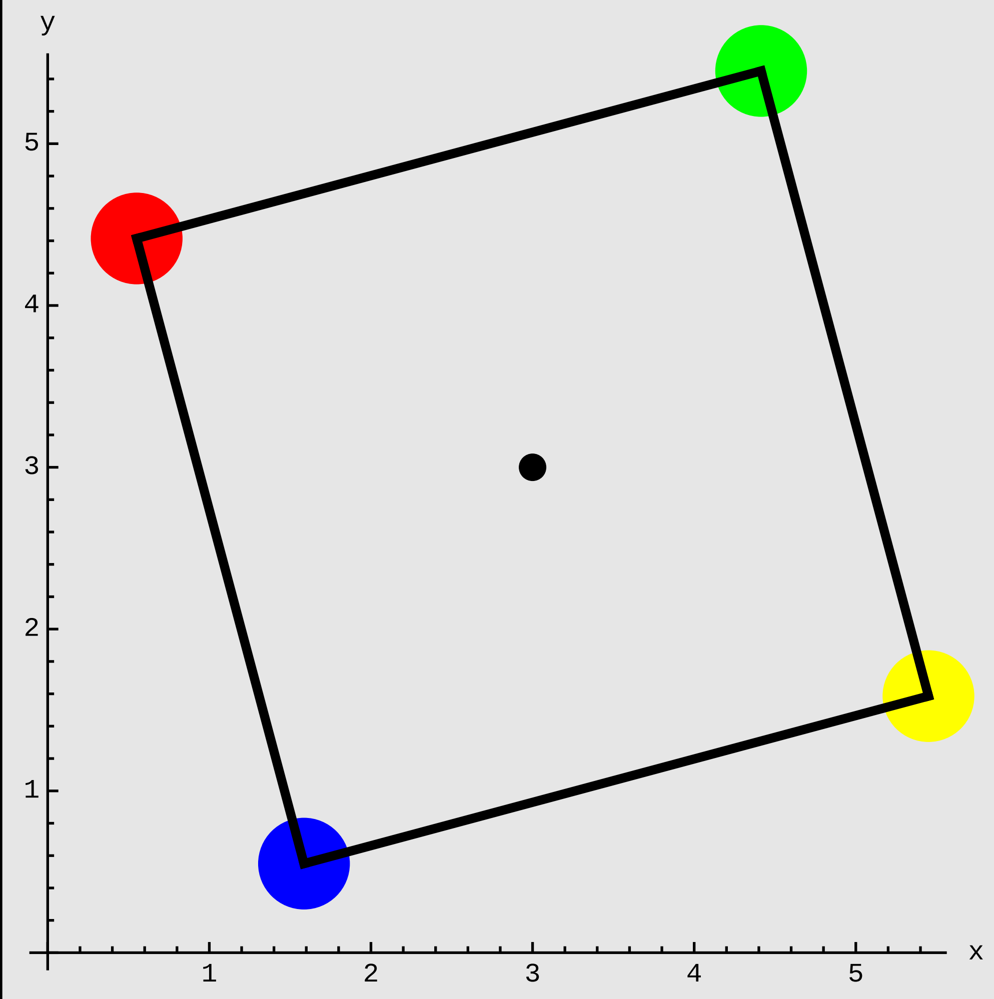
In[8]:=
```
rotateSquareAlternative[points_, angle_?NumericQ,
   pivot : {xCenter_?NumericQ, yCenter_?NumericQ}] /;
  (Length[points] == 4) := Module[
     (* ALLOCATE LOCAL VARIABLES *)
     { p0, p1, p2, p3},

     (* CREATE POINTS *)
     p0 = points[[1]];
     p1 = points[[2]];
     p2 = points[[3]];
     p3 = points[[4]];

     Rmat = {{Cos[angle Degree], -Sin[angle Degree]},
    {Sin[angle Degree], Cos[angle Degree]}};
     Table[Rmat.(points[[i]] - pivot) + pivot, {i, 1, Length[points]}]
 ]
```

In[9]:=
```
newPointsAlternative =
 rotateSquareAlternative[squarePoints, 15, {3, 3}] // N
```

Out[9]=
```
{{0.55051, 4.41421}, {4.41421, 5.44949},
 {1.58579, 0.55051}, {5.44949, 1.58579}}
```

In[10]:=
```
drawSquare[newPointsAlternative]
```

Out[10]=



Lets us verify that the two solutions give the same result

In[11]:=
```
newPointsAlternative == newPoints
```

Out[11]=
```
True
```

Rotation Matrices

Defining rotation matrices

■ Rotating angle ψ around X

In[12]:=
```
RmatPsiAroundX = RotationMatrix[ψ, {1, 0, 0}];
RmatPsiAroundX // MatrixForm
```

Out[13]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \text{Cos}[\psi] & -\text{Sin}[\psi] \\ 0 & \text{Sin}[\psi] & \text{Cos}[\psi] \end{pmatrix}$$

■ Rotating angle ψ around Y

In[14]:=
```
RmatPsiAroundY = RotationMatrix[ψ, {0, 1, 0}];
RmatPsiAroundY // MatrixForm
```

Out[15]//MatrixForm=

$$\begin{pmatrix} \text{Cos}[\psi] & 0 & \text{Sin}[\psi] \\ 0 & 1 & 0 \\ -\text{Sin}[\psi] & 0 & \text{Cos}[\psi] \end{pmatrix}$$

■ Rotating angle ψ around Z

In[16]:=
```
RmatPsiAroundZ = RotationMatrix[ψ, {0, 0, 1}];
RmatPsiAroundZ // MatrixForm
```

Out[17]//MatrixForm=

$$\begin{pmatrix} \text{Cos}[\psi] & -\text{Sin}[\psi] & 0 \\ \text{Sin}[\psi] & \text{Cos}[\psi] & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

■ Rotating angle θ around X

In[18]:=
```
RmatThetaAroundX = RotationMatrix[θ, {1, 0, 0}];
RmatThetaAroundX // MatrixForm
```
Out[19]//MatrixForm=
```
⎛1    0       0    ⎞
⎜0  Cos[θ]  -Sin[θ]⎟
⎝0  Sin[θ]   Cos[θ]⎠
```

■ Rotating angle θ around Y

In[20]:=
```
RmatThetaAroundY = RotationMatrix[θ, {0, 1, 0}];
RmatThetaAroundY // MatrixForm
```
Out[21]//MatrixForm=
```
⎛ Cos[θ]   0  Sin[θ]⎞
⎜   0      1    0   ⎟
⎝-Sin[θ]   0  Cos[θ]⎠
```

■ Rotating angle θ around Z

In[22]:=
```
RmatThetaAroundZ = RotationMatrix[θ, {0, 0, 1}];
RmatThetaAroundZ // MatrixForm
```
Out[23]//MatrixForm=
```
⎛Cos[θ]  -Sin[θ]  0⎞
⎜Sin[θ]   Cos[θ]  0⎟
⎝  0        0     1⎠
```

■ Rotating angle ϕ around X

In[24]:=
```
RmatPhiAroundX = RotationMatrix[ϕ, {1, 0, 0}];
RmatPhiAroundX // MatrixForm
```
Out[25]//MatrixForm=
```
⎛1    0       0    ⎞
⎜0  Cos[ϕ]  -Sin[ϕ]⎟
⎝0  Sin[ϕ]   Cos[ϕ]⎠
```

■ Rotating angle ϕ around Y

In[26]:=
```
RmatPhiAroundY = RotationMatrix[ϕ, {0, 1, 0}];
RmatPhiAroundY // MatrixForm
```

Out[27]//MatrixForm=
```
( Cos[ϕ]  0 Sin[ϕ] )
(   0     1   0    )
(-Sin[ϕ]  0 Cos[ϕ] )
```

■ Rotating angle ϕ around Z

In[28]:=
```
RmatPhiAroundZ = RotationMatrix[ϕ, {0, 0, 1}];
RmatPhiAroundZ // MatrixForm
```

Out[29]//MatrixForm=
```
(Cos[ϕ]  -Sin[ϕ]  0)
(Sin[ϕ]   Cos[ϕ]  0)
(  0        0     1)
```

Rotation using *proper* Euler angles

Euler X - Z - X

■ Euler X-Z-X (Rotating ψ around X, θ around Z, ϕ around X)

In[30]:=
```
RmatXZX = RmatPsiAroundX.RmatThetaAroundZ.RmatPhiAroundX;
RmatXZX // MatrixForm
```

Out[31]//MatrixForm=
```
(    Cos[θ]                      -Cos[ϕ] Sin[θ]                                  Sin[θ] Sin[ϕ]
(Cos[ψ] Sin[θ]  Cos[θ] Cos[ϕ] Cos[ψ] - Sin[ϕ] Sin[ψ]  -Cos[θ] Cos[ψ] Sin[ϕ] - Cos
(Sin[θ] Sin[ψ]  Cos[ψ] Sin[ϕ] + Cos[θ] Cos[ϕ] Sin[ψ]   Cos[ϕ] Cos[ψ] - Cos[θ] Sin
```

■ Euler X-Z-X (Rotating ψ around X, θ around Z, ϕ around X) - using RotationTransform

In[32]:=
```
RotationTransform[ψ, {1, 0, 0}].RotationTransform[θ, {0, 0, 1}].
 RotationTransform[ϕ, {1, 0, 0}]
```

Out[32]=
```
                                   Cos[θ]            -Cos[ϕ] Sin[θ]              
                              Cos[ψ] Sin[θ]      Cos[θ] Cos[ϕ] Cos[ψ] -        -Co
                                                   Sin[ϕ] Sin[ψ]                Co
TransformationFunction[                                                            
                              Sin[θ] Sin[ψ]      Cos[ψ] Sin[ϕ] +               Cos
                                                   Cos[θ] Cos[ϕ] Sin[ψ]          Co
                              ─────────────────────────────────────────────────────
                                    0                   0
```

In[33]:=
```
RmatZYX = RmatPhiAroundZ.RmatThetaAroundY.RmatPsiAroundX;
RmatZYX // MatrixForm
```

Out[34]//MatrixForm=
```
( Cos[θ] Cos[ϕ]  -Cos[ψ] Sin[ϕ] + Cos[ϕ] Sin[θ] Sin[ψ]  Cos[ϕ] Cos[ψ] Sin[θ] + Sir
  Cos[θ] Sin[ϕ]   Cos[ϕ] Cos[ψ] + Sin[θ] Sin[ϕ] Sin[ψ]  Cos[ψ] Sin[θ] Sin[ϕ] - Cos
    -Sin[θ]                   Cos[θ] Sin[ψ]                        Cos[θ] Cos[ψ]
```

Euler X - Y - X

■ Euler X-Y-X (Rotating ψ around X, θ around Y, ϕ around X)

In[35]:=
```
RmatXYX = RmatPsiAroundX.RmatThetaAroundY.RmatPhiAroundX;
RmatXYX // MatrixForm
```

Out[36]//MatrixForm=
```
(     Cos[θ]                    Sin[θ] Sin[ϕ]                          Cos[ϕ] Sin[θ]
  Sin[θ] Sin[ψ]  Cos[ϕ] Cos[ψ] - Cos[θ] Sin[ϕ] Sin[ψ]  -Cos[ψ] Sin[ϕ] - Cos[θ] Cc
 -Cos[ψ] Sin[θ]  Cos[θ] Cos[ψ] Sin[ϕ] + Cos[ϕ] Sin[ψ]   Cos[θ] Cos[ϕ] Cos[ψ] - Si
```

- **Euler X-Y-X (Rotating ψ around X, θ around Y, ϕ around X) - using RotationTransform**

In[37]:=
```
RotationTransform[ψ, {1, 0, 0}].RotationTransform[θ, {0, 1, 0}].
 RotationTransform[ϕ, {1, 0, 0}]
```

Out[37]=
```
                            Cos[θ]               Sin[θ] Sin[ϕ]
                        Sin[θ] Sin[ψ]        Cos[ϕ] Cos[ψ] -              -C
                                               Cos[θ] Sin[ϕ] Sin[ψ]        C
TransformationFunction[
                       -Cos[ψ] Sin[θ]       Cos[θ] Cos[ψ] Sin[ϕ] +        Cos
                                               Cos[ϕ] Sin[ψ]               S
                       ________________________________________________________
                             0                      0
```

Euler Y - X - Y

- **Euler Y-X-Y (Rotating ψ around Y, θ around X, ϕ around Y)**

In[38]:=
```
RmatYXY = RmatPsiAroundY.RmatThetaAroundX.RmatPhiAroundY;
RmatYXY // MatrixForm
```

Out[39]//MatrixForm=
```
( Cos[ϕ] Cos[ψ] - Cos[θ] Sin[ϕ] Sin[ψ]   Sin[θ] Sin[ψ]  Cos[ψ] Sin[ϕ] + Cos[θ] Cos
              Sin[θ] Sin[ϕ]                 Cos[θ]              -Cos[ϕ] Sin[θ]
 -Cos[θ] Cos[ψ] Sin[ϕ] - Cos[ϕ] Sin[ψ]   Cos[ψ] Sin[θ]  Cos[θ] Cos[ϕ] Cos[ψ] - Sir
```

- **Euler Y-X-Y (Rotating ψ around Y, θ around X, ϕ around Y) - using RotationTransform**

In[40]:=
```
RotationTransform[ψ, {0, 1, 0}].RotationTransform[θ, {1, 0, 0}].
 RotationTransform[ϕ, {0, 1, 0}]
```

Out[40]=
```
                            Cos[ϕ] Cos[ψ] -            Sin[θ] Sin[ψ]       Cos
                              Cos[θ] Sin[ϕ] Sin[ψ]                          C
                                Sin[θ] Sin[ϕ]              Cos[θ]
TransformationFunction[
                          -Cos[θ] Cos[ψ] Sin[ϕ] -      Cos[ψ] Sin[θ]       Cos
                            Cos[ϕ] Sin[ψ]                                   Si
                       ________________________________________________________
                                    0                        0
```

Euler Y - Z - Y

■ Euler Y-Z-Y (Rotating ψ around Y, θ around Z, ϕ around Y)

In[41]:=
```
RmatYZY = RmatPsiAroundY.RmatThetaAroundZ.RmatPhiAroundY;
RmatYZY // MatrixForm
```

Out[42]//MatrixForm=
```
( Cos[θ] Cos[ϕ] Cos[ψ] - Sin[ϕ] Sin[ψ]   -Cos[ψ] Sin[θ]  Cos[θ] Cos[ψ] Sin[ϕ] + Co
           Cos[ϕ] Sin[θ]                     Cos[θ]                 Sin[θ] Sin[ϕ
 -Cos[ψ] Sin[ϕ] - Cos[θ] Cos[ϕ] Sin[ψ]   Sin[θ] Sin[ψ]   Cos[ϕ] Cos[ψ] - Cos[θ] Si
```

■ Euler Y-Z-Y (Rotating ψ around Y, θ around Z, ϕ around Y) - using RotationTransform

In[43]:=
```
RotationTransform[ψ, {0, 1, 0}].RotationTransform[θ, {0, 0, 1}].
 RotationTransform[ϕ, {0, 1, 0}]
```

Out[43]=
```
                              Cos[θ] Cos[ϕ] Cos[ψ] -      -Cos[ψ] Sin[θ]     Cos
                               Sin[ϕ] Sin[ψ]                                  Co
                                 Cos[ϕ] Sin[θ]               Cos[θ]
TransformationFunction[       -Cos[ψ] Sin[ϕ] -            Sin[θ] Sin[ψ]      Cos
                               Cos[θ] Cos[ϕ] Sin[ψ]                           Co
                              ______________________________________________________
                                      0                        0
```

Euler Z - Y - Z

■ Euler Z-Y-Z (Rotating ψ around Z, θ around Y, ϕ around Z)

In[44]:=
```
RmatZYZ = RmatPsiAroundZ.RmatThetaAroundY.RmatPhiAroundZ;
RmatZYZ // MatrixForm
```

Out[45]//MatrixForm=
```
(Cos[θ] Cos[ϕ] Cos[ψ] - Sin[ϕ] Sin[ψ]  -Cos[θ] Cos[ψ] Sin[ϕ] - Cos[ϕ] Sin[ψ]  Cos
 Cos[ψ] Sin[ϕ] + Cos[θ] Cos[ϕ] Sin[ψ]   Cos[ϕ] Cos[ψ] - Cos[θ] Sin[ϕ] Sin[ψ]  Sin
          -Cos[ϕ] Sin[θ]                           Sin[θ] Sin[ϕ]
```

■ Rotating ψ around Z, θ around Y, and ϕ around Z - using RotationTransform

In[46]:=
```
RotationTransform[ψ, {0, 0, 1}].RotationTransform[θ, {0, 1, 0}].
 RotationTransform[ϕ, {0, 0, 1}]
```

Out[46]=
```
                                Cos[θ] Cos[ϕ] Cos[ψ] -          -Cos[θ] Cos[ψ] Sin
                                  Sin[ϕ] Sin[ψ]                  Cos[ϕ] Sin[ψ]
TransformationFunction[         Cos[ψ] Sin[ϕ] +                 Cos[ϕ] Cos[ψ] -
                                  Cos[θ] Cos[ϕ] Sin[ψ]            Cos[θ] Sin[ϕ] Si
                                    -Cos[ϕ] Sin[θ]                   Sin[θ] Sin[ϕ]
                              ----------------------------------------------------
                                          0                              0
```

Euler Z - X - Z

■ Euler Z-X-Z (Rotating ψ around Z, θ around X, ϕ around Z)

In[47]:=
```
RmatZXZ = RmatPsiAroundZ.RmatThetaAroundX.RmatPhiAroundZ;
RmatZXZ // MatrixForm
```

Out[48]//MatrixForm=
```
( Cos[ϕ] Cos[ψ] - Cos[θ] Sin[ϕ] Sin[ψ]  -Cos[ψ] Sin[ϕ] - Cos[θ] Cos[ϕ] Sin[ψ]  Si
  Cos[θ] Cos[ψ] Sin[ϕ] + Cos[ϕ] Sin[ψ]   Cos[θ] Cos[ϕ] Cos[ψ] - Sin[ϕ] Sin[ψ]  -Co
              Sin[θ] Sin[ϕ]                          Cos[ϕ] Sin[θ]
```

■ Euler Z-X-Z (Rotating ψ around Z, θ around X, ϕ around Z) - using RotationTransform

In[49]:=
```
RotationTransform[ψ, {0, 0, 1}].RotationTransform[θ, {1, 0, 0}].
 RotationTransform[ϕ, {0, 0, 1}]
```

Out[49]=
```
                                Cos[ϕ] Cos[ψ] -                 -Cos[ψ] Sin[ϕ] -
                                  Cos[θ] Sin[ϕ] Sin[ψ]            Cos[θ] Cos[ϕ] Sin
TransformationFunction[         Cos[θ] Cos[ψ] Sin[ϕ] +          Cos[θ] Cos[ϕ] Cos[ψ
                                  Cos[ϕ] Sin[ψ]                   Sin[ϕ] Sin[ψ]
                                    Sin[θ] Sin[ϕ]                    Cos[ϕ] Sin[θ]
                              ----------------------------------------------------
                                          0                              0
```

Rotation using Tait–Bryan angles

Tait-Bryan X - Z - Y

■ Tait-Bryan X-Z-Y (Rotating ψ around X, ϕ around Z, θ around Y)

In[50]:=

```
RmatXZY = RmatPsiAroundX.RmatPhiAroundZ.RmatThetaAroundY;
RmatXZY // MatrixForm
```

Out[51]//MatrixForm=

```
            Cos[θ] Cos[ϕ]                      -Sin[ϕ]                 Cos[ϕ] Sin[θ]
 Cos[θ] Cos[ψ] Sin[ϕ] + Sin[θ] Sin[ψ]   Cos[ϕ] Cos[ψ]   Cos[ψ] Sin[θ] Sin[ϕ] - Cos
-Cos[ψ] Sin[θ] + Cos[θ] Sin[ϕ] Sin[ψ]   Cos[ϕ] Sin[ψ]   Cos[θ] Cos[ψ] + Sin[θ] Sir
```

■ Tait-Bryan X-Z-Y (Rotating ψ around X, ϕ around Z, θ around Y) - using RotationTransForm

In[52]:=

```
RotationTransform[ψ, {1, 0, 0}].RotationTransform[ϕ, {0, 0, 1}].
 RotationTransform[θ, {0, 1, 0}]
```

Out[52]=

```
                            Cos[θ] Cos[ϕ]             -Sin[ϕ]
                      Cos[θ] Cos[ψ] Sin[ϕ] +      Cos[ϕ] Cos[ψ]      Cos
                        Sin[θ] Sin[ψ]                                  Cc
TransformationFunction[
                      -Cos[ψ] Sin[θ] +            Cos[ϕ] Sin[ψ]      Cos
                        Cos[θ] Sin[ϕ] Sin[ψ]                           Si
                      ----------------------------------------------------
                                 0                      0
```

Tait-Bryan X - Y - Z

■ Tait-Bryan X-Y-Z (Rotating ψ around X, θ around Y, ϕ around Z)

In[53]:=

```
RmatXYZ = RmatPsiAroundX.RmatThetaAroundY.RmatPhiAroundZ;
RmatXYZ // MatrixForm
```

Out[54]//MatrixForm=

```
⎛              Cos[θ] Cos[ϕ]                          -Cos[θ] Sin[ϕ]
  Cos[ψ] Sin[ϕ] + Cos[ϕ] Sin[θ] Sin[ψ]   Cos[ϕ] Cos[ψ] - Sin[θ] Sin[ϕ] Sin[ψ]   -Cc
⎝ -Cos[ϕ] Cos[ψ] Sin[θ] + Sin[ϕ] Sin[ψ]  Cos[ψ] Sin[θ] Sin[ϕ] + Cos[ϕ] Sin[ψ]   Co
```

■ Tait-Bryan X-Y-Z (Rotating ψ around X, θ around Y, ϕ around Z) - using RotationTransForm

In[55]:=

```
RotationTransform[ψ, {1, 0, 0}].RotationTransform[θ, {0, 1, 0}].
 RotationTransform[ϕ, {0, 0, 1}]
```

Out[55]=

```
                                 ⎛       Cos[θ] Cos[ϕ]                 -Cos[θ] Sin[ϕ]
                                   Cos[ψ] Sin[ϕ] +               Cos[ϕ] Cos[ψ] -
                                     Cos[ϕ] Sin[θ] Sin[ψ]          Sin[θ] Sin[ϕ] Sin
TransformationFunction[
                                   -Cos[ϕ] Cos[ψ] Sin[θ] +       Cos[ψ] Sin[θ] Sin[ç
                                    Sin[ϕ] Sin[ψ]                  Cos[ϕ] Sin[ψ]
                                   ─────────────────────────────────────────────────
                                 ⎝             0                          0
```

Tait-Bryan Y - X - Z

■ Tait-Bryan Y-X-Z (Rotation θ around Y, ψ around X, ϕ around Z)

In[56]:=

```
RmatYXZ = RmatThetaAroundY.RmatPsiAroundX.RmatPhiAroundZ;
RmatYXZ // MatrixForm
```

Out[57]//MatrixForm=

```
⎛ Cos[θ] Cos[ϕ] + Sin[θ] Sin[ϕ] Sin[ψ]   -Cos[θ] Sin[ϕ] + Cos[ϕ] Sin[θ] Sin[ψ]  Cc
              Cos[ψ] Sin[ϕ]                          Cos[ϕ] Cos[ψ]
⎝ -Cos[ϕ] Sin[θ] + Cos[θ] Sin[ϕ] Sin[ψ]   Sin[θ] Sin[ϕ] + Cos[θ] Cos[ϕ] Sin[ψ]   Cc
```

■ Tait-Bryan Y-X-Z (Rotation θ around Y, ψ around X, ϕ around Z) - using RotationTransForm

In[58]:=
```
RotationTransform[θ, {0, 1, 0}].RotationTransform[ψ, {1, 0, 0}].
 RotationTransform[ϕ, {0, 0, 1}]
```

Out[58]=
```
                                  Cos[θ] Cos[ϕ] +                  -Cos[θ] Sin[ϕ] +
                                   Sin[θ] Sin[ϕ] Sin[ψ]             Cos[ϕ] Sin[θ] Si
                                      Cos[ψ] Sin[ϕ]                    Cos[ϕ] Cos[ψ]
TransformationFunction[           -Cos[ϕ] Sin[θ] +                  Sin[θ] Sin[ϕ] +
                                   Cos[θ] Sin[ϕ] Sin[ψ]             Cos[θ] Cos[ϕ] Si
                              ─────────────────────────────────────────────────────
                                            0                              0
```

Tait-Bryan Y - Z - X

■ Tait-Bryan Y-Z-X (Rotating θ around Y, ϕ around Z, ψ around X)

In[59]:=
```
RmatYZX = RmatThetaAroundY.RmatPhiAroundZ.RmatPsiAroundX;
RmatYZX // MatrixForm
```

