POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2024

# Dipartimento di Elettronica, Informazione e Bioingegneria

*Computer Graphics*

Milano, 2024

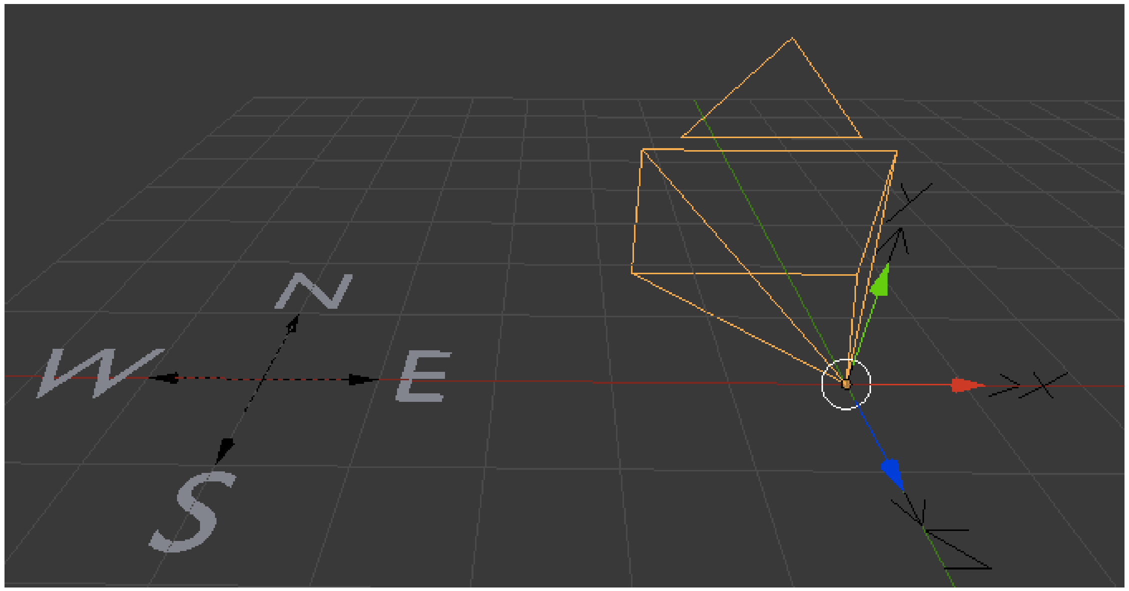
# Computer Graphics

- View and World Matrix

## Axis and orientation

We have defined the *world coordinates* as a reference system to globally specify the positions of an object in the 3D space.

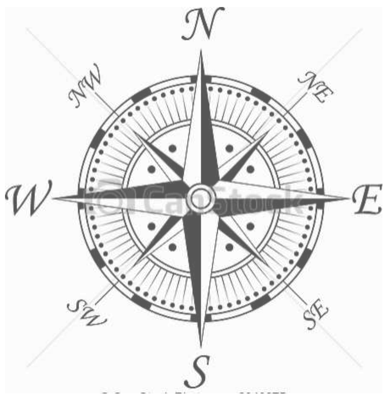
We have chosen a special system, called *y-up*, where the *y-axis* is parallel to the vertical side of the screen, and the *z-axis* is oriented toward the viewer, out of the screen.

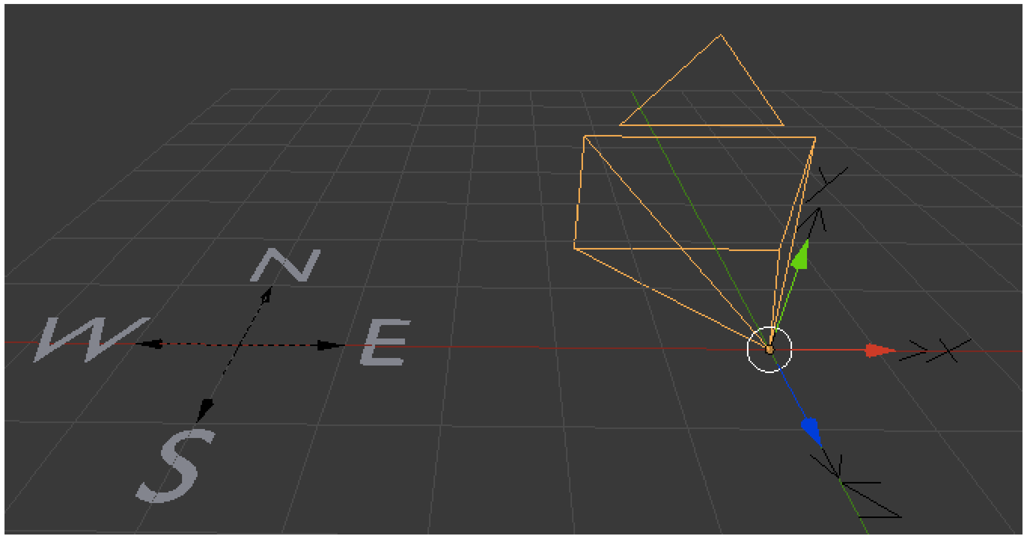


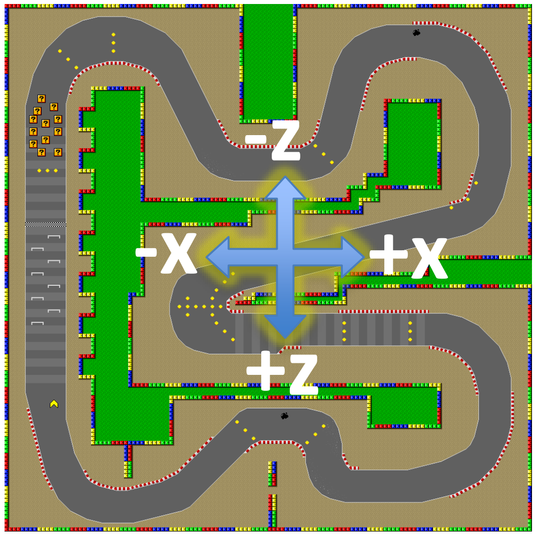
# Axis and orientation

To simplify the presentation of the positions and the directions, we will set the origin in the middle of the 3D world, and use a compass to define directions.

In particular, we will call the *negative z-axis* the *North* direction in the 3D world, and that the *positive x-axis* the *East*.







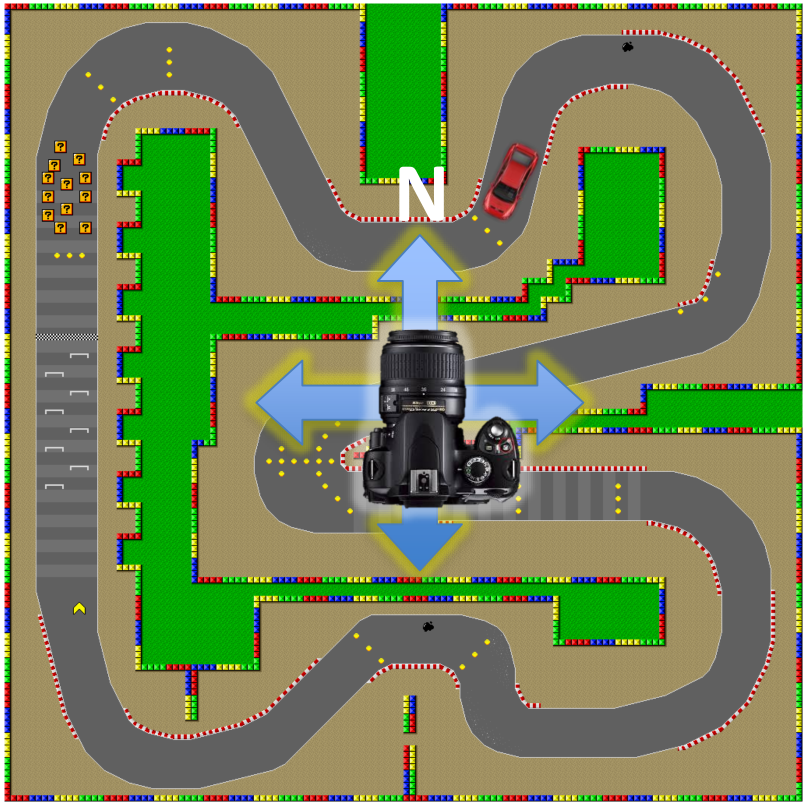
## The View Matrix

The projection matrices that we have seen, assumes that the projection plane is parallel to the *xy-plane*.

For parallel projections, we also assume that the projection rays are oriented along the negative *z-axis*.

For perspective, we consider the center of projection in the *origin*.