Out[60]//MatrixForm=
```
 Cos[θ] Cos[ϕ]   -Cos[θ] Cos[ψ] Sin[ϕ] + Sin[θ] Sin[ψ]  Cos[ψ] Sin[θ] + Cos[θ] Si
     Sin[ϕ]                  Cos[ϕ] Cos[ψ]                         -Cos[ϕ] Sin[ψ
-Cos[ϕ] Sin[θ]    Cos[ψ] Sin[θ] Sin[ϕ] + Cos[θ] Sin[ψ]  Cos[θ] Cos[ψ] - Sin[θ] Si
```

■ Tait-Bryan Y-Z-X (Rotating θ around Y, ϕ around Z, ψ around X) - using RotationTransForm

In[61]:=
```
RotationTransform[θ, {0, 1, 0}].RotationTransform[ϕ, {0, 0, 1}].
 RotationTransform[ψ, {1, 0, 0}]
```

Out[61]=
```
                              Cos[θ] Cos[ϕ]      -Cos[θ] Cos[ψ] Sin[ϕ] +        Cos
                                                   Sin[θ] Sin[ψ]                 Co
                                 Sin[ϕ]              Cos[ϕ] Cos[ψ]
TransformationFunction[      -Cos[ϕ] Sin[θ]       Cos[ψ] Sin[θ] Sin[ϕ] +        Cos
                                                   Cos[θ] Sin[ψ]                 S
                              ─────────────────────────────────────────────────────
                                   0                     0
```

Tait-Bryan Z - Y - X

■ Tait-Bryan Z-Y-X (Rotating ϕ around Z, θ around Y, ψ around X)

In[62]:=
```
RmatZYX = RmatPhiAroundZ.RmatThetaAroundY.RmatPsiAroundX;
RmatZYX // MatrixForm
```

Out[63]//MatrixForm=
```
( Cos[θ] Cos[ϕ]   -Cos[ψ] Sin[ϕ] + Cos[ϕ] Sin[θ] Sin[ψ]   Cos[ϕ] Cos[ψ] Sin[θ] + Sir
  Cos[θ] Sin[ϕ]    Cos[ϕ] Cos[ψ] + Sin[θ] Sin[ϕ] Sin[ψ]   Cos[ψ] Sin[θ] Sin[ϕ] - Cos
    -Sin[θ]                  Cos[θ] Sin[ψ]                          Cos[θ] Cos[ψ]
```

■ Tait-Bryan Z-Y-X (Rotating ϕ around Z, θ around Y, ψ around X) - using RotationTransForm

In[64]:=
```
RotationTransform[ϕ, {0, 0, 1}].RotationTransform[θ, {0, 1, 0}].
 RotationTransform[ψ, {1, 0, 0}]
```

Out[64]=
```
                                  ( Cos[θ] Cos[ϕ]      -Cos[ψ] Sin[ϕ] +            Cos
                                                        Cos[ϕ] Sin[θ] Sin[ψ]        Si
TransformationFunction[             Cos[θ] Sin[ϕ]      Cos[ϕ] Cos[ψ] +             Cos
                                                        Sin[θ] Sin[ϕ] Sin[ψ]        Cc
                                       -Sin[θ]             Cos[θ] Sin[ψ]
                                   ───────────────────────────────────────────────────
                                         0                     0
```

Tait-Bryan Z - X - Y

■ Tait-Bryan Z-X-Y (Rotating ϕ around Z, ψ around X, θ around Y)

In[65]:=
```
RmatZXY = RmatPhiAroundZ.RmatPsiAroundX.RmatThetaAroundY;
RmatZXY // MatrixForm
```

Out[66]//MatrixForm=
```
( Cos[θ] Cos[ϕ] - Sin[θ] Sin[ϕ] Sin[ψ]   -Cos[ψ] Sin[ϕ]   Cos[ϕ] Sin[θ] + Cos[θ] Sir
  Cos[θ] Sin[ϕ] + Cos[ϕ] Sin[θ] Sin[ψ]    Cos[ϕ] Cos[ψ]   Sin[θ] Sin[ϕ] - Cos[θ] Cos
              -Cos[ψ] Sin[θ]                  Sin[ψ]               Cos[θ] Cos[ψ]
```

■ Tait-Bryan Z-X-Y (Rotating ϕ around Z, ψ around X, θ around Y) - using RotationTransForm

In[67]:=
```
RotationTransform[ϕ, {0, 0, 1}].RotationTransform[ψ, {1, 0, 0}].
 RotationTransform[θ, {0, 1, 0}]
```

Out[67]=
```
                           ⎛ Cos[θ] Cos[ϕ] -              -Cos[ψ] Sin[ϕ]     Cos
                           ⎜  Sin[θ] Sin[ϕ] Sin[ψ]                             C
TransformationFunction⎡    ⎜ Cos[θ] Sin[ϕ] +               Cos[ϕ] Cos[ψ]     Si
                      ⎣    ⎜  Cos[ϕ] Sin[θ] Sin[ψ]                             C
                           ⎜    -Cos[ψ] Sin[θ]                 Sin[ψ]
                           ⎜ ─────────────────────────────────────────────────────
                           ⎝          0                          0
```

Tetrahedron

In this section we will implement a couple of functions in *Mathematica*, that makes it possible to rotate the tetrahedron, which is one of the 5 Platonic solids.

Function Definition - Tetrahedron

Below functions related to creating, rotating, and visualizing the tetrahedron are implemented in *Mathematica*.

■ Define *createTetrahedron*

We define a function in *Mathematica* which creates a tetahedron based on the input parameters: *edgeLength* and *center*, where *edgeLength* specifies the length between the 4 vertices that compose the tetrahedron and *center* specifies the center point of the tetrahedron. The function *createTetrahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the tetrahedron.

In[68]:=
```
createTetrahedron[edgeLength_?NumericQ,
  center : {xCenter_?NumericQ, yCenter_?NumericQ, zCenter_?NumericQ}] :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {points, p0, p1, p2, p3},

  (* DEFINE POINTS *)
  p0 = {xCenter, yCenter, zCenter + ((1 / 4) * Sqrt[6] * edgeLength)};
  p1 = {xCenter, yCenter + ((1 / 3) * Sqrt[3] * edgeLength),
    zCenter - ((1 / 12) * Sqrt[6] * edgeLength)};
  p2 = {xCenter - 0.5 * edgeLength,
    yCenter - ((1 / 6) * Sqrt[3] * edgeLength),
    zCenter - ((1 / 12) * Sqrt[6] * edgeLength)};
  p3 = {xCenter + 0.5 * edgeLength,
    yCenter - ((1 / 6) * Sqrt[3] * edgeLength),
    zCenter - ((1 / 12) * Sqrt[6] * edgeLength)};
  points = {p0, p1, p2, p3}
]
```

■ Define *rotateTetrahedron*

We define a function in *Mathematica* which rotates tetrahedron about the XYZ-axes given the input parameters: *points*, ϕ, θ, and ψ. The *points* parameter is a list containing the 4 points representing the 3D coordinates of the vertices in the Tetrahedron. The ϕ parameter specifies the rotation angle, in degrees, about the X-axis. The θ parameter specifies the rotation angle, in degrees, about the Y-axis. The ψ parameter specifies the rotation angle, in degrees, about the Z-axis. The function *rotateTetrahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the tetrahedron after applying the rotation.

In[69]:=
```
rotateTetrahedron[points_,
   angle : {phi_?NumericQ, theta_?NumericQ, psi_?NumericQ}] /;
  (Length[points] == 4) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {rmat, p0, p1, p2, p3, pc, p0new, p1new, p2new, p3new, newpts},

  (* DEFINE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  pc = (p0 + p1 + p2 + p3) / 4;

  (* CREATE ROTATION MATRIX *)
  rmat = RotationMatrix[psi Degree, {0, 0, 1}].
    RotationMatrix[theta Degree, {0, 1, 0}].
    RotationMatrix[phi Degree, {1, 0, 0}];

  (* CALCULATE NEW POINTS *)
  p0new = pc + rmat.(p0 - pc);
  p1new = pc + rmat.(p1 - pc);
  p2new = pc + rmat.(p2 - pc);
  p3new = pc + rmat.(p3 - pc);
  newpts = {p0new, p1new, p2new, p3new}
 ]
```

■ Define *drawTetrahedron*

We define a function in *Mathematica* which is able to display a tetrahedron given the input parameter: *points*. The *points* parameter is a list containing the 4 points representing the 3D coordinates of the vertices in the tetrahedron. The function *drawTetrahedron* draws the vertices and edges which compose the tetrahedron and displays these in a single graphical object.

In[70]:=
```
drawTetrahedron[points_] /; (Length[points] == 4) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {p0, p1, p2, p3, pc,
   v0, v1, v2, v3, vc,
   e01, e02, e03, e12, e13, e23},

  (* CREATE POINTS *)
  p0 = points⟦1⟧;
  p1 = points⟦2⟧;
  p2 = points⟦3⟧;
  p3 = points⟦4⟧;
  pc = (p0 + p1 + p2 + p3) / 4;

  (* CREATE GRAPHICS FOR VERTICES *)
  v0 = Graphics3D[{RGBColor[1, 1, 0], PointSize[0.1], Point[p0]}];
  v1 = Graphics3D[{RGBColor[1, 0, 0], PointSize[0.1], Point[p1]}];
  v2 = Graphics3D[{RGBColor[0, 1, 0], PointSize[0.1], Point[p2]}];
  v3 = Graphics3D[{RGBColor[0, 0, 1], PointSize[0.1], Point[p3]}];
  vc = Graphics3D[{RGBColor[0, 0, 0], PointSize[0.03], Point[pc]}];

  (* CREATE GRAPHICS FOR EDGES *)
  e01 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
     Line[{p0, p1}]}];
  e02 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p2}]}];
  e03 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p3}]}];
  e12 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p2}]}];
  e13 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p3}]}];
  e23 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p3}]}];

  Show[{v0, v1, v2, v3, vc, e01, e02, e03, e12, e13, e23},
   AxesLabel → {"x", "y", "z"}, AspectRatio → 1, Axes → True,
   PlotRange → All, ImageSize → Automatic, Boxed → True]
 ]
```

Function Test - Tetrahedron

Below we test the functions related to creating, rotating, and visualizing the tetrahedron.

■ Test *createTetrahedron*

We create a tetrahedron with edgelength=5 centered at (5, 5, 5).

In[71]:=

```
tetrahedronPoints = createTetrahedron[5, {5, 5, 5}]
```

Out[71]=

$$\left\{\left\{5,\ 5,\ 5+\frac{5\sqrt{\frac{3}{2}}}{2}\right\},\ \left\{5,\ 5+\frac{5}{\sqrt{3}},\ 5-\frac{5}{2\sqrt{6}}\right\},\ \left\{2.5,\ 5-\frac{5}{2\sqrt{3}},\ 5-\frac{5}{2\sqrt{6}}\right\},\ \left\{7.5,\ 5-\frac{5}{2\sqrt{3}},\ 5-\frac{5}{2\sqrt{6}}\right\}\right\}$$

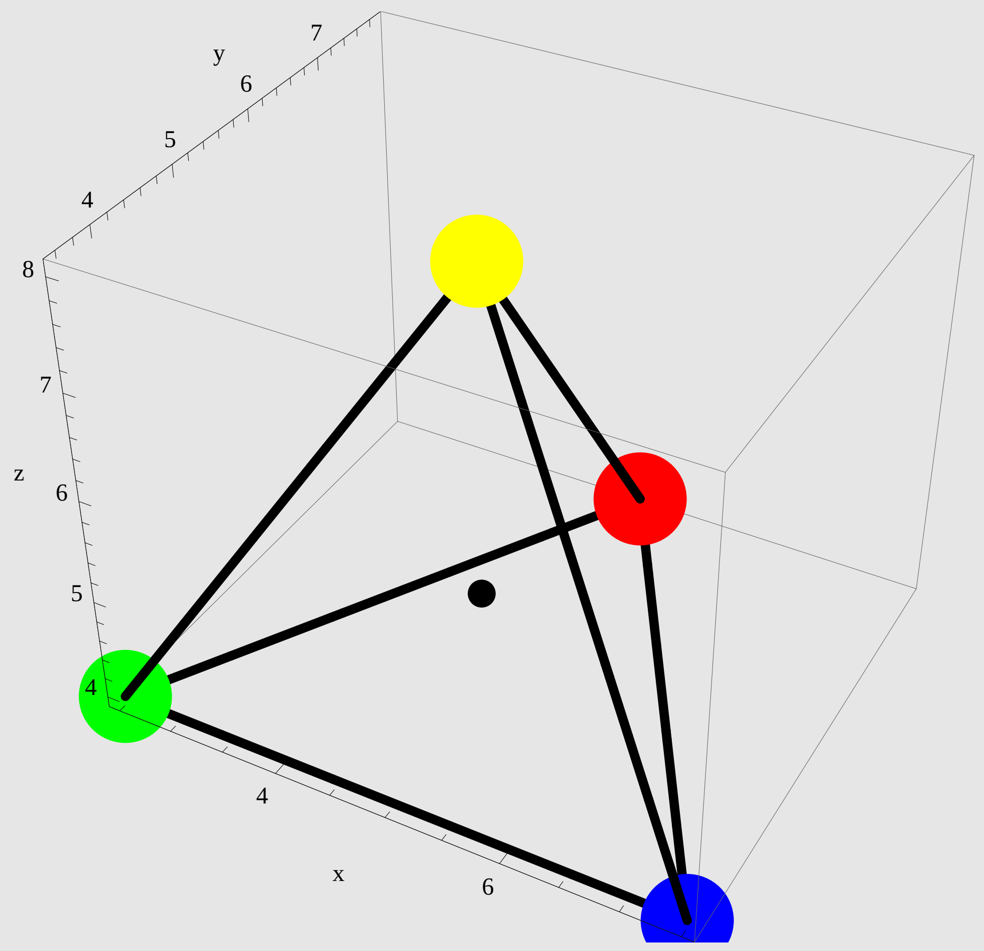
■ Test *drawTetrahedron*

Below we visualize the tetrahedron created above.

In[72]:=

```
drawTetrahedron[tetrahedronPoints]
```

Out[72]=



■ Test *rotateTetrahedron*

Rotating the tetrahedron an angle ϕ counterclockwise about the X-axis (Roll)

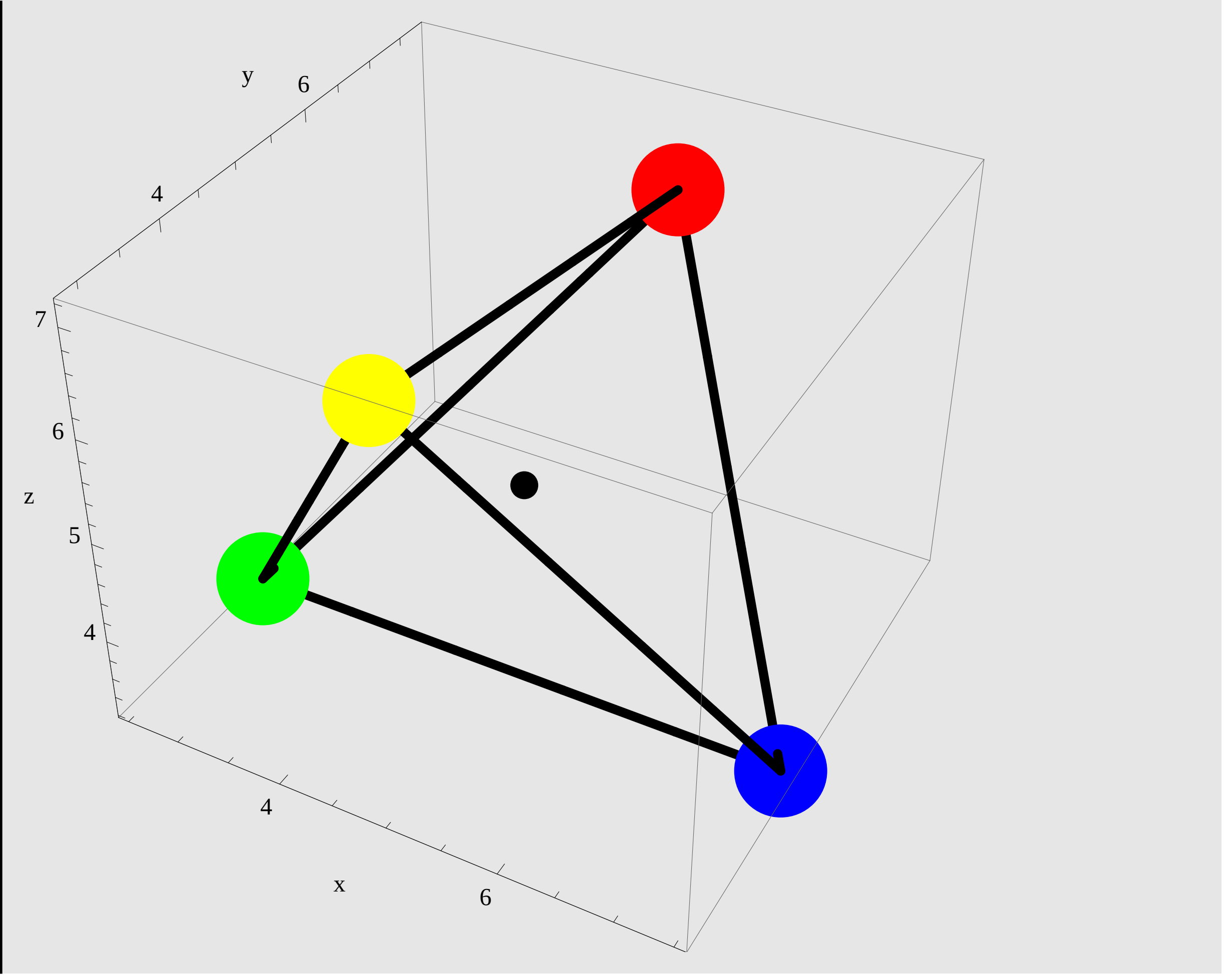
We rotate the tetrahedron an angle $\phi = 45^{\circ}$ counterclockwise about the X-axis.

In[73]:=
```
tetrahedronPointsRotatedAboutX =
  rotateTetrahedron[tetrahedronPoints, {45, 0, 0}];
```

The tetrahedron rotated an angle $\phi = 45^{\circ}$ conterclockwise about the X-axis is displayed below.

In[74]:=
```
drawTetrahedron[tetrahedronPointsRotatedAboutX]
```

Out[74]=



Rotating the tetrahedron an angle θ counterclockwise about the Y-axis (Pitch)

We rotate the tetrahedron an angle $\theta = 45^{\circ}$ counterclockwise about the Y-axis.

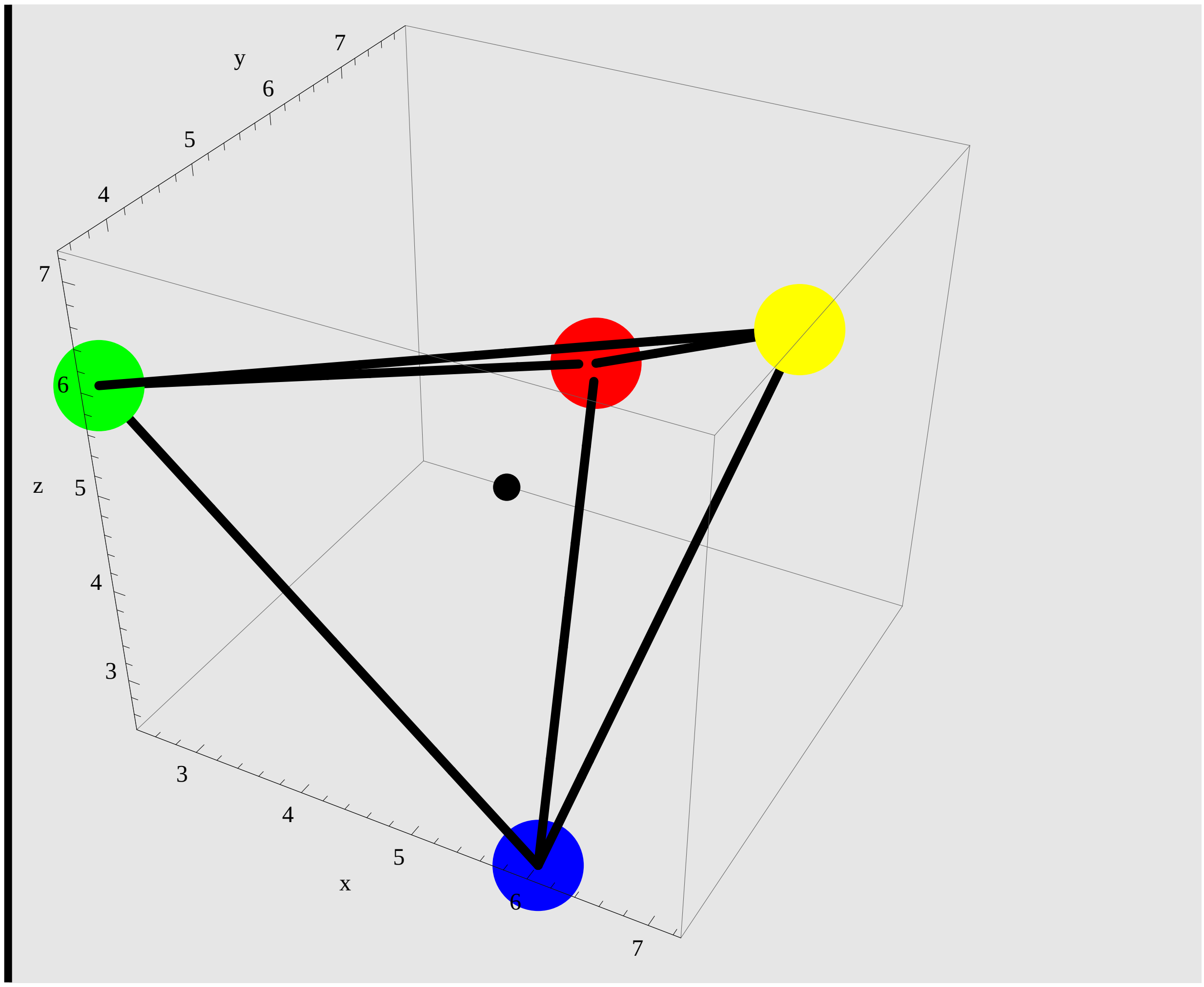
In[75]:=
```
tetrahedronPointsRotatedAboutY =
  rotateTetrahedron[tetrahedronPoints, {0, 45, 0}];
```

The tetrahedron rotated an angle $\theta = 45^{\circ}$ conterclockwise about the Y-axis is displayed below.

In[76]:=

```
drawTetrahedron[tetrahedronPointsRotatedAboutY]
```

Out[76]=



Rotating the tetrahedron an angle ψ counterclockwise about the Z-axis (Yaw)

We rotate the tetrahedron an angle $\psi = 45^\circ$ counterclockwise about the Z-axis.

In[77]:=

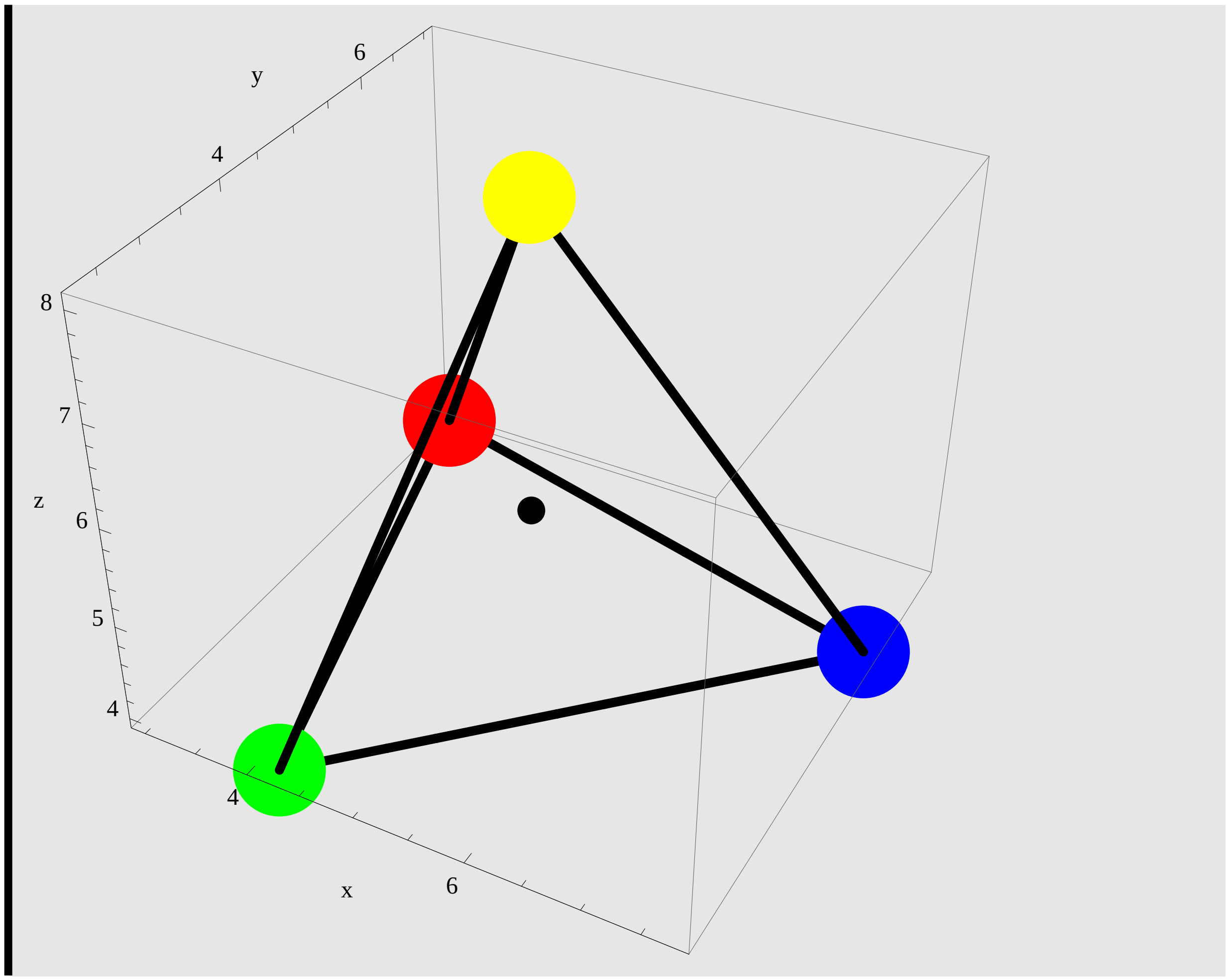
```
tetrahedronPointsRotatedAboutZ =
  rotateTetrahedron[tetrahedronPoints, {0, 0, 45}];
```

The tetrahedron rotated an angle $\psi = 45^\circ$ conterclockwise about the Z-axis is displayed below.

In[78]:=

```
drawTetrahedron[tetrahedronPointsRotatedAboutZ]
```

Out[78]=



■ Rotating in steps

Step 1: rotating the tetrahedron an angle ϕ counterclockwise about the X-axis (Roll)

First we rotate the tetrahedron about the X-axis and stores the result.

In[79]:=

```
rx = rotateTetrahedron[tetrahedronPoints, {45, 0, 0}];
```

Step 2: rotating the tetrahedron from step 1 an angle θ counterclockwise about the Y-axis (Pitch)

We use the result from step 1 and rotate the tetrahedron a second time - this time about the Y-axis

In[80]:=

```
ry = rotateTetrahedron[rx, {0, 45, 0}];
```

Step 3: rotating the tetrahedron from step 2 an angle ψ counterclockwise about the Z-axis (Yaw)

We use the result from step 2 and rotate the tetrahedron a third time - this time about the Z-axis

In[81]:=

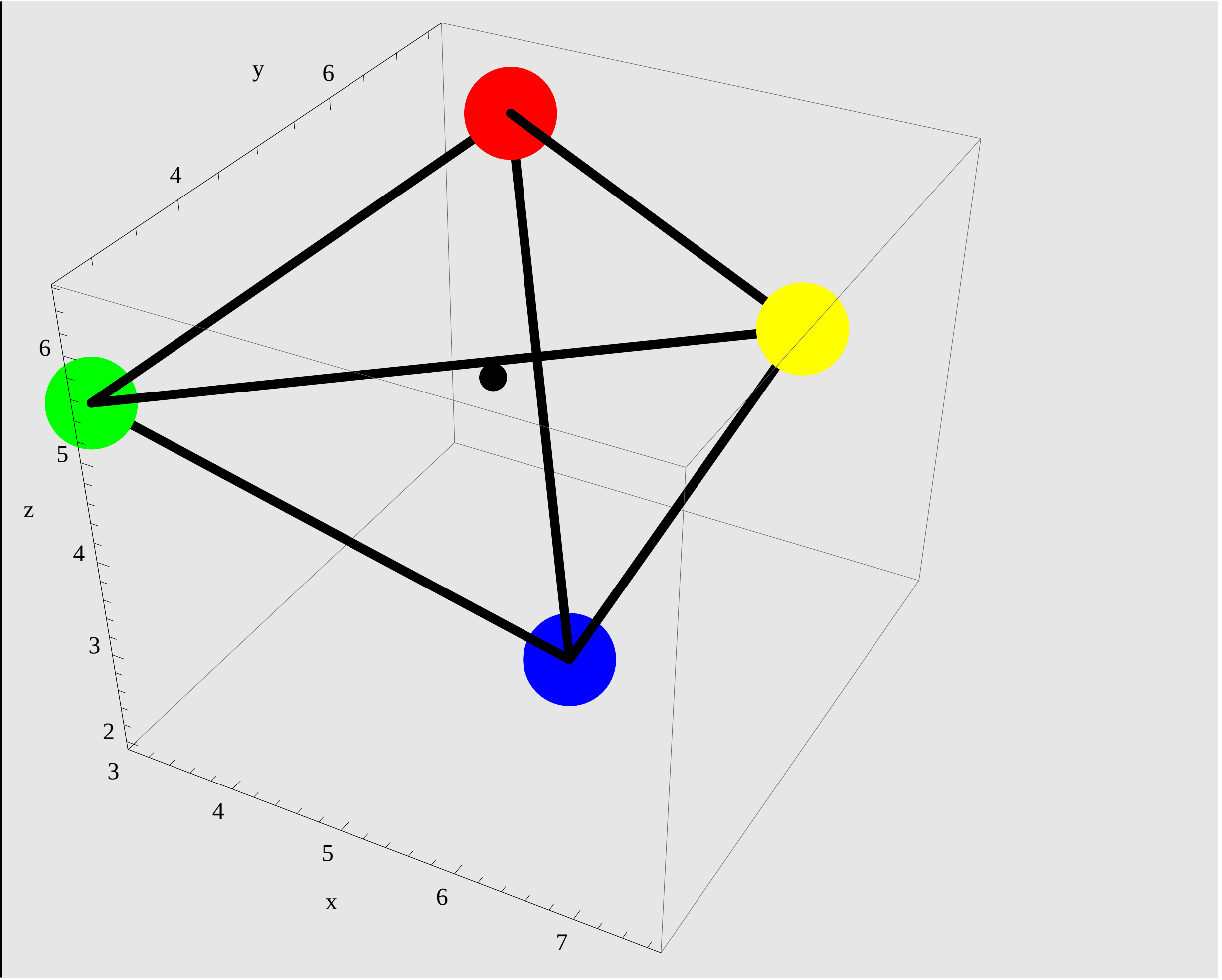
```
rz = rotateTetrahedron[ry, {0, 0, 45}];
```

The result of the three consecutive rotations is shown below

In[82]:=

```
drawTetrahedron[rz]
```

Out[82]=

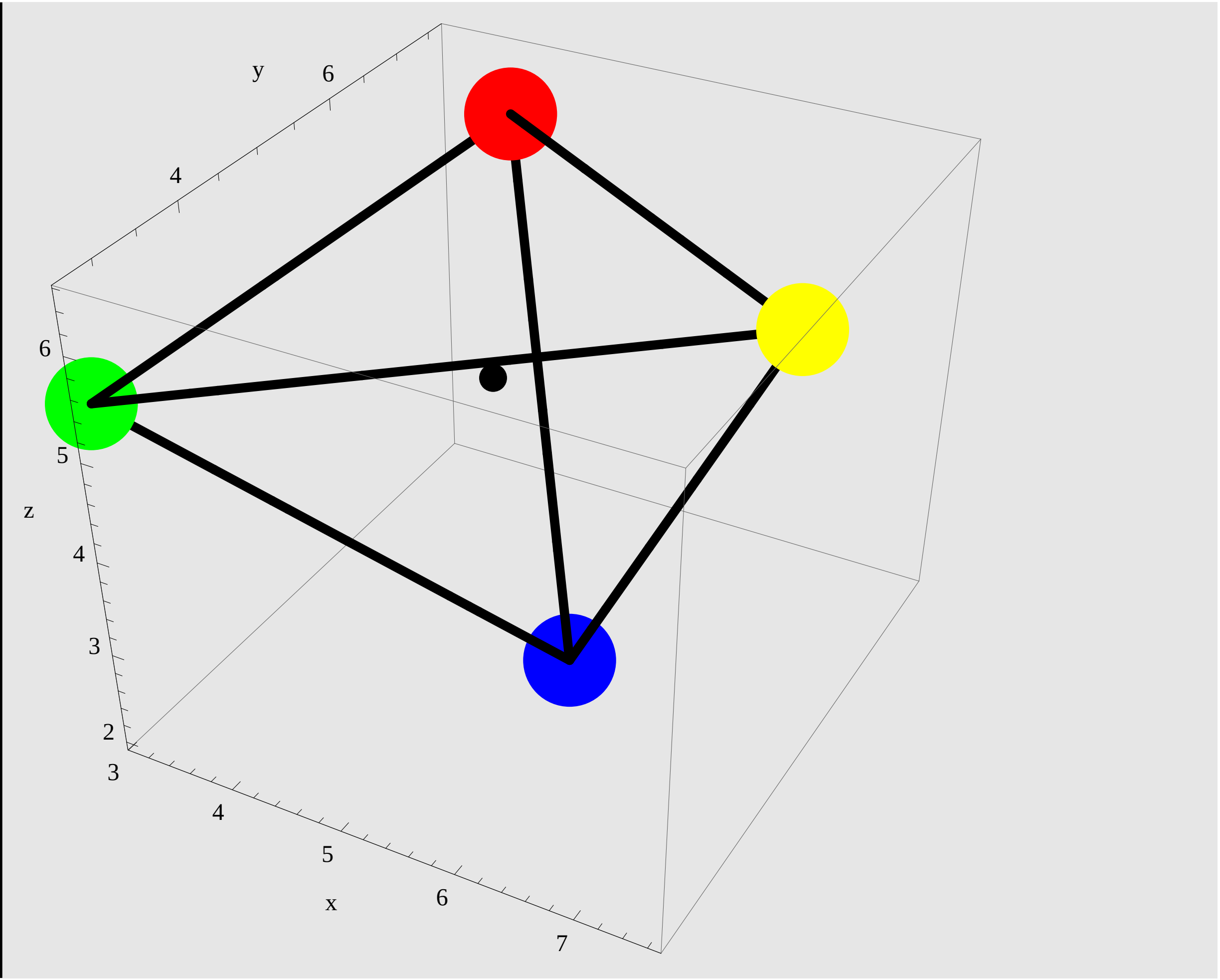


Rotating the tetrahedron an angle ϕ counterclockwise about the X-axis, then an angle θ counterclockwise about the Y-axis, and finally an angle ψ counterclockwise about the Z-axis

Instead of applying the rotations separately we can calculate the resulting rotation directly. This is shown below

In[83]:=
```
drawTetrahedron[rotateTetrahedron[tetrahedronPoints, {45, 45, 45}]]
```

Out[83]=



Octahedron

In this section we will implement a couple of functions in *Mathematica*, that makes it possible to rotate the octahedron, which is one of the 5 Platonic solids.

Function Definition - Octahedron

Below functions related to creating, rotating, and visualizing the octahedron are implemented in *Mathematica*.

■ Define *createOctahedron*

We define a function in *Mathematica* which creates a octahedron based on the input parameters: *edgeLength* and *center*, where *edgeLength* specifies the length between the 4 vertices that compose the octahedron and *center* specifies the center point of the octahedron. The function *createOctahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the octahedron.

In[84]:=

```
createOctahedron[edgeLength_?NumericQ,
  center : {xCenter_?NumericQ, yCenter_?NumericQ, zCenter_?NumericQ}] :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {points, p0, p1, p2, p3, p4, p5},

  (* DEFINE POINTS *)
  p0 = {xCenter, yCenter, zCenter + ((1 / 2) * Sqrt[2] * edgeLength)};
  p1 = {xCenter + ((1 / 2) * edgeLength), yCenter + ((1 / 2) * edgeLength),
    zCenter};
  p2 = {xCenter - ((1 / 2) * edgeLength), yCenter - ((1 / 2) * edgeLength),
    zCenter};
  p3 = {xCenter + ((1 / 2) * edgeLength), yCenter - ((1 / 2) * edgeLength),
    zCenter};
  p4 = {xCenter - ((1 / 2) * edgeLength), yCenter + ((1 / 2) * edgeLength),
    zCenter};
  p5 = {xCenter, yCenter, zCenter - ((1 / 2) * Sqrt[2] * edgeLength)};
  points = {p0, p1, p2, p3, p4, p5}
]
```

■ Define *rotateOctahedron*

We define a function in *Mathematica* which rotates octahedron about the XYZ-axes given the input parameters: *points*, ϕ, θ, and ψ. The *points* parameter is a list containing the 6 points representing the 3D coordinates of the vertices in the octahedron. The ϕ parameter specifies the rotation angle, in degrees, about the X-axis. The θ parameter specifies the rotation angle, in degrees, about the Y-axis. The ψ parameter specifies the rotation angle, in degrees, about the Z-axis. The function *rotateOctahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the octahedron after applying the rotation.

In[85]:=

```
rotateOctahedron[points_,
   angle : {phi_?NumericQ, theta_?NumericQ, psi_?NumericQ}] /;
  (Length[points] == 6) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {rmat, p0, p1, p2, p3, p4, p5, pc,
   p0new, p1new, p2new, p3new, p4new, p5new, newpts},

  (* DEFINE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  p4 = points[[5]];
  p5 = points[[6]];
  pc = (p0 + p1 + p2 + p3 + p4 + p5) / 6;

  (* CREATE ROTATION MATRIX *)
  rmat = RotationMatrix[psi Degree, {0, 0, 1}].
    RotationMatrix[theta Degree, {0, 1, 0}].
    RotationMatrix[phi Degree, {1, 0, 0}];

  (* CALCULATE NEW POINTS *)
  p0new = pc + rmat.(p0 - pc);
  p1new = pc + rmat.(p1 - pc);
  p2new = pc + rmat.(p2 - pc);
  p3new = pc + rmat.(p3 - pc);
  p4new = pc + rmat.(p4 - pc);
  p5new = pc + rmat.(p5 - pc);
  newpts = {p0new, p1new, p2new, p3new, p4new, p5new}
 ]
```

■ Define *drawOctahedron*

We define a function in *Mathematica* which is able to display a octahedron given the input parameter: *points*. The *points* parameter is a list containing the 6 points representing the 3D coordinates of the vertices in the octahedron. The function *drawOctahedron* draws the vertices and edges which compose the octahedron and displays these in a single graphical object.

In[86]:=

```
drawOctahedron[points_] /; (Length[points] == 6) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {p0, p1, p2, p3, p4, p5, pc,
   v0, v1, v2, v3, v4, v5, vc,
   e01, e02, e03, e12, e13, e23},

  (* CREATE POINTS *)
  p0 = points⟦1⟧;
  p1 = points⟦2⟧;
  p2 = points⟦3⟧;
  p3 = points⟦4⟧;
  p4 = points⟦5⟧;
  p5 = points⟦6⟧;
  pc = (p0 + p1 + p2 + p3 + p4 + p5) / 6;

  (* CREATE GRAPHICS FOR VERTICES *)
  v0 = Graphics3D[{RGBColor[1, 1, 0], PointSize[0.1], Point[p0]}];
  v1 = Graphics3D[{RGBColor[1, 0, 0], PointSize[0.1], Point[p1]}];
  v2 = Graphics3D[{RGBColor[0, 1, 0], PointSize[0.1], Point[p2]}];
  v3 = Graphics3D[{RGBColor[0, 0, 1], PointSize[0.1], Point[p3]}];
  v4 = Graphics3D[{RGBColor[0, 1, 1], PointSize[0.1], Point[p4]}];
  v5 = Graphics3D[{RGBColor[1, 0, 1], PointSize[0.1], Point[p5]}];
  vc = Graphics3D[{RGBColor[0, 0, 0], PointSize[0.03], Point[pc]}];

  (* CREATE GRAPHICS FOR EDGES *)
  e01 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
     Line[{p0, p1}]}];
  e02 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p2}]}];
  e03 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p3}]}];
  e04 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p4}]}];
  e13 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p3}]}];
  e14 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p4}]}];
  e23 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p3}]}];
  e24 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p4}]}];
  e15 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p5}]}];
  e25 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p5}]}];
  e35 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p5}]}];
  e45 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p5}]}];

  Show[{v0, v1, v2, v3, v4, v5, vc, e01, e02, e03, e04, e13, e14,
    e23, e24, e15, e25, e35, e45}, AxesLabel → {"x", "y", "z"},
   AspectRatio → 1, Axes → True, PlotRange → All, ImageSize → Automatic,
   Boxed → True]
 ]
```

Function Test - Octahedron

Below we test the functions related to creating, rotating, and visualizing the octahedron.

■ Test *createOctahedron*

We create a octahedron with edgelength=5 centered at (5, 5, 5).

In[87]:=
```
octahedronPoints = createOctahedron[5, {5, 5, 5}]
```

Out[87]=

$$\left\{\left\{5, 5, 5+\frac{5}{\sqrt{2}}\right\}, \left\{\frac{15}{2}, \frac{15}{2}, 5\right\}, \left\{\frac{5}{2}, \frac{5}{2}, 5\right\}, \left\{\frac{15}{2}, \frac{5}{2}, 5\right\}, \left\{\frac{5}{2}, \frac{15}{2}, 5\right\}, \left\{5, 5, 5-\frac{5}{\sqrt{2}}\right\}\right\}$$

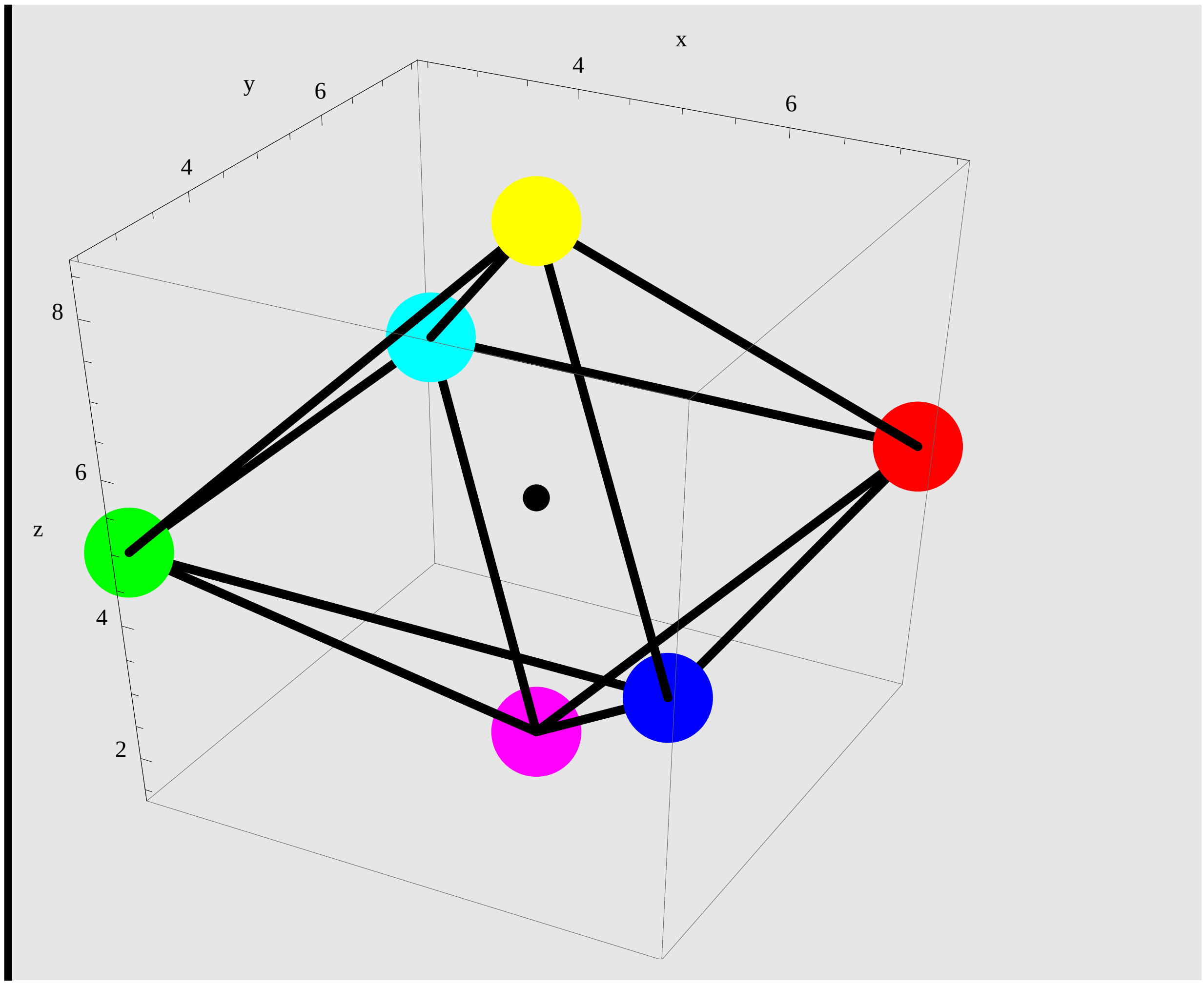
■ Test *drawOctahedron*

Below we visualize the octahedron created above.

In[88]:=

```
drawOctahedron[octahedronPoints]
```

Out[88]=



■ Test *rotateOctahedron*

Rotating the Octahedron an angle ϕ counterclockwise about the X-axis (Roll)

We rotate the octahedron an angle $\phi = 45^\circ$ counterclockwise about the X-axis.

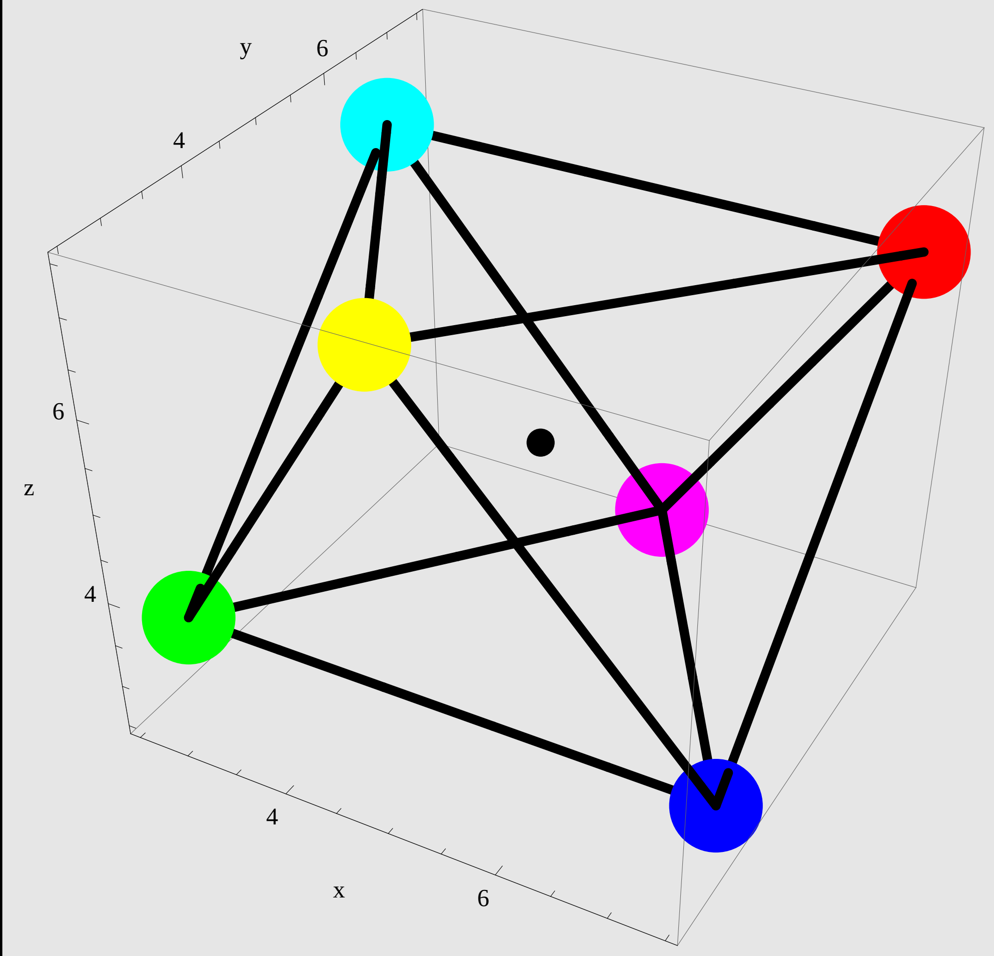
In[89]:=

```
octahedronPointsRotatedAboutX =
  rotateOctahedron[octahedronPoints, {45, 0, 0}];
```

The octahedron rotated an angle $\phi = 45^\circ$ conterclockwise about the X-axis is displayed below.

In[90]:=
```
drawOctahedron[octahedronPointsRotatedAboutX]
```

Out[90]=



Rotating the octahedron an angle θ counterclockwise about the Y-axis (Pitch)

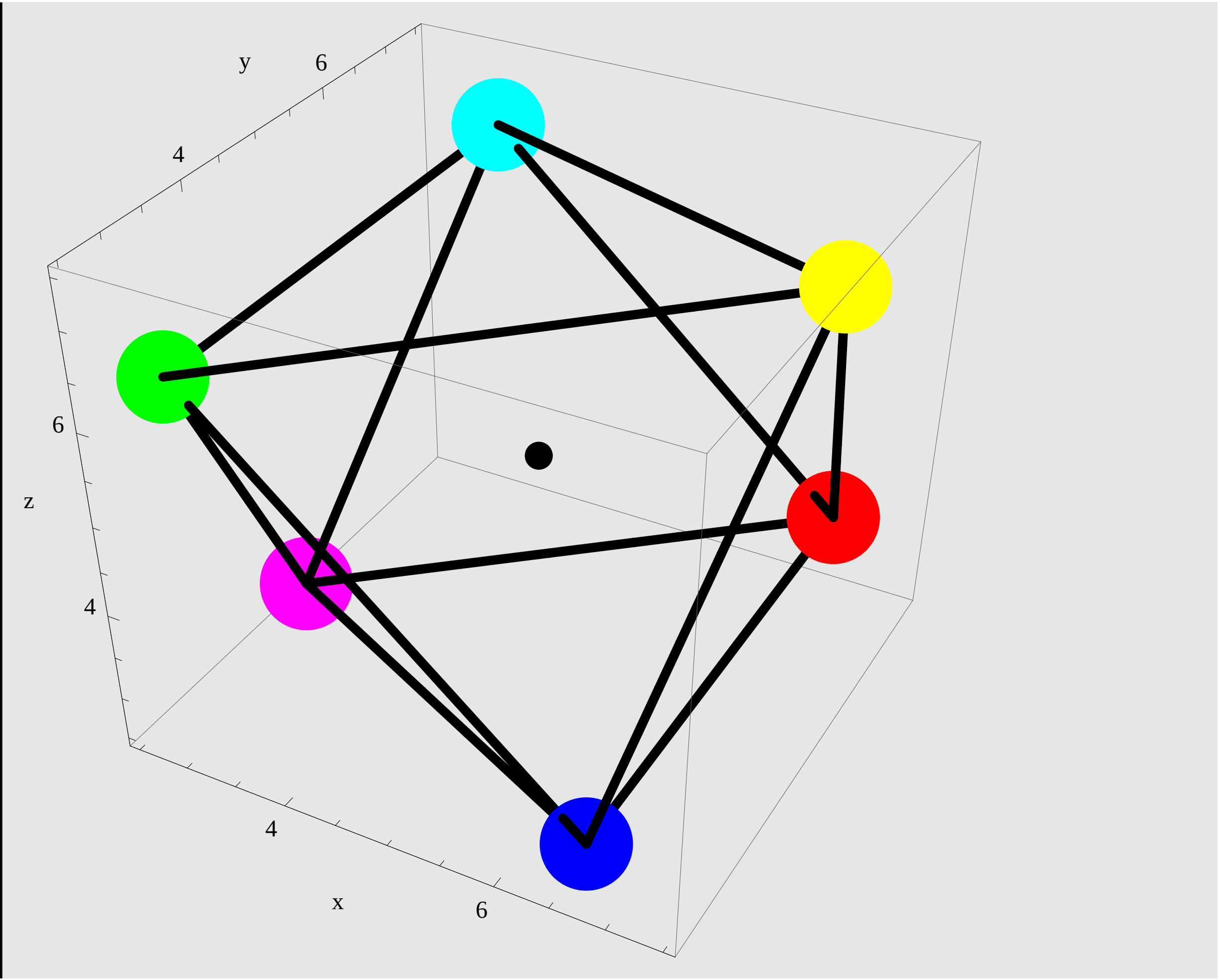
We rotate the octahedron an angle $\theta = 45^\circ$ counterclockwise about the Y-axis.

In[91]:=
```
octahedronPointsRotatedAboutY =
  rotateOctahedron[octahedronPoints, {0, 45, 0}];
```

The octahedron rotated an angle $\theta = 45^\circ$ conterclockwise about the Y-axis is displayed below.

In[92]:=
```
drawOctahedron[octahedronPointsRotatedAboutY]
```

Out[92]=



Rotating the octahedron an angle ψ counterclockwise about the Z-axis (Yaw)

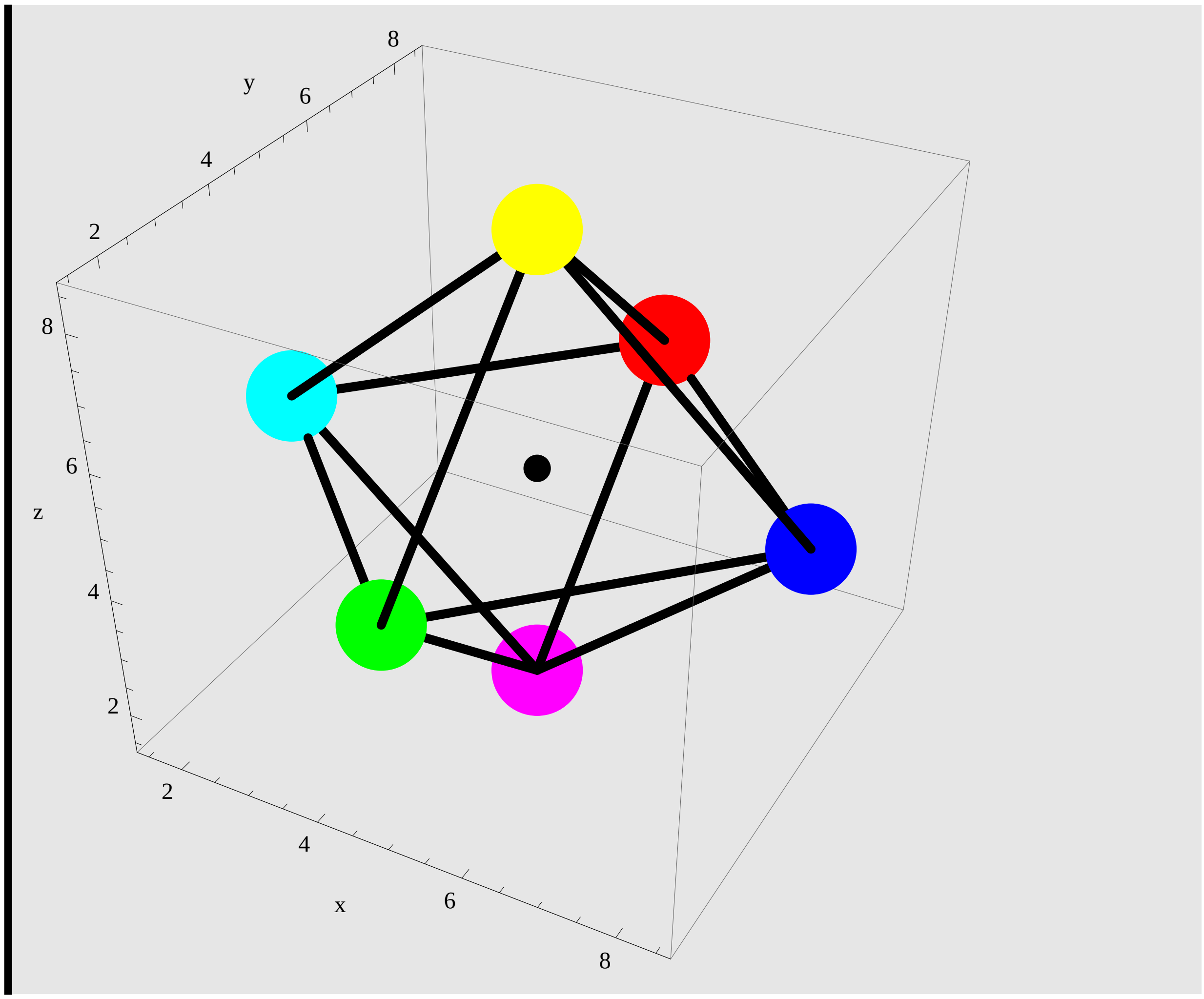
We rotate the octahedron an angle $\psi = 45^\circ$ counterclockwise about the Z-axis.

In[93]:=
```
octahedronPointsRotatedAboutZ =
  rotateOctahedron[octahedronPoints, {0, 0, 45}];
```

The octahedron rotated an angle $\psi = 45^\circ$ conterclockwise about the Z-axis is displayed below.

In[94]:=
```
drawOctahedron[octahedronPointsRotatedAboutZ]
```

Out[94]=



■ Rotating in steps

Step 1: rotating the octahedron an angle ϕ counterclockwise about the X-axis (Roll)

First we rotate the octahedron about the X-axis and stores the result.

In[95]:=
```
rx = rotateOctahedron[octahedronPoints, {45, 0, 0}];
```

Step 2: rotating the octahedron from step 1 an angle θ counterclockwise about the Y-axis (Pitch)

We use the result from step 1 and rotate the octahedron a second time - this time about the Y-axis

In[96]:=
```
ry = rotateOctahedron[rx, {0, 45, 0}];
```

Step 3: rotating the octahedron from step 2 an angle ψ counterclockwise about the Z-axis (Yaw)

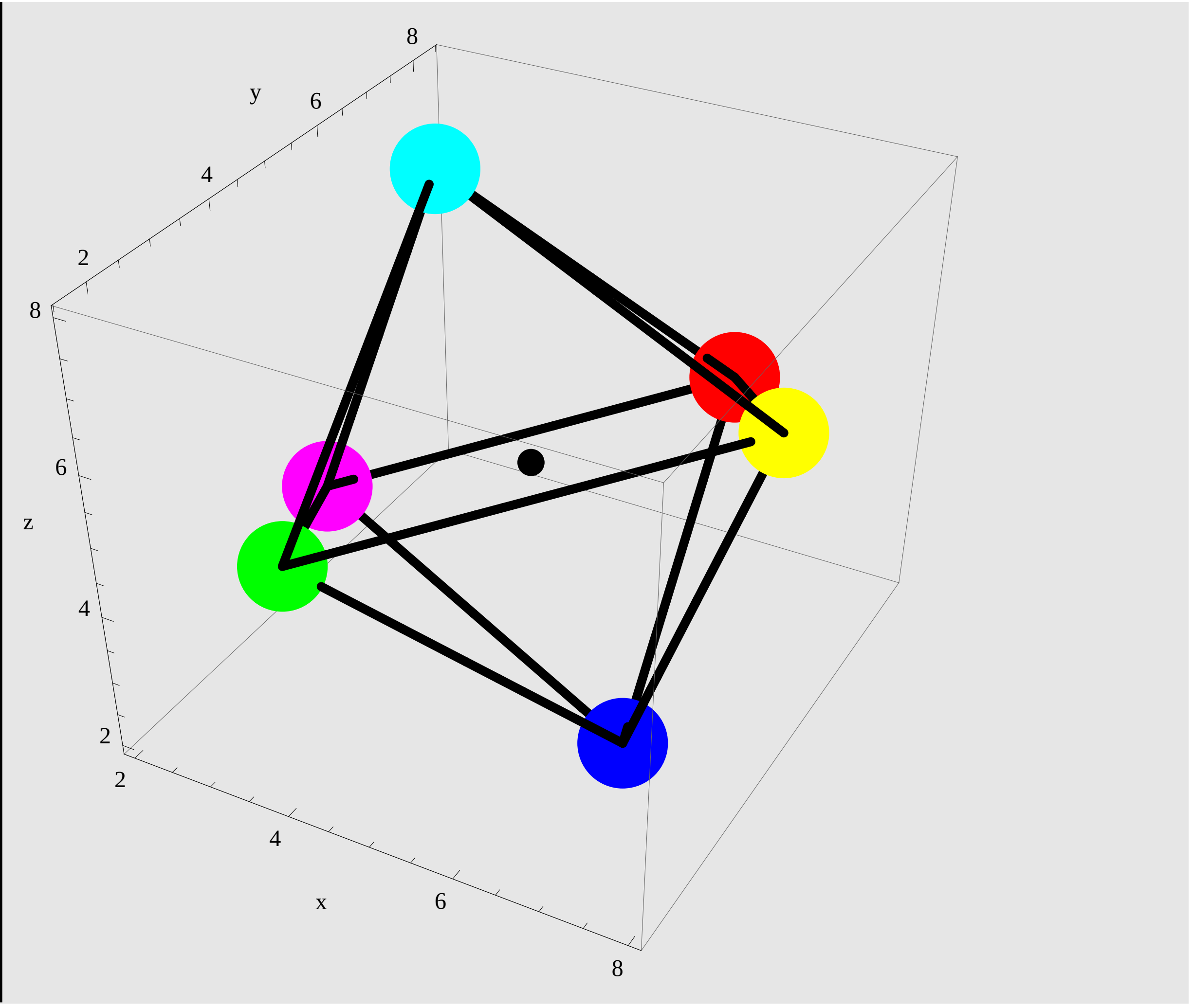
We use the result from step 2 and rotate the octahedron a third time - this time about the Z-axis

In[97]:=
```
rz = rotateOctahedron[ry, {0, 0, 45}];
```

The result of the three consecutive rotations is shown below

In[98]:=
```
drawOctahedron[rz]
```

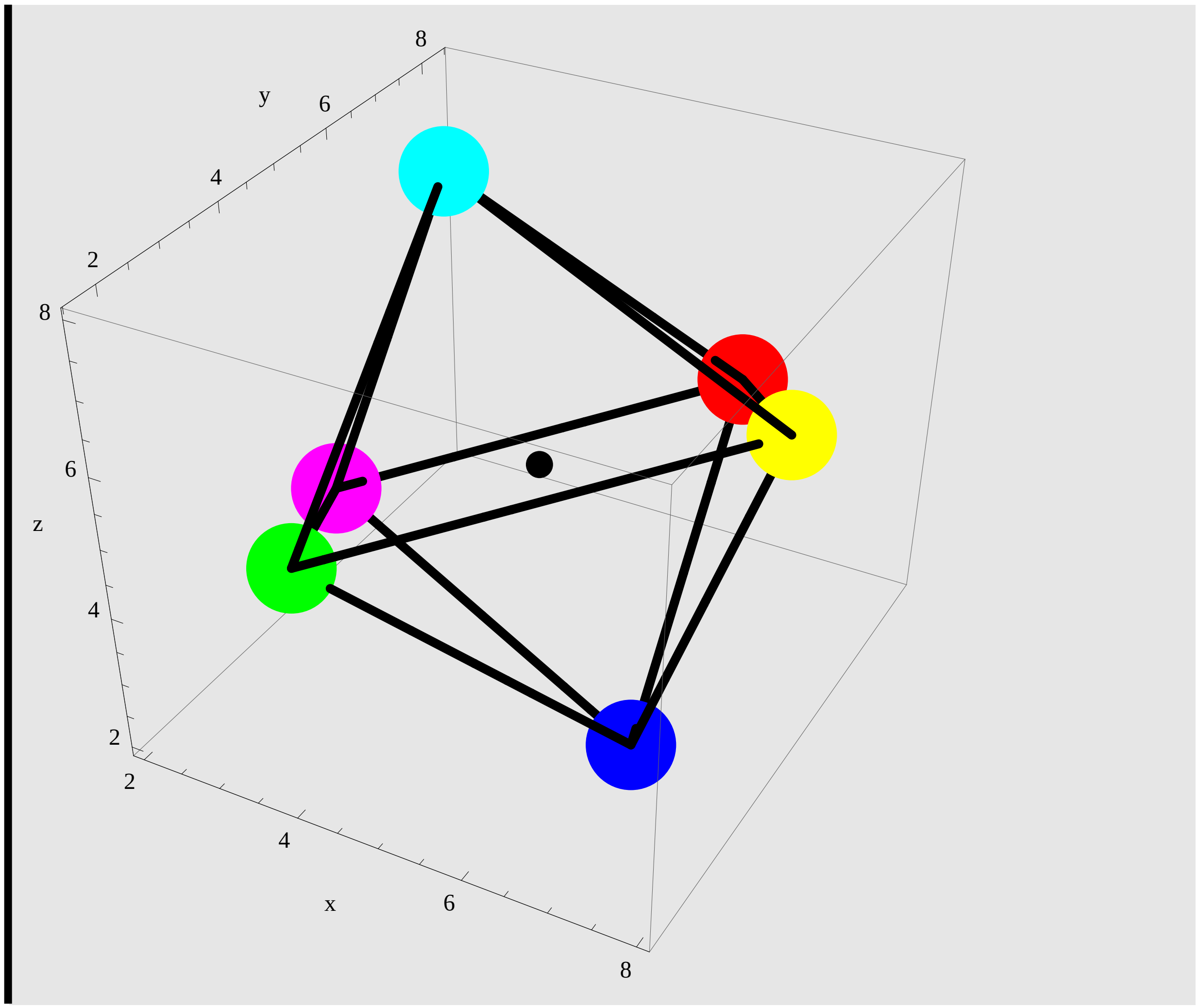
Out[98]=



Rotating the octahedron an angle ϕ counterclockwise about the X-axis, then an angle θ counterclockwise about the Y-axis, and finally an angle ψ counterclockwise about the Z-axis

Instead of applying the rotations separately we can calculate the resulting rotation directly. This is shown below

In[99]:=
```
drawOctahedron[rotateOctahedron[octahedronPoints, {45, 45, 45}]]
```

Out[99]=


Hexahedron

In this section we will implement a couple of functions in *Mathematica*, that makes it possible to rotate the Hexahedron, which is one of the 5 Platonic solids.

Function Definition - Hexahedron

Below functions related to creating, rotating, and visualizing the hexahedron are implemented in *Mathematica*.

■ Define *createHexahedron*

We define a function in *Mathematica* which creates a hexahedron (cube) based on the input parameters: *edgeLength* and *center*, where *edgeLength* specifies the length between the 8 vertices that compose the hexahedron and *center* specifies the center point of the hexahedron. The function *createHexahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the hexahedron.

In[100]:=

```
createHexahedron[edgeLength_?NumericQ,
  center : {xCenter_?NumericQ, yCenter_?NumericQ, zCenter_?NumericQ}] :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {points, p0, p1, p2, p3, p4, p5, p6, p7},

  (* DEFINE POINTS *)
  p0 = {xCenter + 0.5 * edgeLength, yCenter - 0.5 * edgeLength,
    zCenter + 0.5 * edgeLength};
  p1 = {xCenter - 0.5 * edgeLength, yCenter - 0.5 * edgeLength,
    zCenter + 0.5 * edgeLength};
  p2 = {xCenter - 0.5 * edgeLength, yCenter + 0.5 * edgeLength,
    zCenter + 0.5 * edgeLength};
  p3 = {xCenter + 0.5 * edgeLength, yCenter + 0.5 * edgeLength,
    zCenter + 0.5 * edgeLength};
  p4 = {xCenter - 0.5 * edgeLength, yCenter + 0.5 * edgeLength,
    zCenter - 0.5 * edgeLength};
  p5 = {xCenter + 0.5 * edgeLength, yCenter + 0.5 * edgeLength,
    zCenter - 0.5 * edgeLength};
  p6 = {xCenter + 0.5 * edgeLength, yCenter - 0.5 * edgeLength,
    zCenter - 0.5 * edgeLength};
  p7 = {xCenter - 0.5 * edgeLength, yCenter - 0.5 * edgeLength,
    zCenter - 0.5 * edgeLength};
  points = {p0, p1, p2, p3, p4, p5, p6, p7}
]
```

■ Define *rotateHexahedron*

We define a function in *Mathematica* which rotates a hexahedron about the XYZ-axes given the input parameters: *points*, ϕ, θ, and ψ. The *points* parameter is a list containing the 8 points representing the 3D coordinates of the vertices in the hexahedron. The ϕ parameter specifies the rotation angle, in degrees, about the X-axis. The θ parameter specifies the rotation angle, in degrees, about the Y-axis. The ψ parameter specifies the rotation angle, in degrees, about the Z-axis. The function *rotateHexahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the hexahedron after applying the rotation.

In[101]:=

```
rotateHexahedron[points_,
   angle : {phi_?NumericQ, theta_?NumericQ, psi_?NumericQ}] /;
  (Length[points] == 8) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {rmat, p0, p1, p2, p3, p4, p5, p6, p7, p8, pc,
   p0new, p1new, p2new, p3new, p4new, p5new, p6new, p7new, newpts},

  (* DEFINE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  p4 = points[[5]];
  p5 = points[[6]];
  p6 = points[[7]];
  p7 = points[[8]];
  pc = (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7) / 8;

  (* CREATE ROTATION MATRIX *)
  rmat = RotationMatrix[psi Degree, {0, 0, 1}].
    RotationMatrix[theta Degree, {0, 1, 0}].
    RotationMatrix[phi Degree, {1, 0, 0}];

  (* CALCULATE NEW POINTS *)
  p0new = pc + rmat.(p0 - pc);
  p1new = pc + rmat.(p1 - pc);
  p2new = pc + rmat.(p2 - pc);
  p3new = pc + rmat.(p3 - pc);
  p4new = pc + rmat.(p4 - pc);
  p5new = pc + rmat.(p5 - pc);
  p6new = pc + rmat.(p6 - pc);
  p7new = pc + rmat.(p7 - pc);
  newpts = {p0new, p1new, p2new, p3new, p4new, p5new, p6new, p7new}
 ]
```

■ Define *drawHexahedron*

We define a function in *Mathematica* which is able to display a hexahedron given the input parameter: *points*. The *points* parameter is a list containing the 8 points representing the 3D coordinates of the vertices in the hexahedron. The function *drawHexahedron* draws the vertices and edges which compose the hexahedron and displays these in a single graphical object.

In[102]:=

```
drawHexahedron[points_] /; (Length[points] == 8) :=
 Module[
(* ALLOCATE LOCAL VARIABLES *)
{ p0, p1, p2, p3, p4, p5, p6, p7, pc,
 v0, v1, v2, v3, v4, v5, v6, v7, vc,
 e01, e03, e06, e12, e17, e23,
 e24, e35, e45, e47, e56, e67},

(* CREATE POINTS *)
p0 = points[[1]];
p1 = points[[2]];
p2 = points[[3]];
p3 = points[[4]];
p4 = points[[5]];
p5 = points[[6]];
p6 = points[[7]];
p7 = points[[8]];
pc = (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7) / 8;

(* CREATE GRAPHICS FOR VERTICES *)
v0 = Graphics3D[{RGBColor[0, 0, 0], PointSize[0.1], Point[p0]}];
v1 = Graphics3D[{RGBColor[1, 0, 0], PointSize[0.1], Point[p1]}];
v2 = Graphics3D[{RGBColor[0, 1, 0], PointSize[0.1], Point[p2]}];
v3 = Graphics3D[{RGBColor[0, 0, 1], PointSize[0.1], Point[p3]}];
v4 = Graphics3D[{RGBColor[0, 1, 1], PointSize[0.1], Point[p4]}];
v5 = Graphics3D[{RGBColor[1, 0, 1], PointSize[0.1], Point[p5]}];
v6 = Graphics3D[{RGBColor[1, 1, 1], PointSize[0.1], Point[p6]}];
v7 = Graphics3D[{RGBColor[1, 1, 0], PointSize[0.1], Point[p7]}];
vc = Graphics3D[{RGBColor[0, 0, 0], PointSize[0.03], Point[pc]}];

(* CREATE GRAPHICS FOR EDGES *)
e01 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p0, p1}]}];
e03 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p3}]}];
e06 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p6}]}];
e12 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p2}]}];
e17 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p7}]}];
e23 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p3}]}];
e24 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p4}]}];
e35 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p5}]}];
e45 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p5}]}];
e47 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p7}]}];
e56 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p5, p6}]}];
e67 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p6, p7}]}];

(* SHOW COMPONENTS AND SET GRAPHICS OPTIONS *)
Show[{v0, v1, v2, v3, v4, v5, v6, v7, vc , e01, e03, e06, e12,
  e17, e23, e24, e35, e45, e47, e56, e67}, AxesLabel → {"x", "y", "z"},
    Axes → True, AspectRatio → 1, ImageSize → Automatic, Boxed → True]
]
```

Function Test - Hexahedron

Below we test the functions related to creating, rotating, and visualizing the hexahedron.

■ Test *createHexahedron*

We create a hexahedron with edgelength=5 centered at (5, 5, 5).

In[103]:=
```
hexahedronPoints = createHexahedron[8, {5, 5, 5}]
```

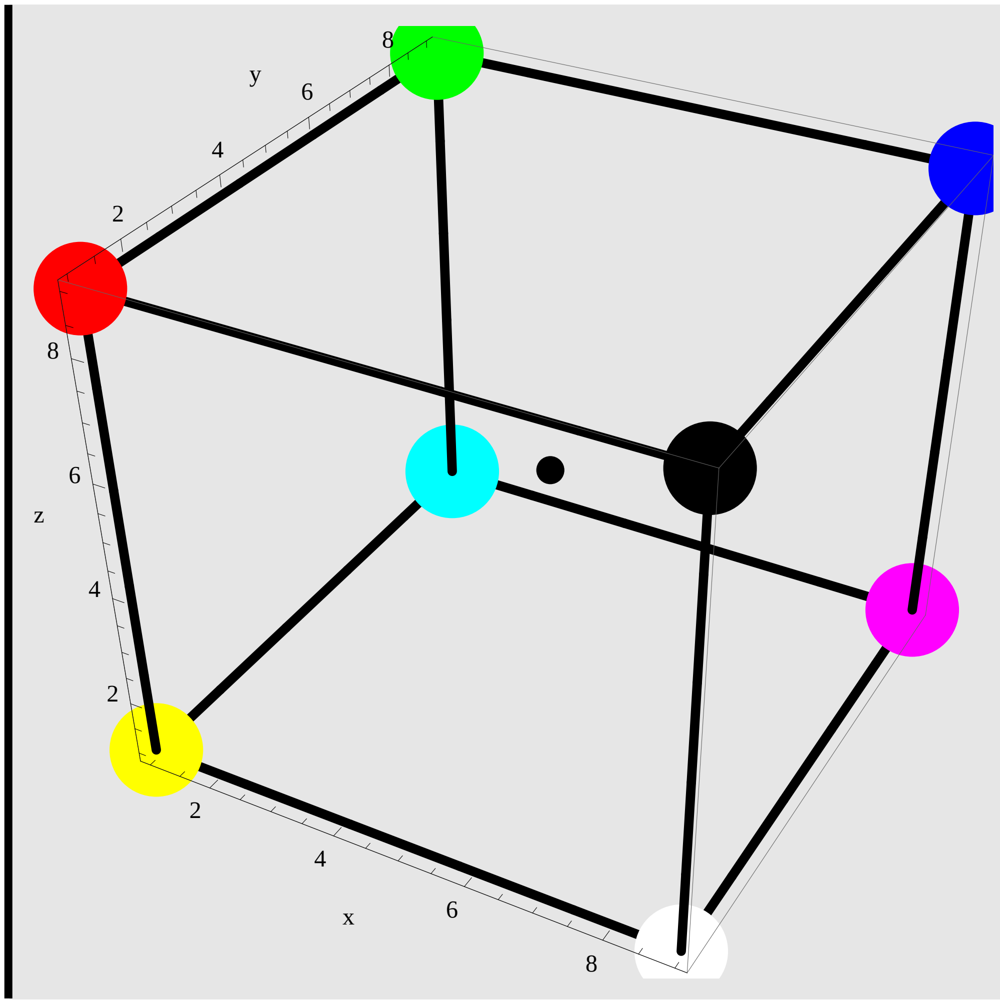
Out[103]=
```
{{9., 1., 9.}, {1., 1., 9.}, {1., 9., 9.}, {9., 9., 9.},
 {1., 9., 1.}, {9., 9., 1.}, {9., 1., 1.}, {1., 1., 1.}}
```

■ Test *drawHexahedron*

Below we visualize the hexahedron created above.

In[104]:=
```
drawHexahedron[hexahedronPoints]
```

Out[104]=



■ Test *rotateHexahedron*

Rotating the hexahedron an angle ϕ counterclockwise about the X-axis (Roll)

We rotate the hexahedron an angle $\phi = 45^\circ$ counterclockwise about the X-axis.

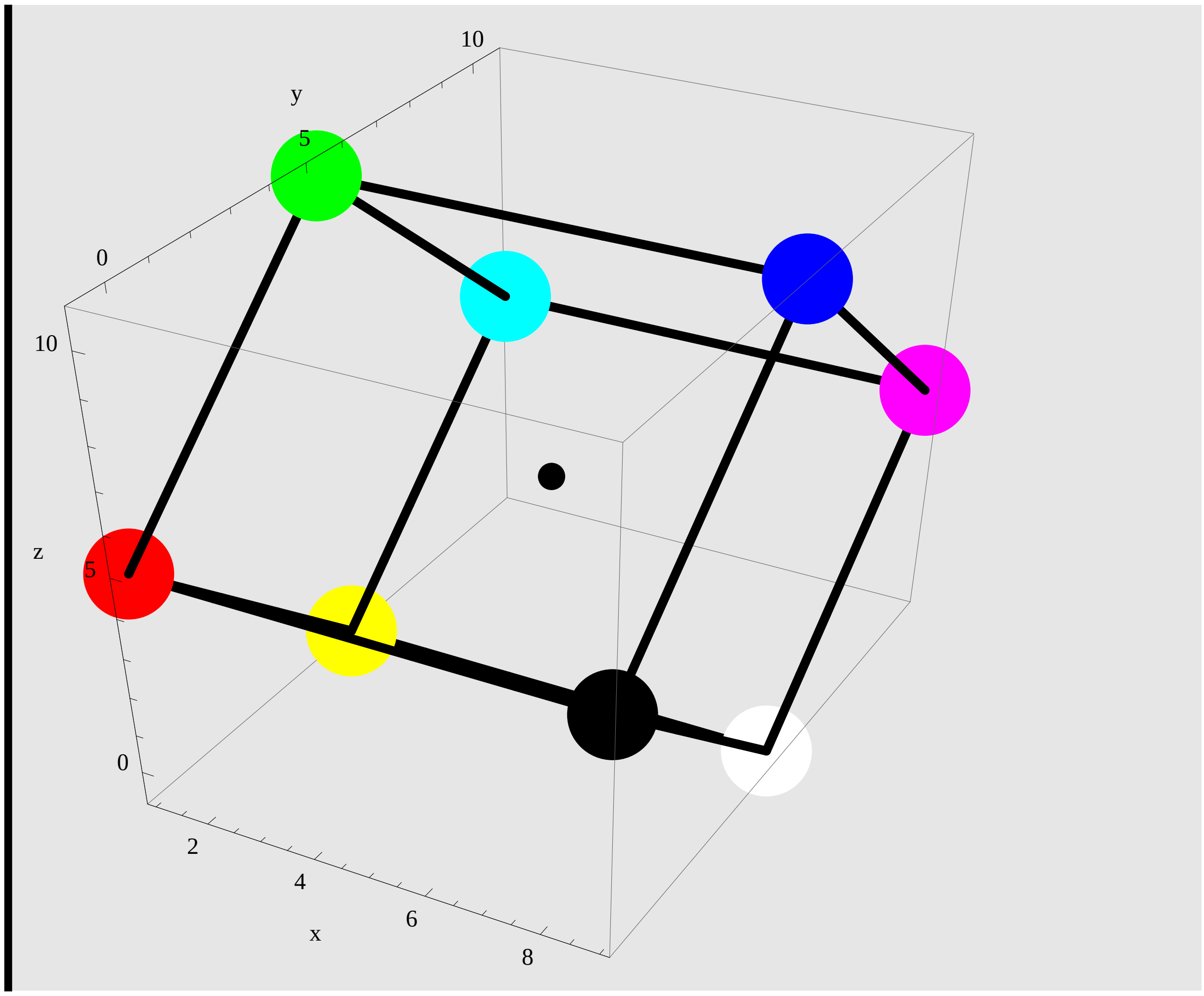
In[105]:=
```
hexahedronPointsRotatedAboutX =
  rotateHexahedron[hexahedronPoints, {45, 0, 0}];
```

The hexahedron rotated an angle $\phi = 45^\circ$ conterclockwise about the X-axis is displayed below.

In[106]:=

```
drawHexahedron[hexahedronPointsRotatedAboutX]
```

Out[106]=



Rotating the hexahedron an angle θ counterclockwise about the Y-axis (Pitch)

We rotate the hexahedron an angle $\theta = 45^{\circ}$ counterclockwise about the Y-axis.

In[107]:=

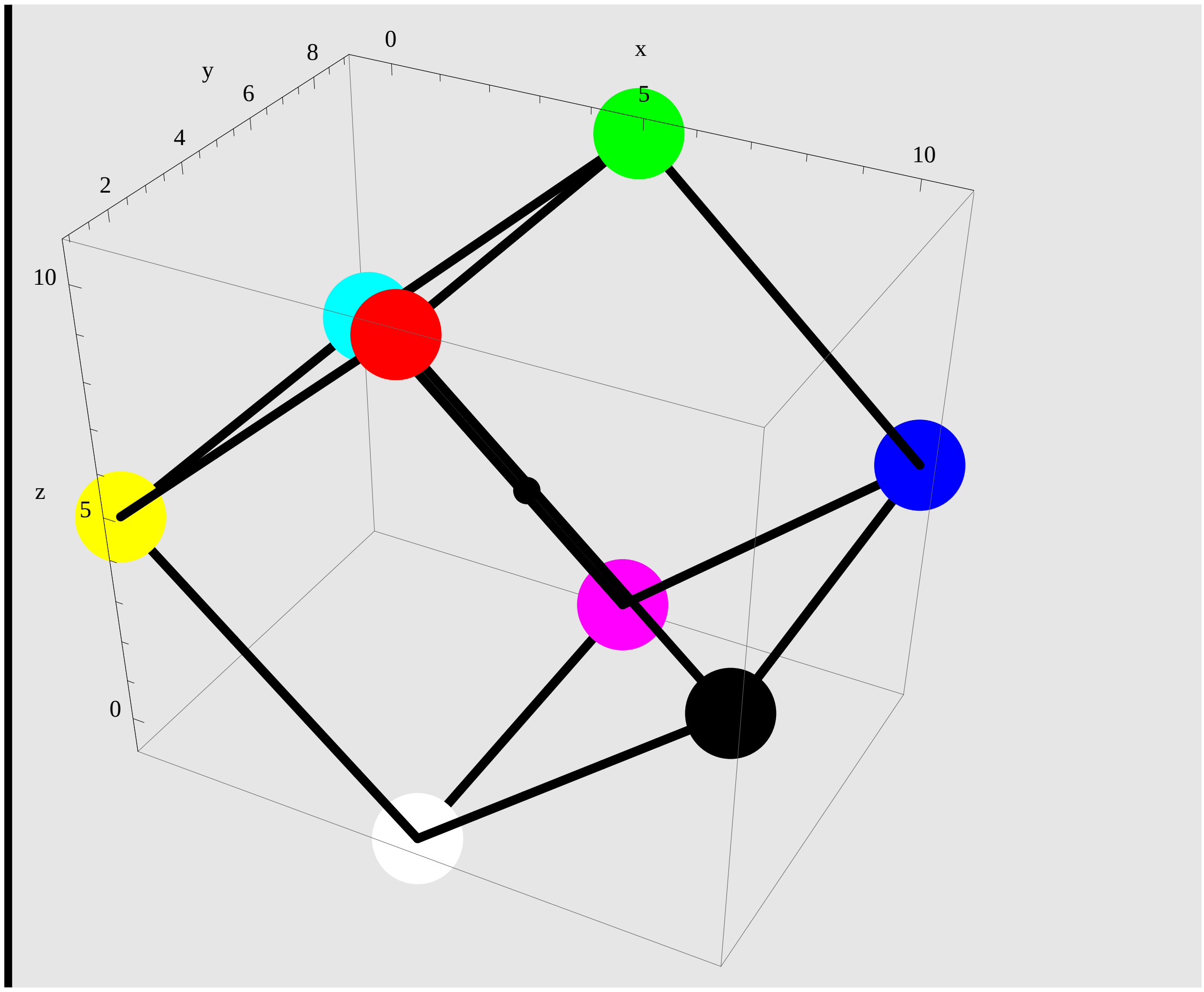
```
hexahedronPointsRotatedAboutY =
  rotateHexahedron[hexahedronPoints, {0, 45, 0}];
```

The hexahedron rotated an angle $\theta = 45^{\circ}$ conterclockwise about the Y-axis is displayed below.

In[108]:=

```
drawHexahedron[hexahedronPointsRotatedAboutY]
```

Out[108]=



Rotating the hexahedron an angle ψ counterclockwise about the Z-axis (Yaw)

We rotate the hexahedron an angle $\psi = 45^\circ$ counterclockwise about the Z-axis.

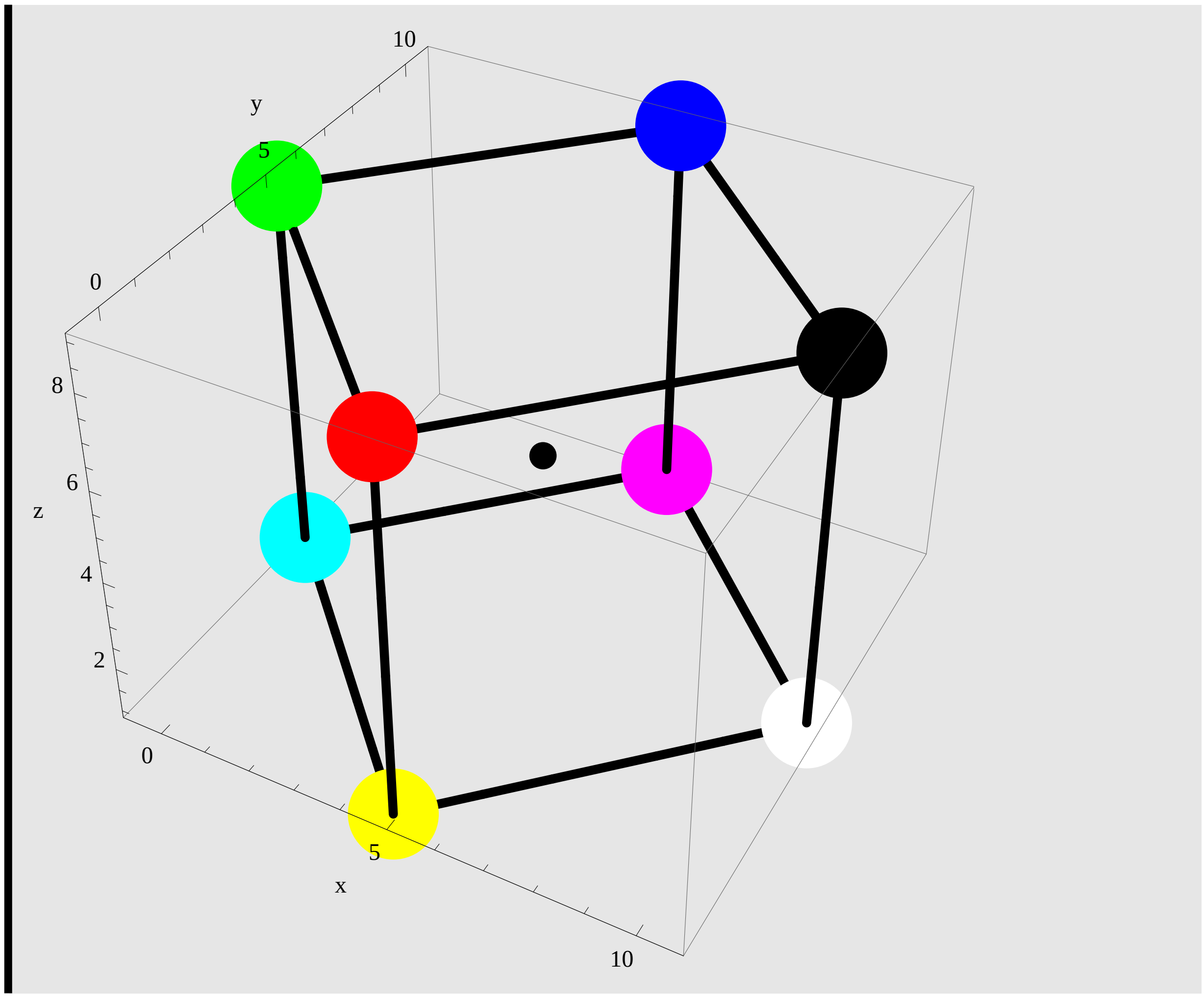
In[109]:=

```
hexahedronPointsRotatedAboutZ =
  rotateHexahedron[hexahedronPoints, {0, 0, 45}];
```

The hexahedron rotated an angle $\psi = 45^\circ$ conterclockwise about the Z-axis is displayed below.

In[110]:=
```
drawHexahedron[hexahedronPointsRotatedAboutZ]
```

Out[110]=



■ Rotating in steps

Step 1: rotating the hexahedron an angle ϕ counterclockwise about the X-axis (Roll)

First we rotate the hexahedron about the X-axis and stores the result.

In[111]:=
```
rx = rotateHexahedron[hexahedronPoints, {45, 0, 0}];
```

Step 2: rotating the hexahedron from step 1 an angle θ counterclockwise about the Y-axis (Pitch)

We use the result from step 1 and rotate the hexahedron a second time - this time about the Y-axis

In[112]:=
```
ry = rotateHexahedron[rx, {0, 45, 0}];
```

Step 3: rotating the hexahedron from step 2 an angle ψ counterclockwise about the Z-axis (Yaw)

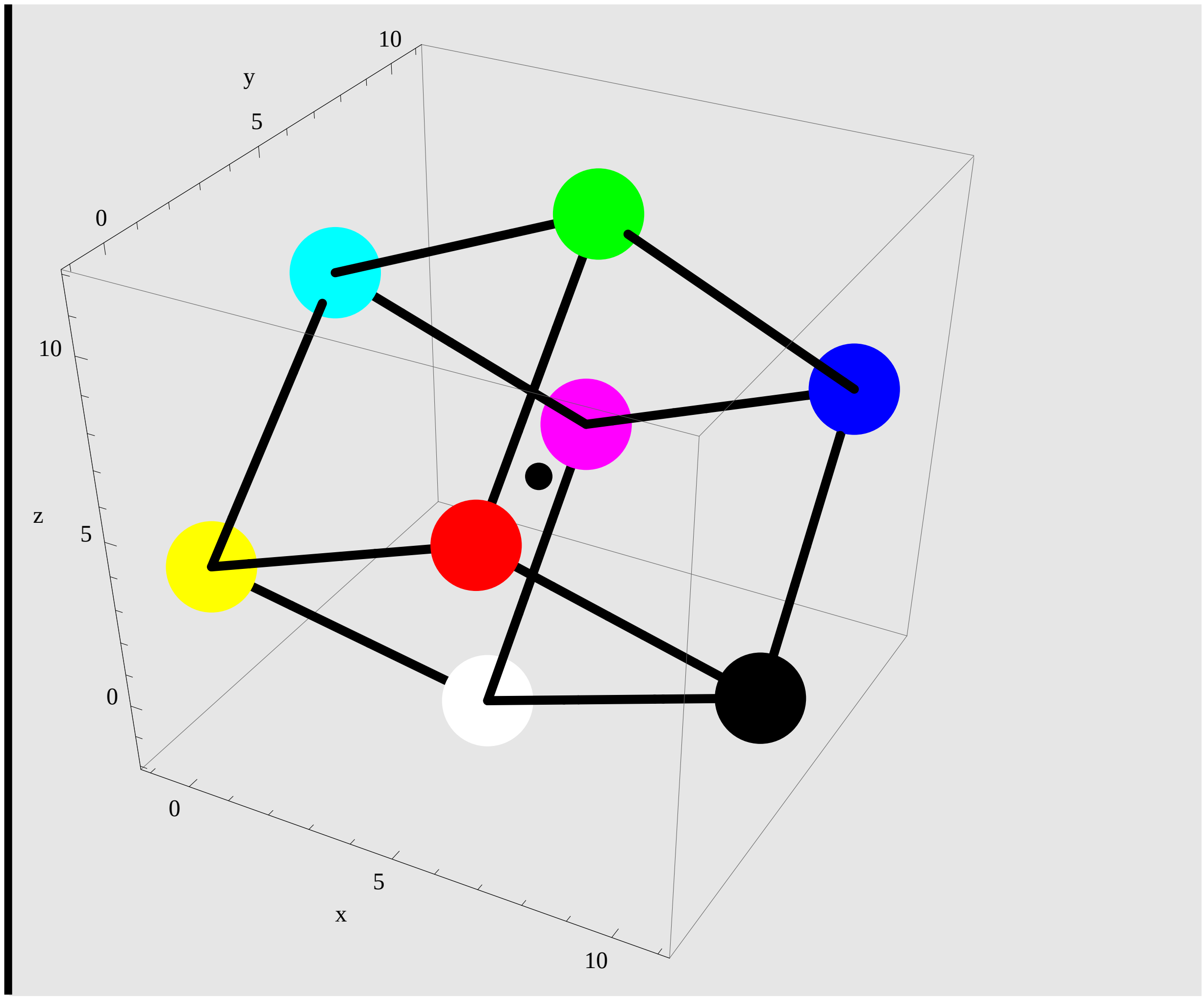
We use the result from step 2 and rotate the hexahedron a third time - this time about the Z-axis

In[113]:=
```
rz = rotateHexahedron[ry, {0, 0, 45}];
```

The result of the three consecutive rotations is shown below

In[114]:=
```
drawHexahedron[rz]
```

Out[114]=

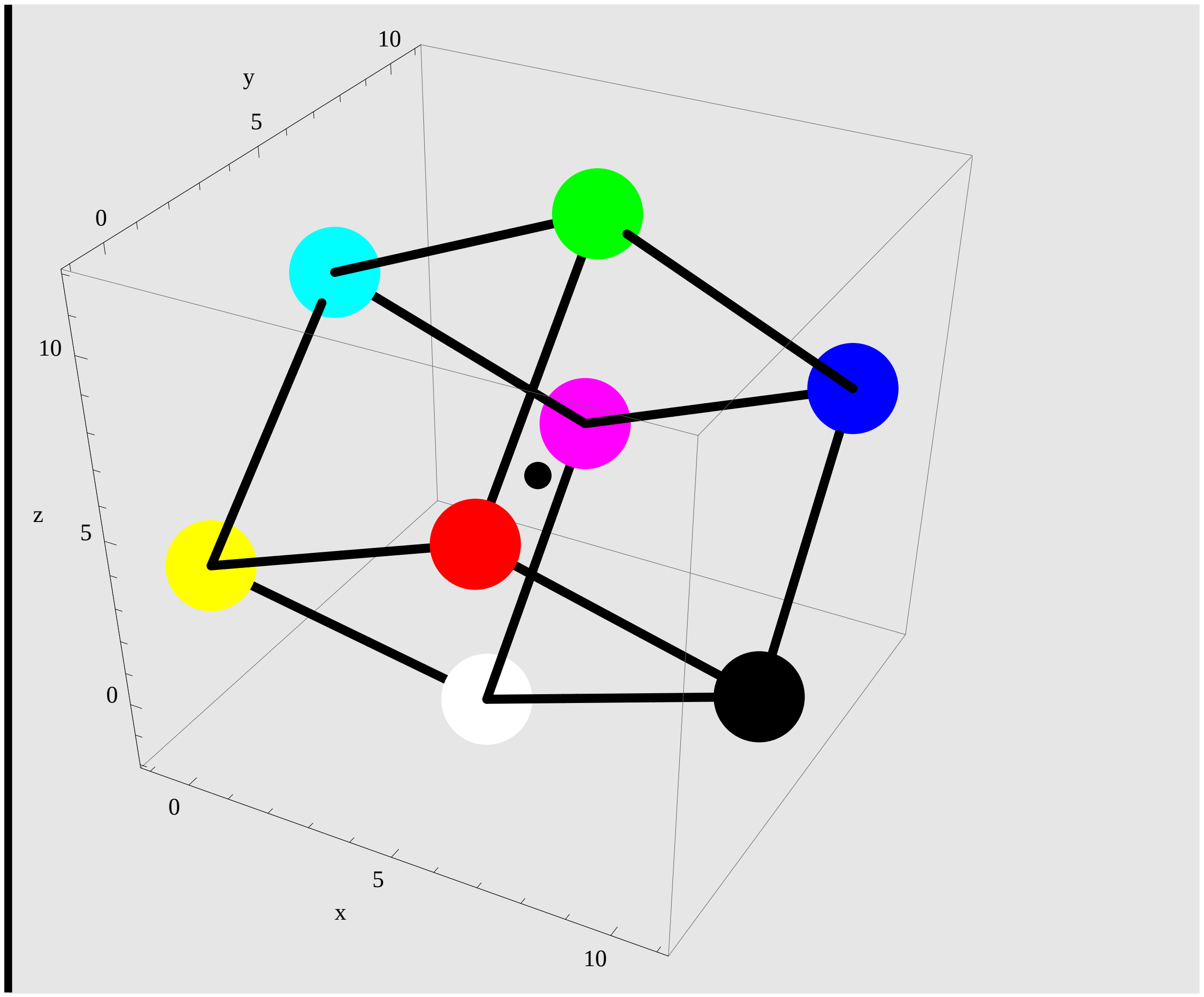


Rotating the hexahedron an angle ϕ counterclockwise about the X-axis, then an angle θ counterclockwise about the Y-axis, and finally an angle ψ counterclockwise about the Z-axis

Instead of applying the rotations separately we can calculate the resulting rotation directly. This is shown below

In[115]:=
```
drawHexahedron[rotateHexahedron[hexahedronPoints, {45, 45, 45}]]
```

Out[115]=



Icosahedron

In this section we will implement a couple of functions in *Mathematica*, that makes it possible to rotate the icosahedron, which is one of the 5 Platonic solids.

Function Definition - Icosahedron

Below functions related to creating, rotating, and visualizing the icosahedron are implemented in *Mathematica*.

■ **Define *createIcosahedron***

We define a function in *Mathematica* which creates a icosahedron based on the input parameters: *edgeLength* and *center*, where *edgeLength* specifies the length between the 12 vertices that compose the icosahedron and *center* specifies the center point of the icosahedron. The function *createIcosahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the icosahedron.

In[116]:=

```
createIcosahedron[edgeLength_?NumericQ,
  center : {xCenter_?NumericQ, yCenter_?NumericQ, zCenter_?NumericQ}] :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {points, phi, ratio, radius, p, q,
   p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11},

  (* DEFINE HELPERS *)
  phi = (1 + Sqrt[5]) / 2;
  ratio = Sqrt[10 + 2 * Sqrt[5]] / (4 * phi);
  radius = Sqrt[10 + 2 * Sqrt[5]] / 4 * edgeLength;
  p = (radius / ratio) / 2;
  q = (radius / ratio) / (2 * phi);

  (* DEFINE POINTS *)
  p0 = {xCenter, yCenter + q, zCenter - p};
  p1 = {xCenter + q, yCenter + p, zCenter};
  p2 = {xCenter - q, yCenter + p, zCenter};
  p3 = {xCenter, yCenter + q, zCenter + p};
  p4 = {xCenter, yCenter - q, zCenter + p};
  p5 = {xCenter - p, yCenter, zCenter + q};
  p6 = {xCenter, yCenter - q, zCenter - p};
  p7 = {xCenter + p, yCenter, zCenter - q};
  p8 = {xCenter + p, yCenter, zCenter + q};
  p9 = {xCenter - p, yCenter, zCenter - q};
  p10 = {xCenter + q, yCenter - p, zCenter};
  p11 = {xCenter - q, yCenter - p, zCenter};
  points = {p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11}
]
```

■ Define *rotateIcosahedron*

We define a function in *Mathematica* which rotates icosahedron about the XYZ-axes given the input parameters: *points*, ϕ, θ, and ψ. The *points* parameter is a list containing the 12 points representing the 3D coordinates of the vertices in the icosahedron. The ϕ parameter specifies the rotation angle, in degrees, about the X-axis. The θ parameter specifies the rotation angle, in degrees, about the Y-axis. The ψ parameter specifies the rotation angle, in degrees, about the Z-axis. The function *rotateIcosahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the icosahedron after applying the rotation.

In[117]:=

```
rotateIcosahedron[points_,
   angle : {phi_?NumericQ, theta_?NumericQ, psi_?NumericQ}] /;
  (Length[points] == 12) :=
Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {rmat, p0, p1, p2, p3, p4, p5, p6,
   p7, p8, p9, p10, p11, pc, p0new,
   p1new, p2new, p3new, pnew4, pnew5,
   pnew6, pnew7, pnew8, pnew9, pnew10,
   pnew11, newpts},

  (* DEFINE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  p4 = points[[5]];
  p5 = points[[6]];
  p6 = points[[7]];
  p7 = points[[8]];
  p8 = points[[9]];
  p9 = points[[10]];
  p10 = points[[11]];
  p11 = points[[12]];
  pc = (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9 + p10 + p11) / 12;

  (* CREATE ROTATION MATRIX *)
  rmat = RotationMatrix[psi Degree, {0, 0, 1}].
    RotationMatrix[theta Degree, {0, 1, 0}].
    RotationMatrix[phi Degree, {1, 0, 0}];

  (* CALCULATE NEW POINTS *)
  p0new = pc + rmat.(p0 - pc);
  p1new = pc + rmat.(p1 - pc);
  p2new = pc + rmat.(p2 - pc);
  p3new = pc + rmat.(p3 - pc);
  p4new = pc + rmat.(p4 - pc);
  p5new = pc + rmat.(p5 - pc);
  p6new = pc + rmat.(p6 - pc);
  p7new = pc + rmat.(p7 - pc);
  p8new = pc + rmat.(p8 - pc);
  p9new = pc + rmat.(p9 - pc);
  p10new = pc + rmat.(p10 - pc);
  p11new = pc + rmat.(p11 - pc);
  newpts = {p0new, p1new, p2new, p3new, p4new, p5new, p6new, p7new,
    p8new, p9new, p10new, p11new}
]
```

Define *drawIcosahedron*

We define a function in *Mathematica* which is able to display a icosahedron given the input parameter: *points*. The *points* parameter is a list containing the 12 points representing the 3D coordinates of the vertices in the icosahedron. The function *drawIcosahedron* draws the vertices and edges which compose the icosahedron and displays these in a single graphical object.

In[118]:=

```
drawIcosahedron[points_] /; (Length[points] == 12) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, pc,
   v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, vc,
   e01, e02, e03, e12, e13, e23},

  (* CREATE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  p4 = points[[5]];
  p5 = points[[6]];
  p6 = points[[7]];
  p7 = points[[8]];
  p8 = points[[9]];
  p9 = points[[10]];
  p10 = points[[11]];
  p11 = points[[12]];
  pc = (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9 + p10 + p11) / 12;

  (* CREATE GRAPHICS FOR VERTICES *)
  v0 = Graphics3D[{Red, PointSize[0.04], Point[p0]}];
  v1 = Graphics3D[{Blue, PointSize[0.04], Point[p1]}];
  v2 = Graphics3D[{Green, PointSize[0.04], Point[p2]}];
  v3 = Graphics3D[{Yellow, PointSize[0.04], Point[p3]}];
  v4 = Graphics3D[{Cyan, PointSize[0.04], Point[p4]}];
  v5 = Graphics3D[{Magenta, PointSize[0.04], Point[p5]}];
  v6 = Graphics3D[{Orange, PointSize[0.04], Point[p6]}];
  v7 = Graphics3D[{Gray, PointSize[0.04], Point[p7]}];
  v8 = Graphics3D[{Purple, PointSize[0.04], Point[p8]}];
  v9 = Graphics3D[{Brown, PointSize[0.04], Point[p9]}];
  v10 = Graphics3D[{Black, PointSize[0.04], Point[p10]}];
  v11 = Graphics3D[{Pink, PointSize[0.04], Point[p11]}];
  vc = Graphics3D[{RGBColor[0, 0, 0], PointSize[0.015], Point[pc]}];

  (* CREATE GRAPHICS FOR EDGES *)
```

```
e01 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p0, p1}]}];
e02 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p2}]}];
e06 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p6}]}];
e07 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p7}]}];
e09 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p9}]}];
e12 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p2}]}];
e13 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p3}]}];
e17 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p7}]}];
e18 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p8}]}];
e23 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p3}]}];
e25 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p5}]}];
e29 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p9}]}];
e34 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p4}]}];
e35 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p5}]}];
e38 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p8}]}];
e45 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p5}]}];
e48 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p8}]}];
e410 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p4, p10}]}];
e411 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p4, p11}]}];
e59 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p5, p9}]}];
e511 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p5, p11}]}];
e67 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p6, p7}]}];
e69 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p6, p9}]}];
e610 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p6, p10}]}];
e611 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p6, p11}]}];
e78 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p7, p8}]}];
e710 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p7, p10}]}];
e810 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p8, p10}]}];
e911 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p9, p11}]}];
e1011 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
    Line[{p10, p11}]}];

Show[{v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, vc, e01,
  e02, e09, e06, e07, e12, e13, e17, e18, e23, e25, e29, e34,
  e35, e38, e45, e48, e410, e411, e59, e511, e67, e69, e610, e611,
  e78, e710, e810, e911, e1011}, AxesLabel → {"x", "y", "z"},
 AspectRatio → 1, Axes → True, PlotRange → All, ImageSize → Automatic,
    Boxed → True]
  ]
```

Function Test - Icosahedron

Below we test the functions related to creating, rotating, and visualizing the icosahedron.

■ Test *createIcosahedron*

We create a icosahedron with edgelength=5 centered at (5, 5, 5).

In[119]:=
```
icosahedronPoints = createIcosahedron[5, {5, 5, 5}]
```

Out[119]=

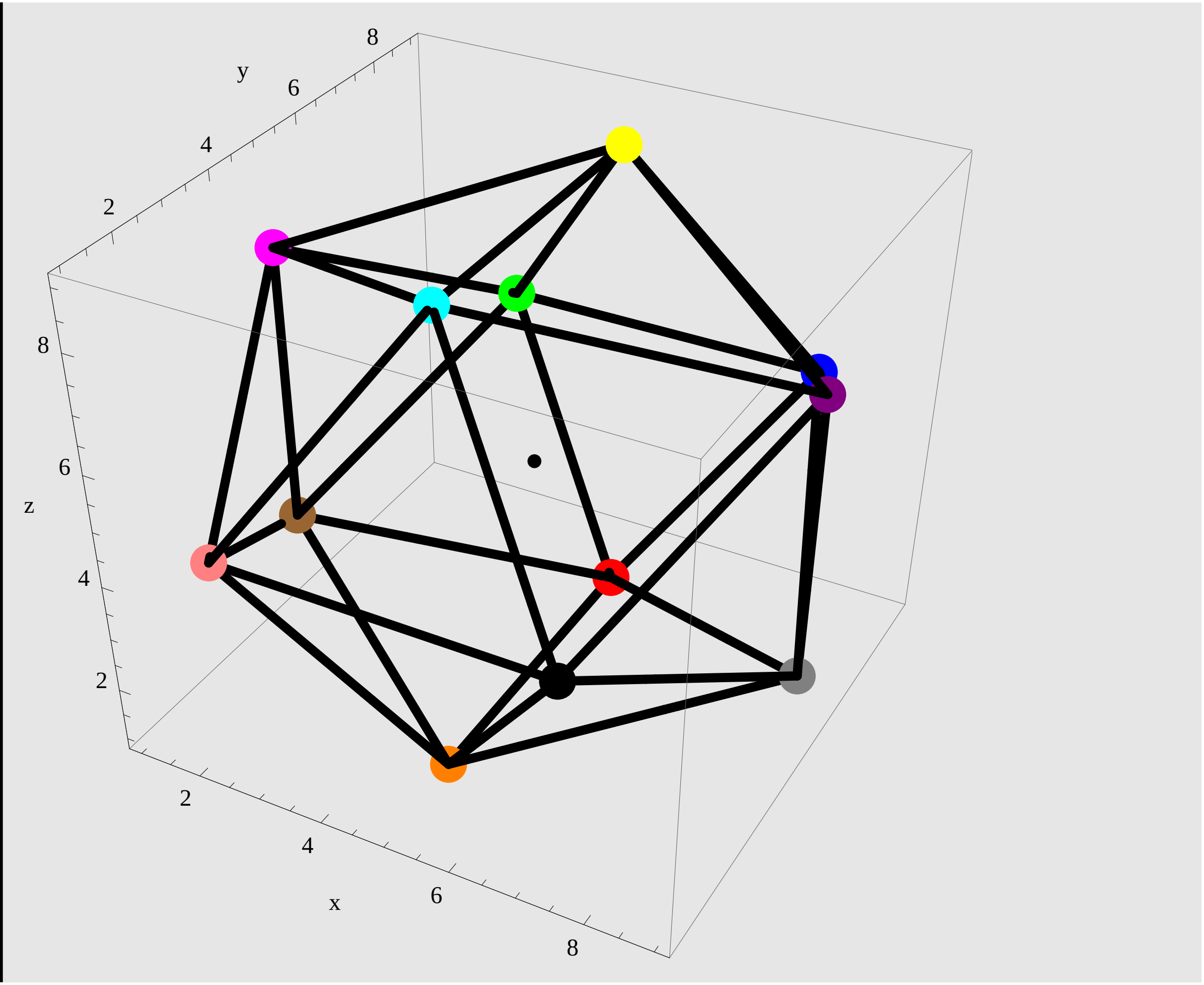
$$\left\{\left\{5, \frac{15}{2}, 5-\frac{5}{4}\left(1+\sqrt{5}\right)\right\}, \left\{\frac{15}{2}, 5+\frac{5}{4}\left(1+\sqrt{5}\right), 5\right\}, \left\{\frac{5}{2}, 5+\frac{5}{4}\left(1+\sqrt{5}\right), 5\right\},\right.$$
$$\left\{5, \frac{15}{2}, 5+\frac{5}{4}\left(1+\sqrt{5}\right)\right\}, \left\{5, \frac{5}{2}, 5+\frac{5}{4}\left(1+\sqrt{5}\right)\right\}, \left\{5-\frac{5}{4}\left(1+\sqrt{5}\right), 5, \frac{15}{2}\right\},$$
$$\left\{5, \frac{5}{2}, 5-\frac{5}{4}\left(1+\sqrt{5}\right)\right\}, \left\{5+\frac{5}{4}\left(1+\sqrt{5}\right), 5, \frac{5}{2}\right\}, \left\{5+\frac{5}{4}\left(1+\sqrt{5}\right), 5, \frac{15}{2}\right\},$$
$$\left.\left\{5-\frac{5}{4}\left(1+\sqrt{5}\right), 5, \frac{5}{2}\right\}, \left\{\frac{15}{2}, 5-\frac{5}{4}\left(1+\sqrt{5}\right), 5\right\}, \left\{\frac{5}{2}, 5-\frac{5}{4}\left(1+\sqrt{5}\right), 5\right\}\right\}$$

■ Test *drawIcosahedron*

Below we visualize the icosahedron created above.

In[120]:=
```
drawIcosahedron[icosahedronPoints]
```

Out[120]=



■ Test *rotateIcosahedron*

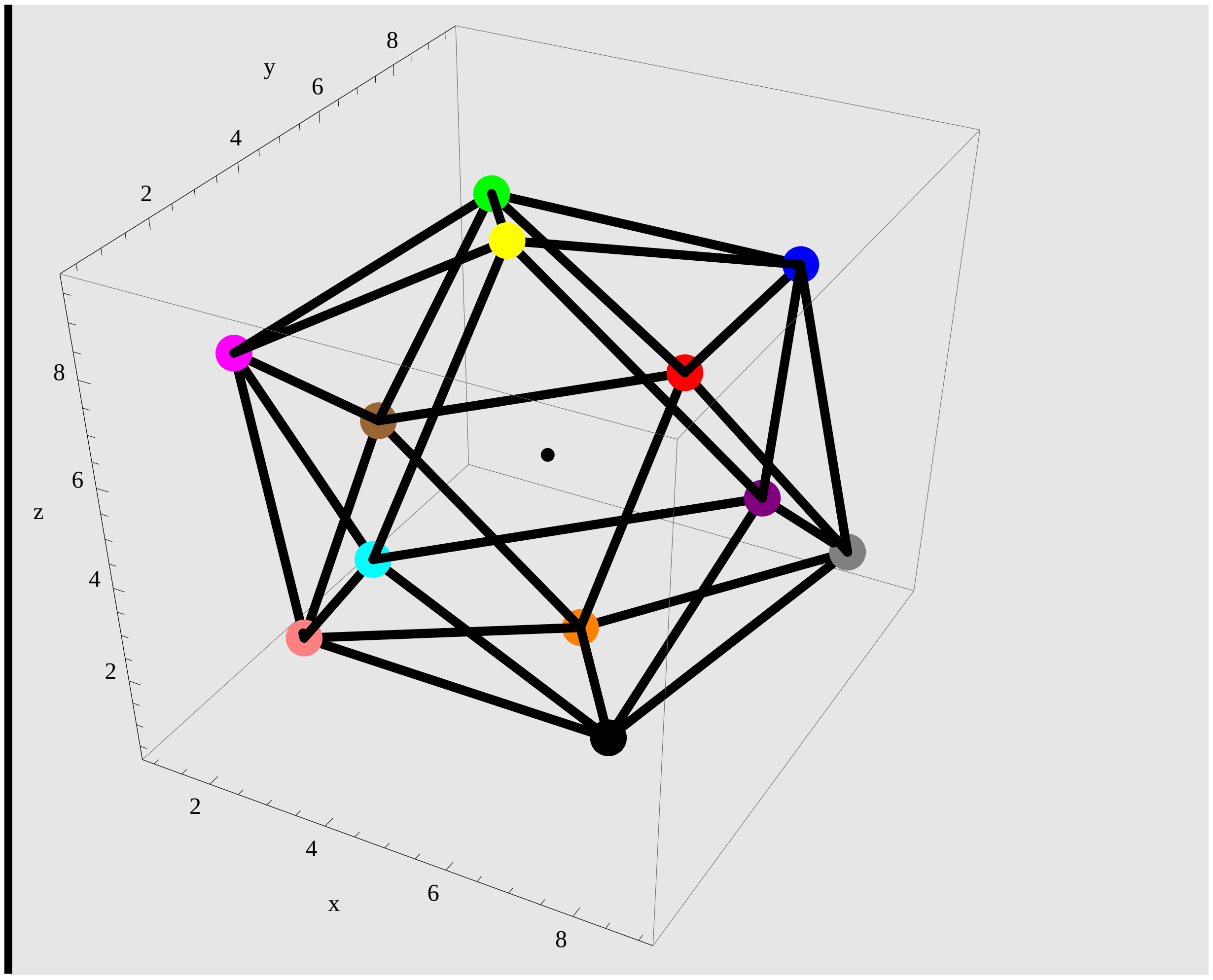
Rotating the icosahedron an angle ϕ counterclockwise about the X-axis (Roll)

We rotate the icosahedron an angle $\phi = 45^{\circ}$ counterclockwise about the X-axis.

In[121]:=
```
icosahedronPointsRotatedAboutX =
  rotateIcosahedron[icosahedronPoints, {45, 0, 0}];
```

The icosahedron rotated an angle $\phi = 45^{\circ}$ conterclockwise about the X-axis is displayed below.

In[122]:=
```
drawIcosahedron[icosahedronPointsRotatedAboutX]
```

Out[122]=


Rotating the icosahedron an angle θ counterclockwise about the Y-axis (Pitch)

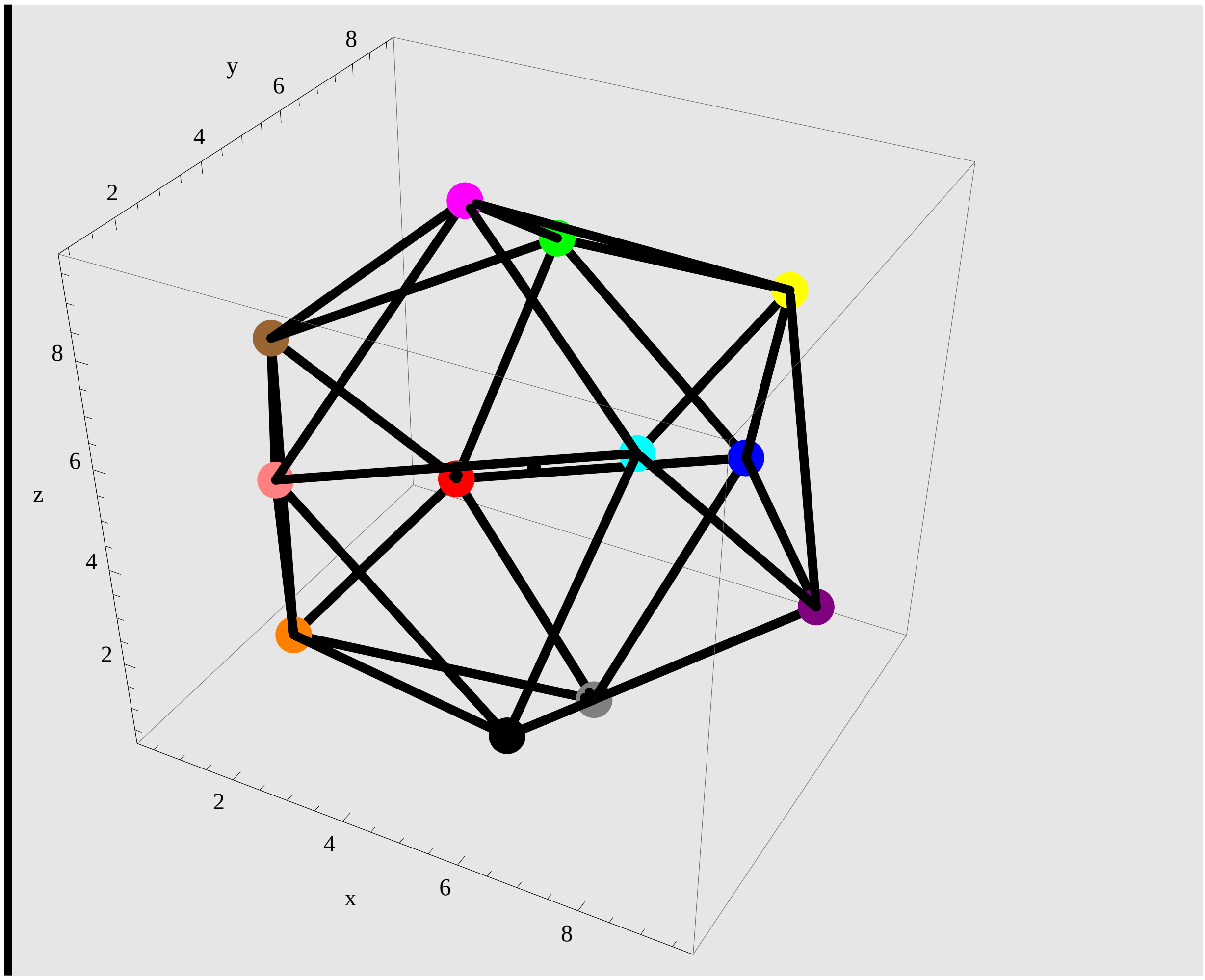
We rotate the icosahedron an angle $\theta = 45^\circ$ counterclockwise about the Y-axis.

In[123]:=
```
icosahedronPointsRotatedAboutY =
  rotateIcosahedron[icosahedronPoints, {0, 45, 0}];
```

The icosahedron rotated an angle $\theta = 45^\circ$ conterclockwise about the Y-axis is displayed below.

In[124]:=
```
drawIcosahedron[icosahedronPointsRotatedAboutY]
```

Out[124]=



Rotating the icosahedron an angle ψ counterclockwise about the Z-axis (Yaw)

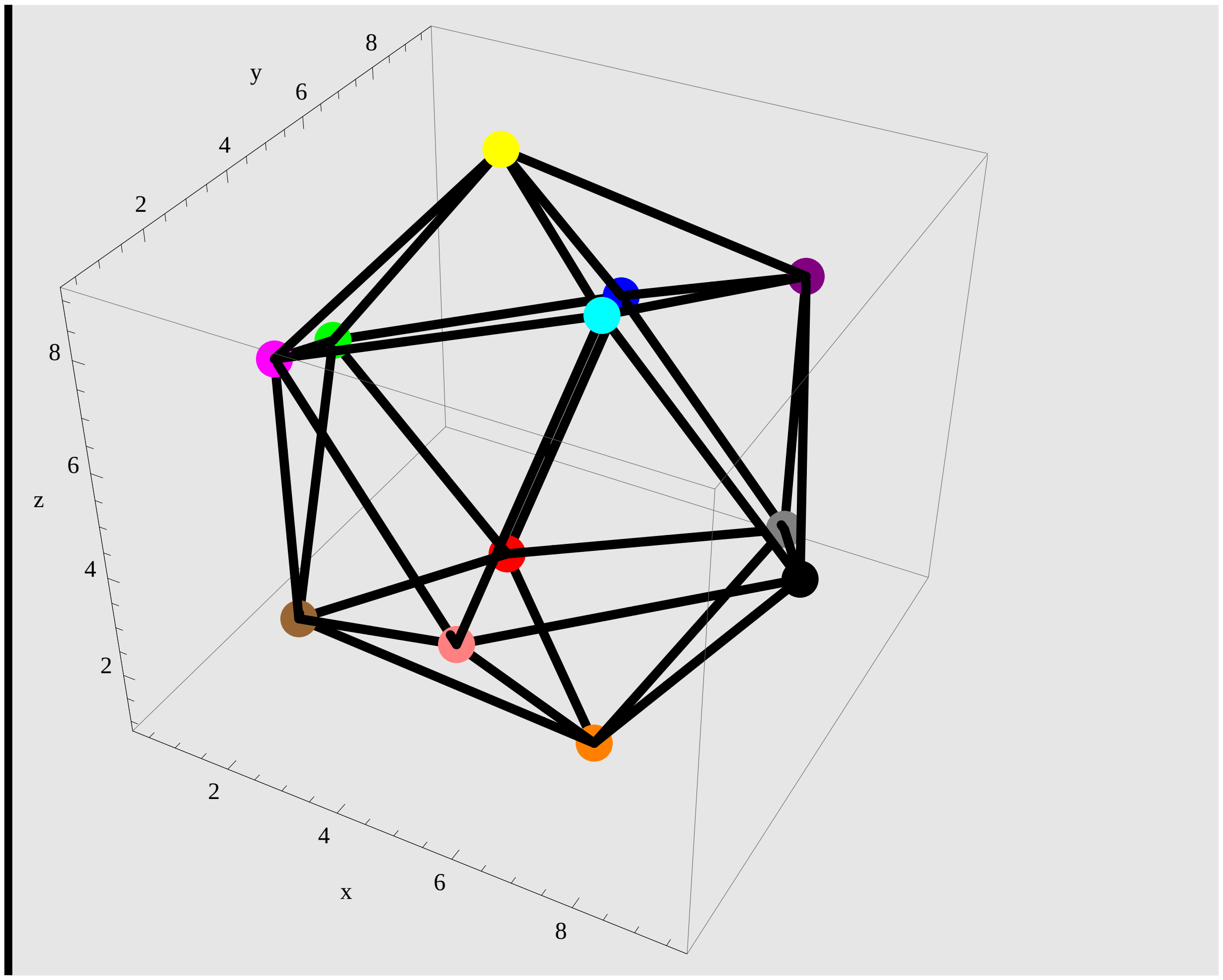
We rotate the icosahedron an angle $\psi = 45^\circ$ counterclockwise about the Z-axis.

In[125]:=
```
icosahedronPointsRotatedAboutZ =
  rotateIcosahedron[icosahedronPoints, {0, 0, 45}];
```

The icosahedron□ rotated an angle $\psi = 45^\circ$ conterclockwise about the Z-axis is displayed below.

In[126]:=
```
drawIcosahedron[icosahedronPointsRotatedAboutZ]
```

Out[126]=



■ Rotating in steps

Step 1: rotating the icosahedron an angle ϕ counterclockwise about the X-axis (Roll)

First we rotate the icosahedron about the X-axis and stores the result.

In[127]:=
```
rx = rotateIcosahedron[icosahedronPoints, {45, 0, 0}];
```

Step 2: rotating the icosahedron from step 1 an angle θ counterclockwise about the Y-axis (Pitch)

We use the result from step 1 and rotate the icosahedron a second time - this time about the Y-axis

In[128]:=
```
ry = rotateIcosahedron[rx, {0, 45, 0}];
```

Step 3: rotating the icosahedron from step 2 an angle ψ counterclockwise about the Z-axis (Yaw)

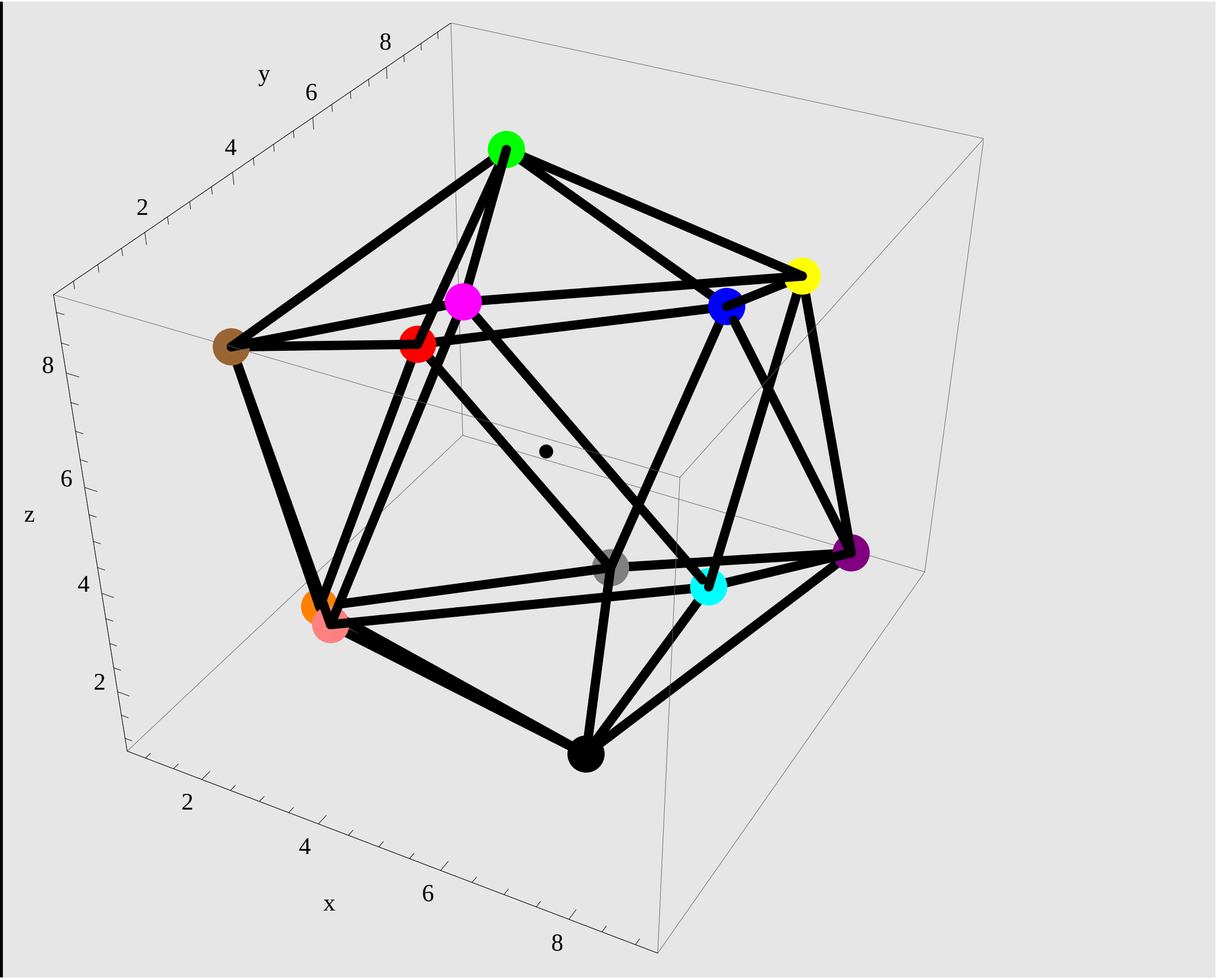
We use the result from step 2 and rotate the icosahedron a third time - this time about the Z-axis

In[129]:=
```
rz = rotateIcosahedron[ry, {0, 0, 45}];
```

The result of the three consecutive rotations is shown below

In[130]:=
```
drawIcosahedron[rz]
```

Out[130]=



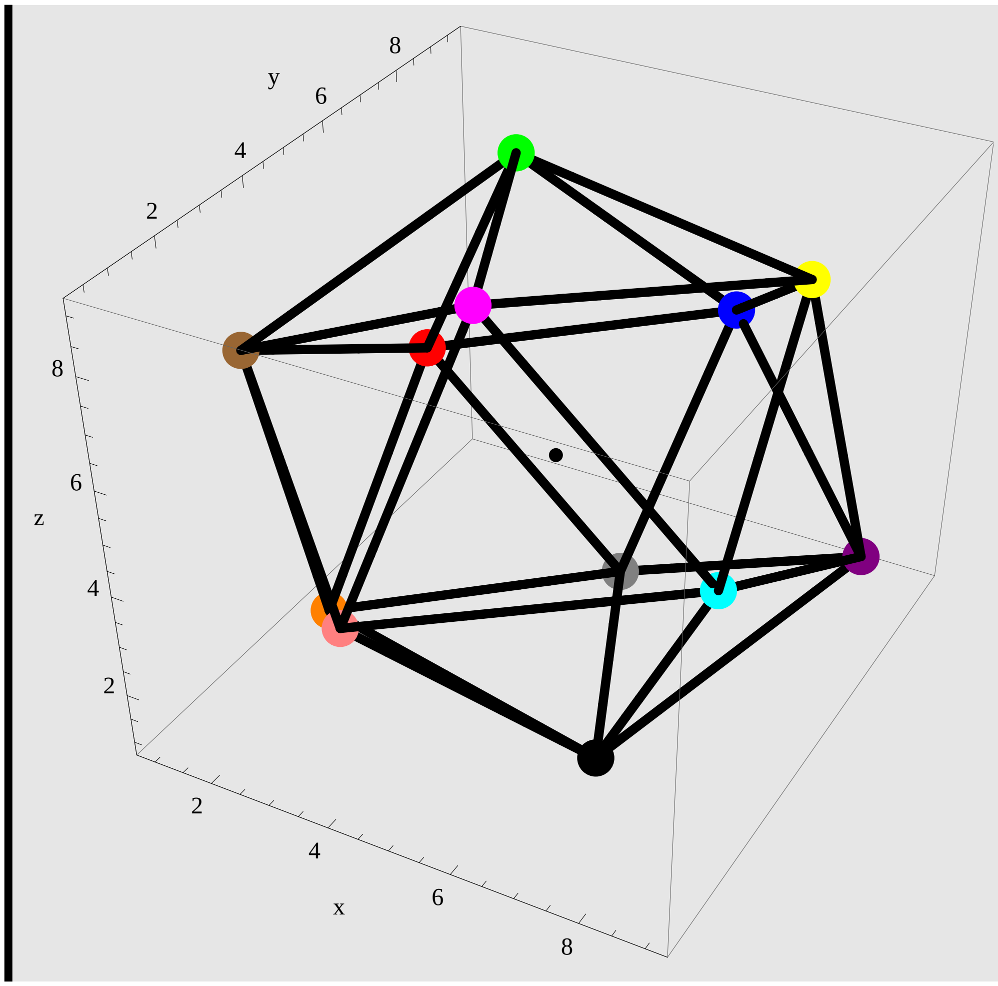
Rotating the icosahedron an angle ϕ counterclockwise about the X-axis, then an angle θ counterclockwise about the Y-axis, and finally an angle ψ counterclockwise about the Z-axis

Instead of applying the rotations separately we can calculate the resulting rotation directly. This is shown below

In[131]:=

```
drawIcosahedron[rotateIcosahedron[icosahedronPoints, {45, 45, 45}]]
```

Out[131]=



Dodecahedron

In this section we will implement a couple of functions in *Mathematica*, that makes it possible to rotate the dodecahedron, which is one of the 5 Platonic solids.

Function Definition - Dodecahedron

Below functions related to creating, rotating, and visualizing the dodecahedron are implemented in *Mathematica*.

■ Define *createDodecahedron*

We define a function in *Mathematica* which creates a dodecahedron based on the input parameters: *edgeLength* and *center*, where *edgeLength* specifies the length between the 20 vertices that compose the dodecahedron and *center* specifies the center point of the dodecahedron. The function *createDodecahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the dodecahedron.

In[132]:=

```
createDodecahedron[edgeLength_?NumericQ,
  center : {xCenter_?NumericQ, yCenter_?NumericQ, zCenter_?NumericQ}] :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {points, phi1, phi2, phia, phib, phic,
   phid, r, theta72, thetab, theta},

  (* DEFINE VARIABLES AND CONSTANTS *)
  phi1 = 52.62;
  phi2 = 10.81;
  phia = (π / 180) * phi1;
  phib = (π / 180) * phi2;
  phic = - (π / 180) * phi2;
  phid = - (π / 180) * phi1;
  r = ((Sqrt[15] + Sqrt[3]) / 4) * edgeLength;
  theta72 = (72 / 180) * π;
  thetab = theta72 / 2;
  theta = 0;

  (* CREATE PLACEHOLDER FOR POINTS *)
  points = Table[{0, 0, 0}, {i, 1, 20}];

  For[i = 1, i <= 5, i++,
   points[[i, 1]] = r * Cos[theta] * Cos[phia];
   points[[i, 2]] = r * Sin[theta] * Cos[phia];
   points[[i, 3]] = r * Sin[phia];
   theta = theta + theta72;
  ];
  theta = 0;
  For[i = 6, i <= 10, i++,
   points[[i, 1]] = r * Cos[theta] * Cos[phib];
   points[[i, 2]] = r * Sin[theta] * Cos[phib];
   points[[i, 3]] = r * Sin[phib];
   theta = theta + theta72;
  ];
  theta = thetab;
  For[i = 11, i <= 15, i++,
   points[[i, 1]] = r * Cos[theta] * Cos[phic];
   points[[i, 2]] = r * Sin[theta] * Cos[phic];
   points[[i, 3]] = r * Sin[phic];
   theta = theta + theta72;
  ];
  theta = thetab;
  For[i = 16, i <= 20, i++,
   points[[i, 1]] = r * Cos[theta] * Cos[phid];
   points[[i, 2]] = r * Sin[theta] * Cos[phid];
   points[[i, 3]] = r * Sin[phid];
   theta = theta + theta72;
  ];
  Return[points];
 ]
```

■ Define *rotateDodecahedron*

We define a function in *Mathematica* which rotates dodecahedron about the XYZ-axes given the input parameters: *points*, ϕ, θ, and ψ. The *points* parameter is a list containing the 20 points representing the 3D coordinates of the vertices in the dodecahedron. The ϕ parameter specifies the rotation angle, in degrees, about the X-axis. The θ parameter specifies the rotation angle, in degrees, about the Y-axis. The ψ parameter specifies the rotation angle, in degrees, about the Z-axis. The function *rotateDodecahedron* returns a list of points, which gives the Cartensian positions of the each of the vertices in the dodecahedron after applying the rotation.

In[133]:=

```
rotateDodecahedron[points_,
   angle : {phi_?NumericQ, theta_?NumericQ, psi_?NumericQ}] /;
  (Length[points] == 20) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {rmat, p0, p1, p2, p3, p4, p5, p6,
   p7, p8, p9, p10, p11, p12, p13,
   p14, p15, p16, p17, p18, p19, pc,
   p0new, p1new, p2new, p3new, pnew4,
   pnew5, pnew6, pnew7, pnew8, pnew9,
   pnew10, pnew11, pnew12, pnew13,
   pnew14, pnew15, pnew16, pnew17,
   pnew18, pnew19, newpts},

  (* DEFINE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  p4 = points[[5]];
  p5 = points[[6]];
  p6 = points[[7]];
  p7 = points[[8]];
  p8 = points[[9]];
  p9 = points[[10]];
  p10 = points[[11]];
```

```
    p11 = points[[12]];
    p12 = points[[13]];
    p13 = points[[14]];
    p14 = points[[15]];
    p15 = points[[16]];
    p16 = points[[17]];
    p17 = points[[18]];
    p18 = points[[19]];
    p19 = points[[20]];
    pc =
     (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9 + p10 + p11 + p12 + p13 + p14 +
        p15 + p16 + p17 + p18 + p19) / 20;

    (* CREATE ROTATION MATRIX *)
    rmat = RotationMatrix[psi Degree, {0, 0, 1}].
      RotationMatrix[theta Degree, {0, 1, 0}].
      RotationMatrix[phi Degree, {1, 0, 0}];

    (* CALCULATE NEW POINTS *)
    p0new = pc + rmat.(p0 - pc);
    p1new = pc + rmat.(p1 - pc);
    p2new = pc + rmat.(p2 - pc);
    p3new = pc + rmat.(p3 - pc);
    p4new = pc + rmat.(p4 - pc);
    p5new = pc + rmat.(p5 - pc);
    p6new = pc + rmat.(p6 - pc);
    p7new = pc + rmat.(p7 - pc);
    p8new = pc + rmat.(p8 - pc);
    p9new = pc + rmat.(p9 - pc);
    p10new = pc + rmat.(p10 - pc);
    p11new = pc + rmat.(p11 - pc);
    p12new = pc + rmat.(p12 - pc);
    p13new = pc + rmat.(p13 - pc);
    p14new = pc + rmat.(p14 - pc);
    p15new = pc + rmat.(p15 - pc);
    p16new = pc + rmat.(p16 - pc);
    p17new = pc + rmat.(p17 - pc);
    p18new = pc + rmat.(p18 - pc);
    p19new = pc + rmat.(p19 - pc);

    newpts = {p0new, p1new, p2new, p3new, p4new,
      p5new, p6new, p7new, p8new, p9new, p10new,
      p11new, p12new, p13new, p14new, p15new,
      p16new, p17new, p18new, p19new}
   ]
```

Define *drawDodecahedron*

We define a function in *Mathematica* which is able to display a dodecahedron given the input parameter: *points*. The *points* parameter is a list containing the 20 points representing the 3D coordinates of the vertices in the dodecahedron. The function *drawDodecahedron* draws the vertices and edges which compose the dodecahedron and displays these in a single graphical object.

In[134]:=
```
drawDodecahedron[points_] /; (Length[points] == 20) :=
 Module[
  (* ALLOCATE LOCAL VARIABLES *)
  {p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, pc,
   v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, vc,
   e01, e02, e03, e12, e13, e23},

  (* CREATE POINTS *)
  p0 = points[[1]];
  p1 = points[[2]];
  p2 = points[[3]];
  p3 = points[[4]];
  p4 = points[[5]];
  p5 = points[[6]];
  p6 = points[[7]];
  p7 = points[[8]];
  p8 = points[[9]];
  p9 = points[[10]];
  p10 = points[[11]];
  p11 = points[[12]];
  p12 = points[[13]];
  p13 = points[[14]];
  p14 = points[[15]];
  p15 = points[[16]];
  p16 = points[[17]];
  p17 = points[[18]];
  p18 = points[[19]];
  p19 = points[[20]];
  pc =
   (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9 + p10 + p11 + p12 + p13 + p14 +
      p15 + p16 + p17 + p18 + p19) / 20;

  (* CREATE GRAPHICS FOR VERTICES *)
  v0 = Graphics3D[{Red, PointSize[0.04], Point[p0]}];
  v1 = Graphics3D[{Blue, PointSize[0.04], Point[p1]}];
  v2 = Graphics3D[{Green, PointSize[0.04], Point[p2]}];
  v3 = Graphics3D[{Yellow, PointSize[0.04], Point[p3]}];
  v4 = Graphics3D[{Cyan, PointSize[0.04], Point[p4]}];
```

```
v5 = Graphics3D[{Magenta, PointSize[0.04], Point[p5]}];
v6 = Graphics3D[{Orange, PointSize[0.04], Point[p6]}];
v7 = Graphics3D[{Gray, PointSize[0.04], Point[p7]}];
v8 = Graphics3D[{Purple, PointSize[0.04], Point[p8]}];
v9 = Graphics3D[{Brown, PointSize[0.04], Point[p9]}];
v10 = Graphics3D[{Black, PointSize[0.04], Point[p10]}];
v11 = Graphics3D[{LightRed, PointSize[0.04], Point[p11]}];
v12 = Graphics3D[{LightBlue, PointSize[0.04], Point[p12]}];
v13 = Graphics3D[{LightGreen, PointSize[0.04], Point[p13]}];
v14 = Graphics3D[{LightYellow, PointSize[0.04], Point[p14]}];
v15 = Graphics3D[{LightCyan, PointSize[0.04], Point[p15]}];
v16 = Graphics3D[{LightMagenta, PointSize[0.04], Point[p16]}];
v17 = Graphics3D[{LightOrange, PointSize[0.04], Point[p17]}];
v18 = Graphics3D[{LightGray, PointSize[0.04], Point[p18]}];
v19 = Graphics3D[{LightPurple, PointSize[0.04], Point[p19]}];
vc = Graphics3D[{RGBColor[0, 0, 0], PointSize[0.015], Point[pc]}];

(* CREATE GRAPHICS FOR EDGES *)
(* CREATE PENTAGONS *)
e01 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p0, p1}]}];
e12 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p2}]}];
e23 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p3}]}];
e34 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p4}]}];
e40 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p0}]}];

e01 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p1}]}];
e16 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p6}]}];
e610 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p6, p10}]}];
e510 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p5, p10}]}];
e05 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p5}]}];

e12 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p1, p2}]}];
e27 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p7}]}];
e711 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p7, p11}]}];
e116 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p11, p6}]}];
e61 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p6, p1}]}];

e23 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p2, p3}]}];
e38 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p8}]}];
e812 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p8, p12}]}];
```

```
e127 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p12, p7}]}];
e72 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p7, p2}]}];

e34 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p3, p4}]}];
e49 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p9}]}];
e913 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p9, p13}]}];
e138 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p13, p8}]}];
e83 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p8, p3}]}];

e40 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p4, p0}]}];
e05 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p0, p5}]}];
e514 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p5, p14}]}];
e149 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p14, p9}]}];
e94 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01], Line[{p9, p4}]}];

e1516 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p15, p16}]}];
e1611 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p16, p11}]}];
e116 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p11, p6}]}];
e610 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p6, p10}]}];
e1015 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p10, p15}]}];

e1617 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p16, p17}]}];
e1712 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p17, p12}]}];
e127 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p12, p7}]}];
e711 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p7, p11}]}];
e1116 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p11, p16}]}];

e1718 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p17, p18}]}];
e1813 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p18, p13}]}];
```

```
e138 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p13, p8}]}];
e812 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p8, p12}]}];
e1217 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p12, p17}]}];

e1819 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p18, p19}]}];
e1914 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p19, p14}]}];
e149 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p14, p9}]}];
e913 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p9, p13}]}];
e1318 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p13, p18}]}];

e1516 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p15, p16}]}];
e1617 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p16, p17}]}];
e1718 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p17, p18}]}];
e1819 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p18, p19}]}];
e1915 = Graphics3D[{RGBColor[0, 0, 0], Thickness[0.01],
   Line[{p19, p15}]}];

Show[{v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13,
  v14, v15, v16, v17, v18, v19, vc,
  e01, e12, e23, e34, e40,
  e01, e16, e610, e510, e05,
  e12, e27, e711, e116, e61,
  e23, e38, e812, e127, e72,
  e34, e49, e913, e138, e83,
  e40, e05, e514, e149, e94,
  e1516, e1611, e116, e610, e1015,
  e1617, e1712, e127, e711, e1116,
  e1718, e1813, e138, e812, e1217,
  e1819, e1914, e149, e913, e1318,
  e1516, e1617, e1718, e1819, e1915}, AxesLabel → {"x", "y", "z"},
 AspectRatio → 1, Axes → True, PlotRange → All, ImageSize → Automatic,
 Boxed → True]
]
```

Function Test - *Dodecahedron*

Below we test the functions related to creating, rotating, and visualizing the dodecahedron.

■ Test *createDodecahedron*

We create a dodecahedron with edgelength=5 centered at (5, 5, 5).

In[135]:=

```
dodecahedronPoints = createDodecahedron[5, {5, 5, 5}]
```

Out[135]=

```
{{4.25351, 0., 5.56739}, {1.31441, 4.04533, 5.56739},
 {-3.44116, 2.50015, 5.56739}, {-3.44116, -2.50015, 5.56739},
 {1.31441, -4.04533, 5.56739}, {6.88196, 0., 1.31405},
 {2.12664, 6.54514, 1.31405}, {-5.56762, 4.04512, 1.31405},
 {-5.56762, -4.04512, 1.31405}, {2.12664, -6.54514, 1.31405},
 {5.56762, 4.04512, -1.31405}, {-2.12664, 6.54514, -1.31405},
 {-6.88196, 0., -1.31405}, {-2.12664, -6.54514, -1.31405},
 {5.56762, -4.04512, -1.31405}, {3.44116, 2.50015, -5.56739},
 {-1.31441, 4.04533, -5.56739}, {-4.25351, 0., -5.56739},
 {-1.31441, -4.04533, -5.56739}, {3.44116, -2.50015, -5.56739}}
```

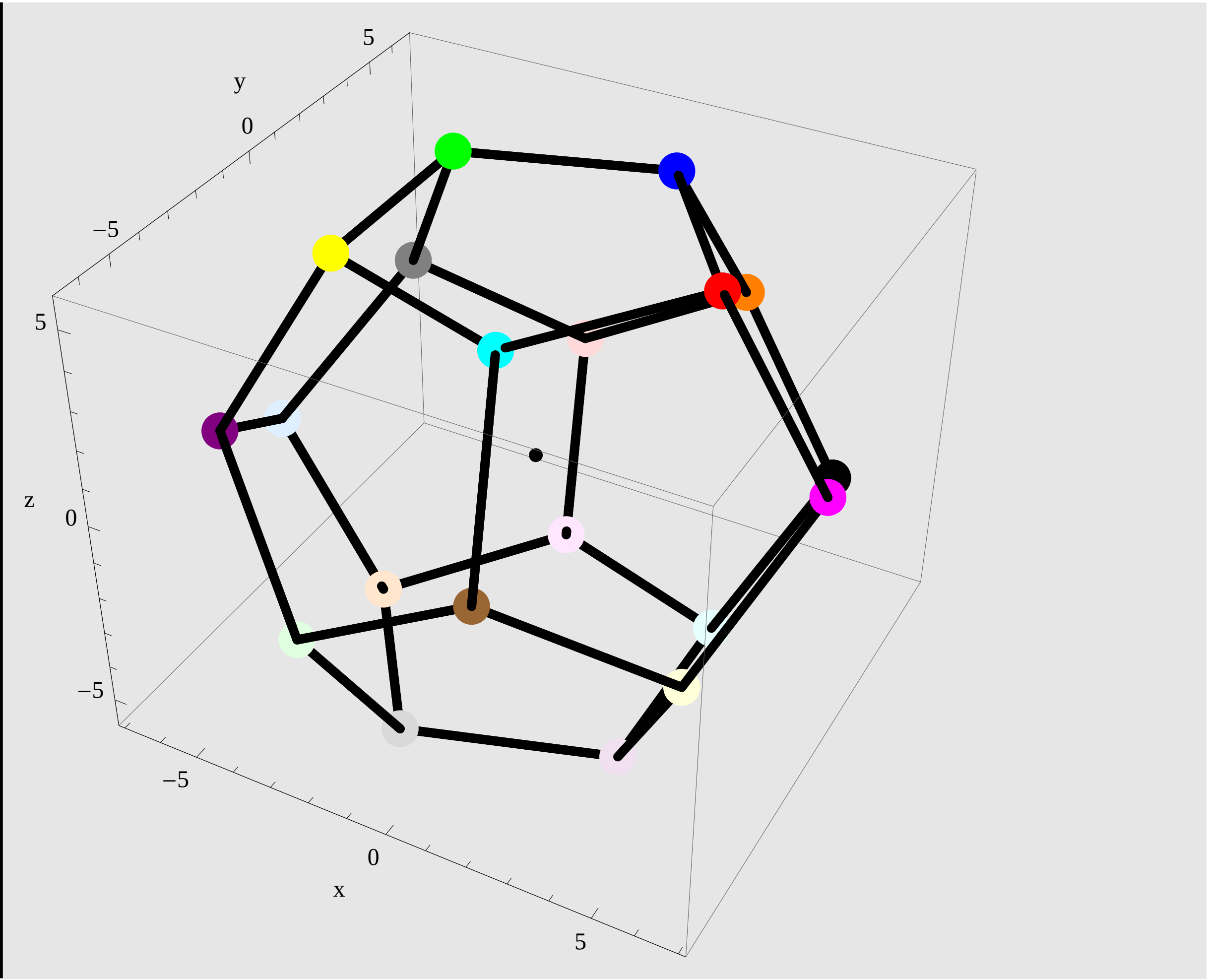
■ Test *drawDodecahedron*

Below we visualize the dodecahedron created above.

In[136]:=

```
drawDodecahedron[dodecahedronPoints]
```

Out[136]=



■ Test *rotateDodecahedron*

Rotating the dodecahedron an angle ϕ counterclockwise about the X-axis (Roll)

We rotate the dodecahedron an angle $\phi = 45^\circ$ counterclockwise about the X-axis.

In[137]:=

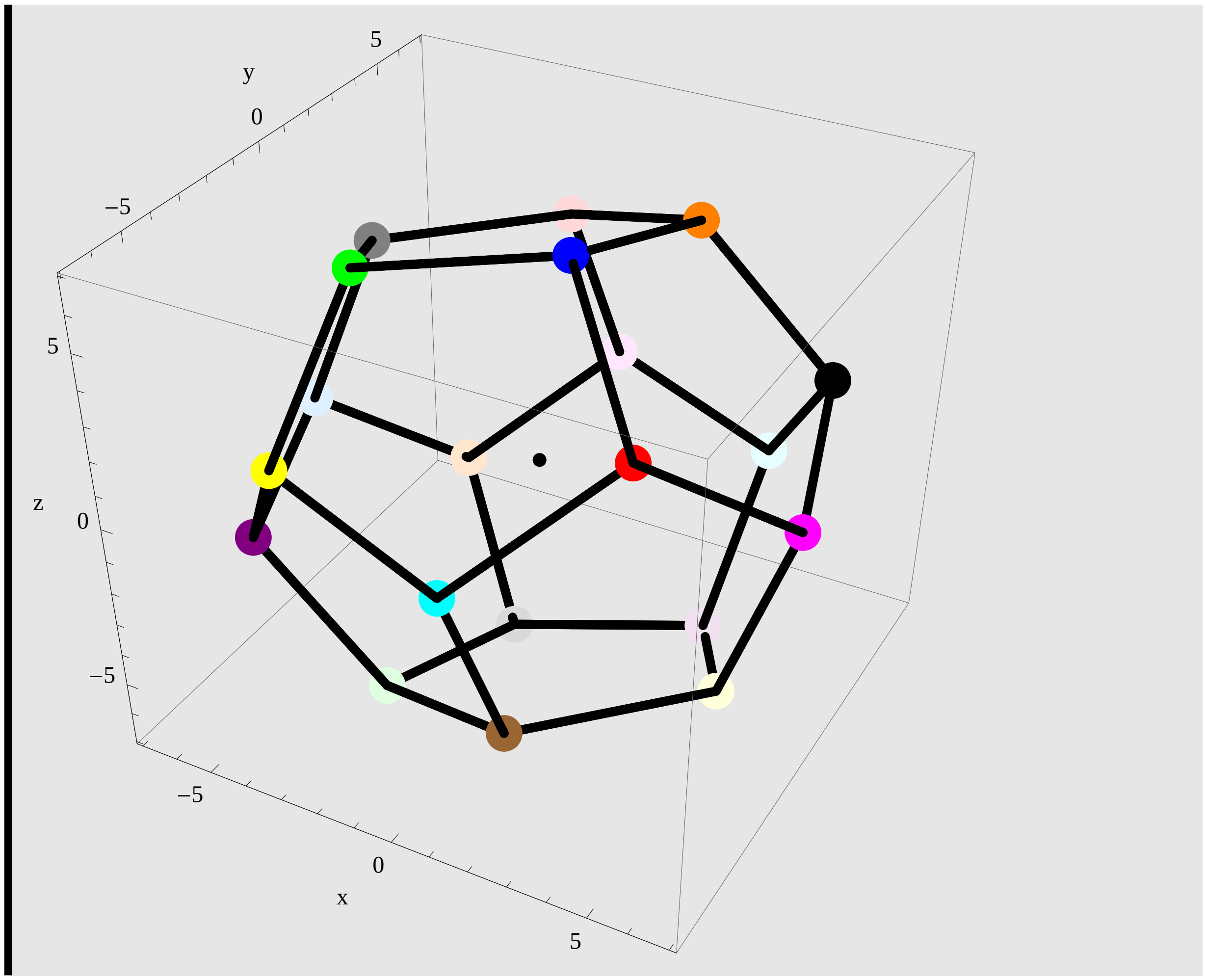
```
dodecahedronPointsRotatedAboutX =
  rotateDodecahedron[dodecahedronPoints, {45, 0, 0}];
```

The dodecahedron rotated an angle $\phi = 45^\circ$ conterclockwise about the X-axis is displayed below.

In[138]:=

```
drawDodecahedron[dodecahedronPointsRotatedAboutX]
```

Out[138]=



Rotating the dodecahedron an angle θ counterclockwise about the Y-axis (Pitch)

We rotate the dodecahedron an angle $\theta = 45^{\circ}$ counterclockwise about the Y-axis.

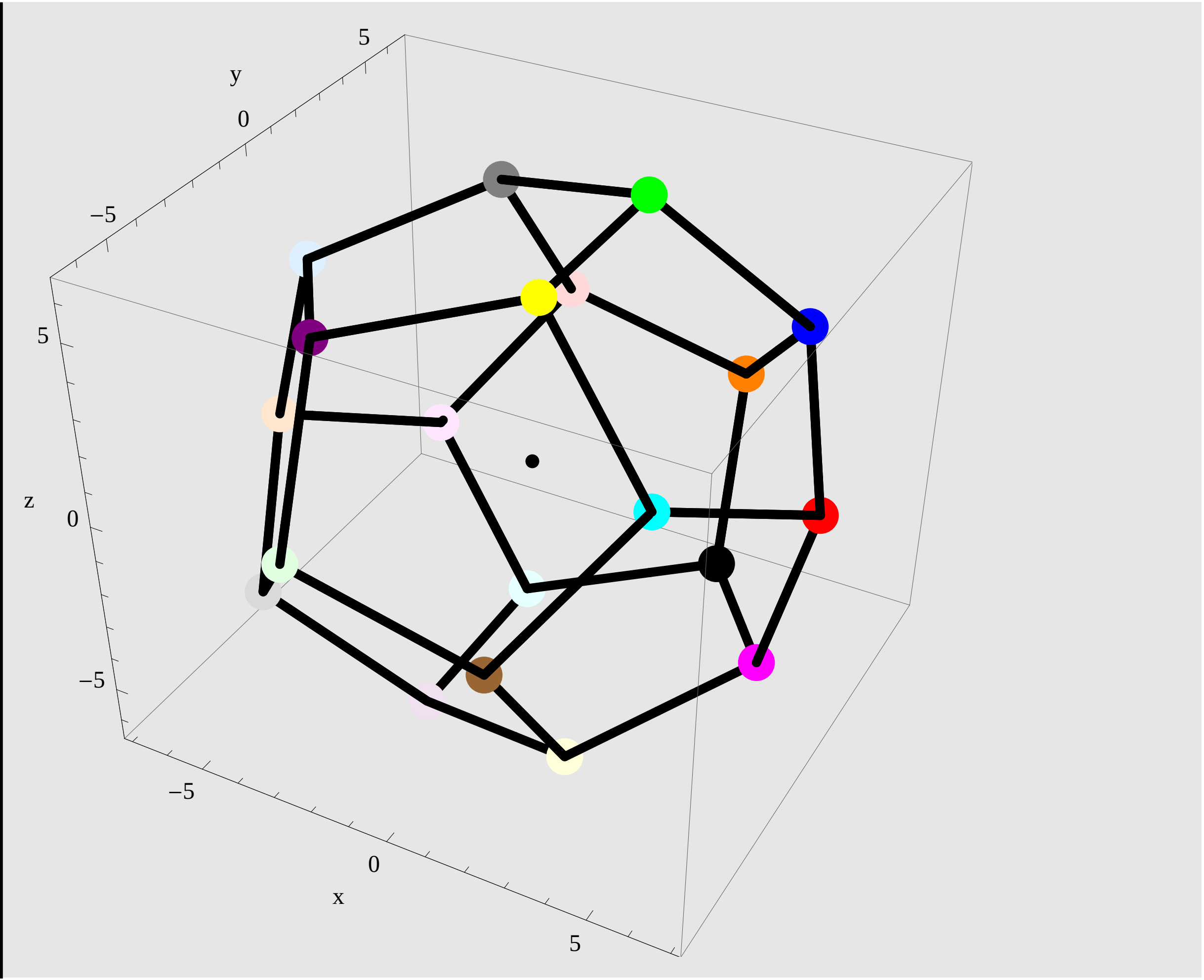
In[139]:=

```
dodecahedronPointsRotatedAboutY =
  rotateDodecahedron[dodecahedronPoints, {0, 45, 0}];
```

The dodecahedron rotated an angle $\theta = 45^{\circ}$ conterclockwise about the Y-axis is displayed below.

In[140]:=
```
drawDodecahedron[dodecahedronPointsRotatedAboutY]
```

Out[140]=



Rotating the dodecahedron an angle ψ counterclockwise about the Z-axis (Yaw)

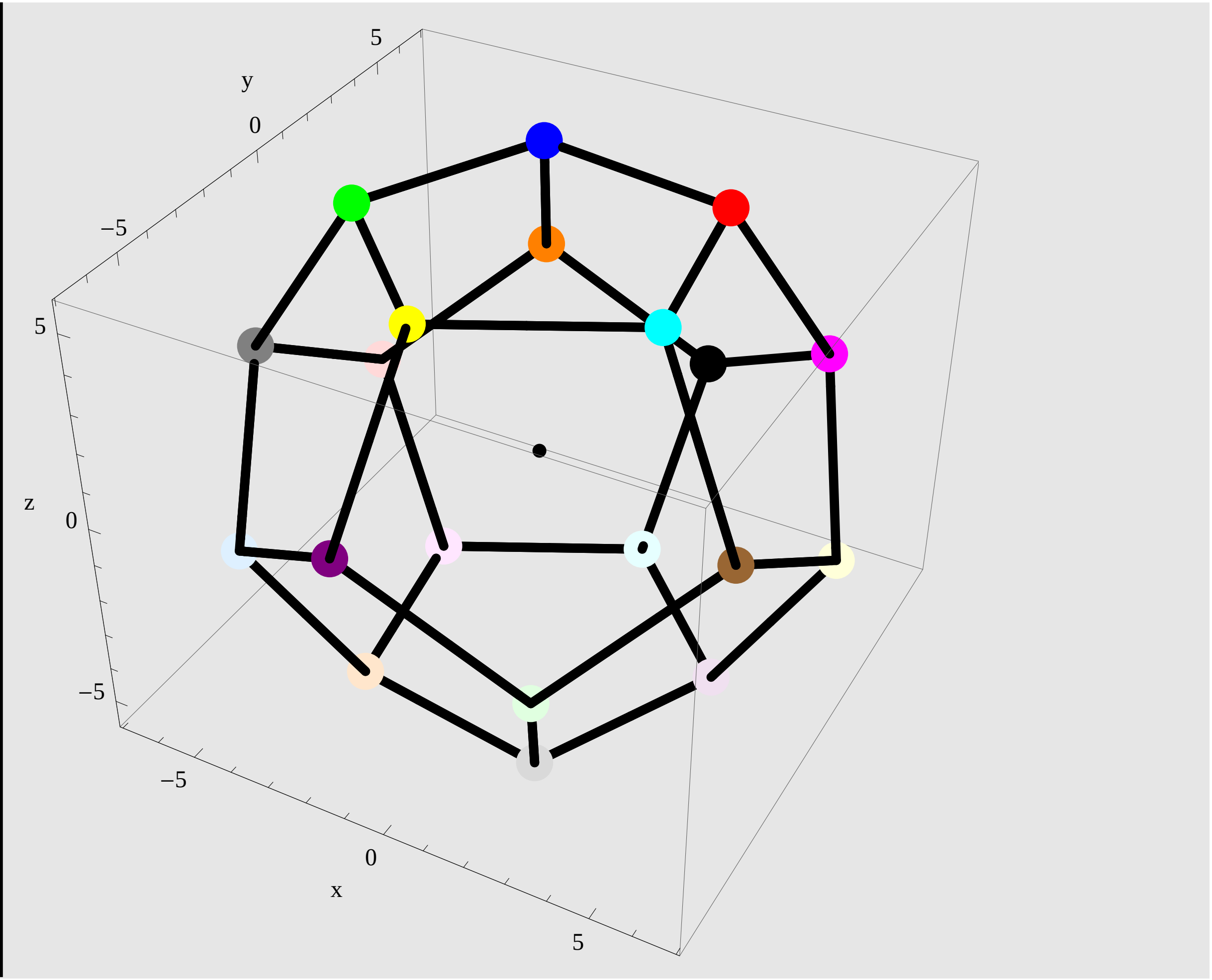
We rotate the dodecahedron an angle $\psi = 45^{\circ}$ counterclockwise about the Z-axis.

In[141]:=
```
dodecahedronPointsRotatedAboutZ =
  rotateDodecahedron[dodecahedronPoints, {0, 0, 45}];
```

The dodecahedron rotated an angle $\psi = 45^{\circ}$ conterclockwise about the Z-axis is displayed below.

In[142]:=
```
drawDodecahedron[dodecahedronPointsRotatedAboutZ]
```

Out[142]=



■ Rotating in steps

Step 1: rotating the dodecahedron an angle ϕ counterclockwise about the X-axis (Roll)

First we rotate the dodecahedron about the X-axis and stores the result.

In[143]:=
```
rx = rotateDodecahedron[dodecahedronPoints, {45, 0, 0}];
```

Step 2: rotating the dodecahedron from step 1 an angle θ counterclockwise about the Y-axis (Pitch)

We use the result from step 1 and rotate the dodecahedron a second time - this time about the Y-axis

In[144]:=
```
ry = rotateDodecahedron[rx, {0, 45, 0}];
```

Step 3: rotating the dodecahedron from step 2 an angle ψ counterclockwise about the Z-axis (Yaw)

We use the result from step 2 and rotate the dodecahedron a third time - this time about the Z-axis

In[145]:=

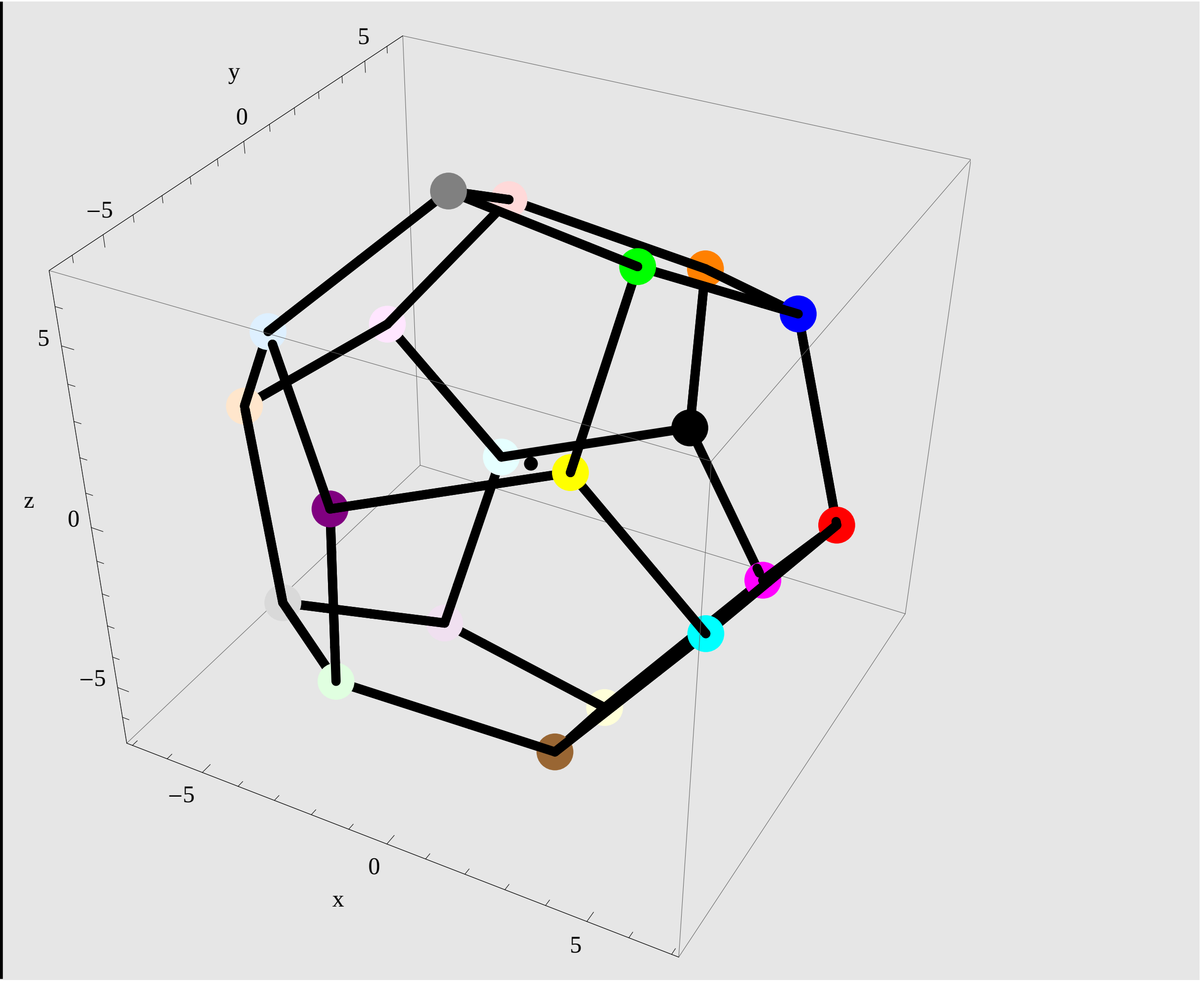
```
rz = rotateDodecahedron[ry, {0, 0, 45}];
```

The result of the three consecutive rotations is shown below

In[146]:=

```
drawDodecahedron[rz]
```

Out[146]=

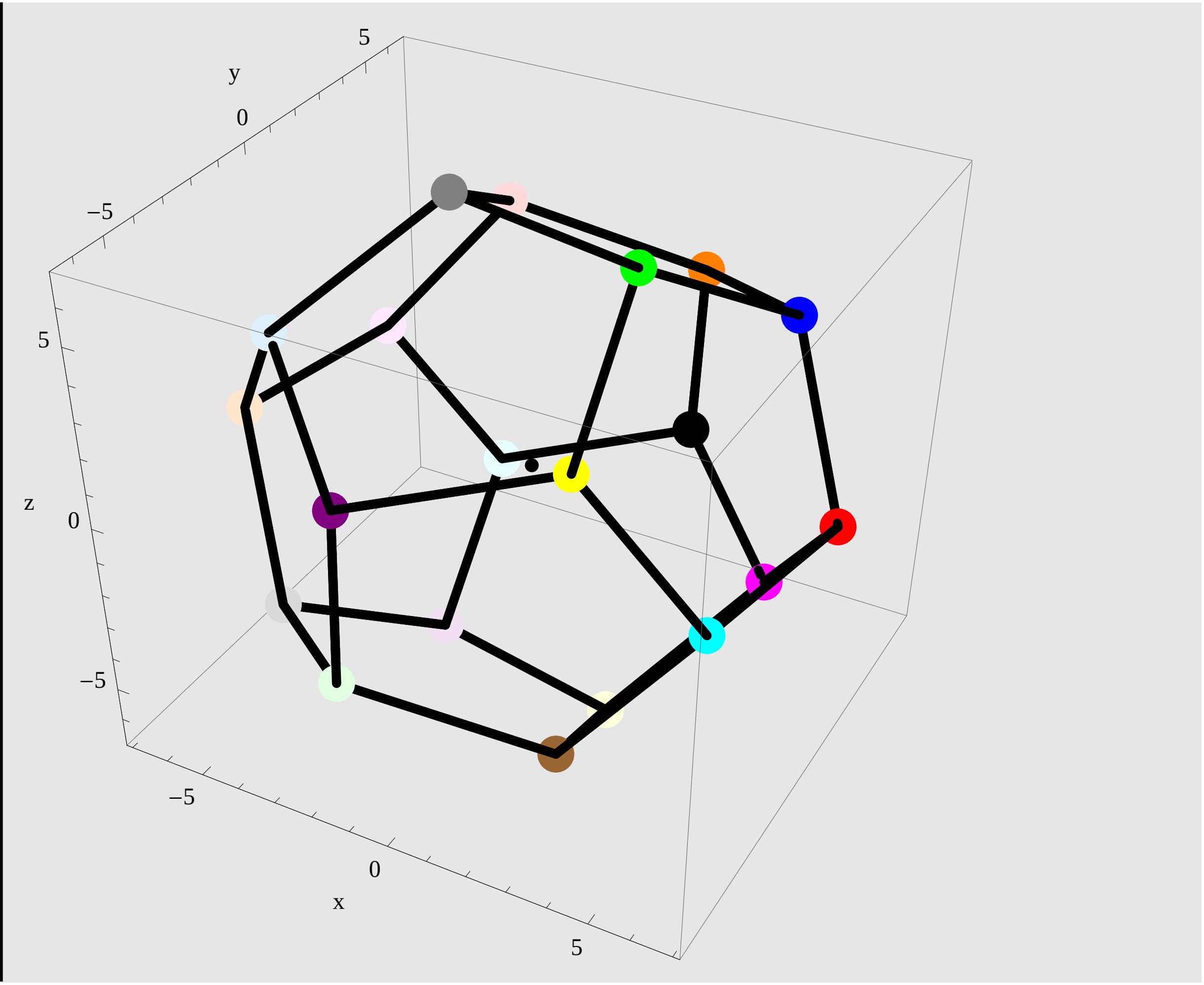


Rotating the dodecahedron an angle ϕ counterclockwise about the X-axis, then an angle θ counterclockwise about the Y-axis, and finally an angle ψ counterclockwise about the Z-axis

Instead of applying the rotations separately we can calculate the resulting rotation directly. This is shown below

In[147]:=
```
drawDodecahedron[rotateDodecahedron[dodecahedronPoints, {45, 45, 45}]]
```

Out[147]=



Output

In[148]:=
```
Export["rotating_platonic_solids.pdf", EvaluationNotebook[]]
```