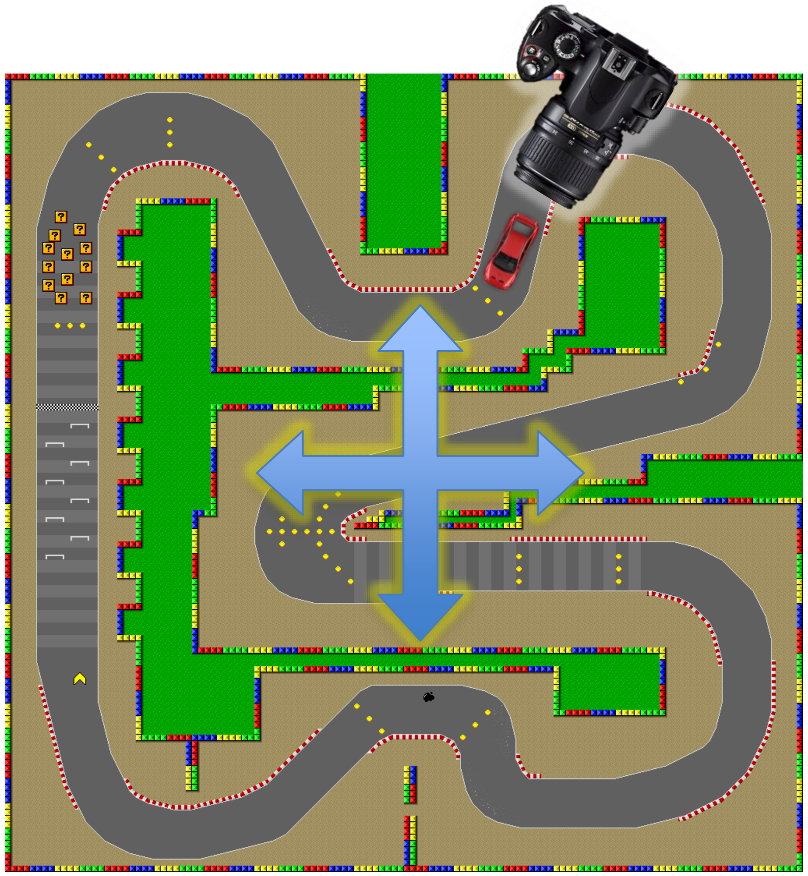
In both cases, it corresponds to a camera looking *North*.



## The View Matrix

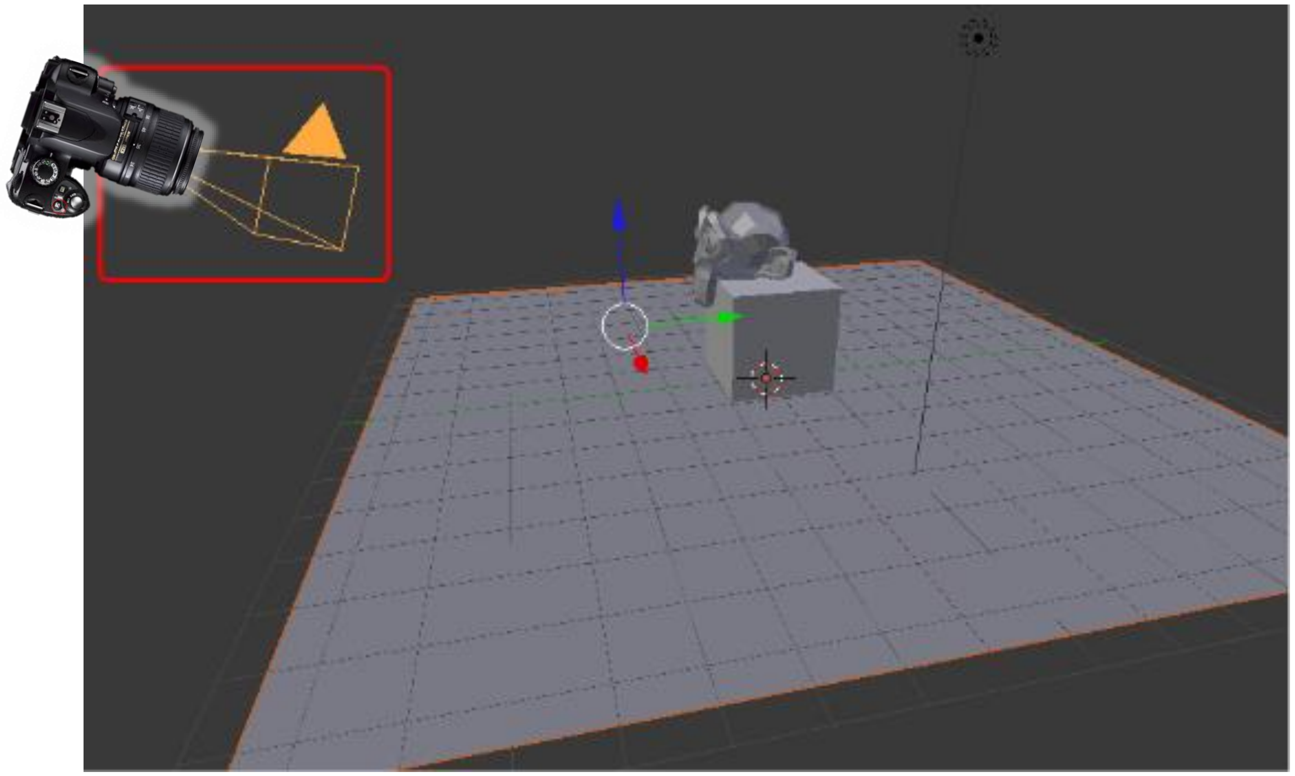
In actual applications, we are interested in generating a 2D view for a plane arbitrarily positioned in the space.

This is achieved by adding a set of additional transformations before the projection.



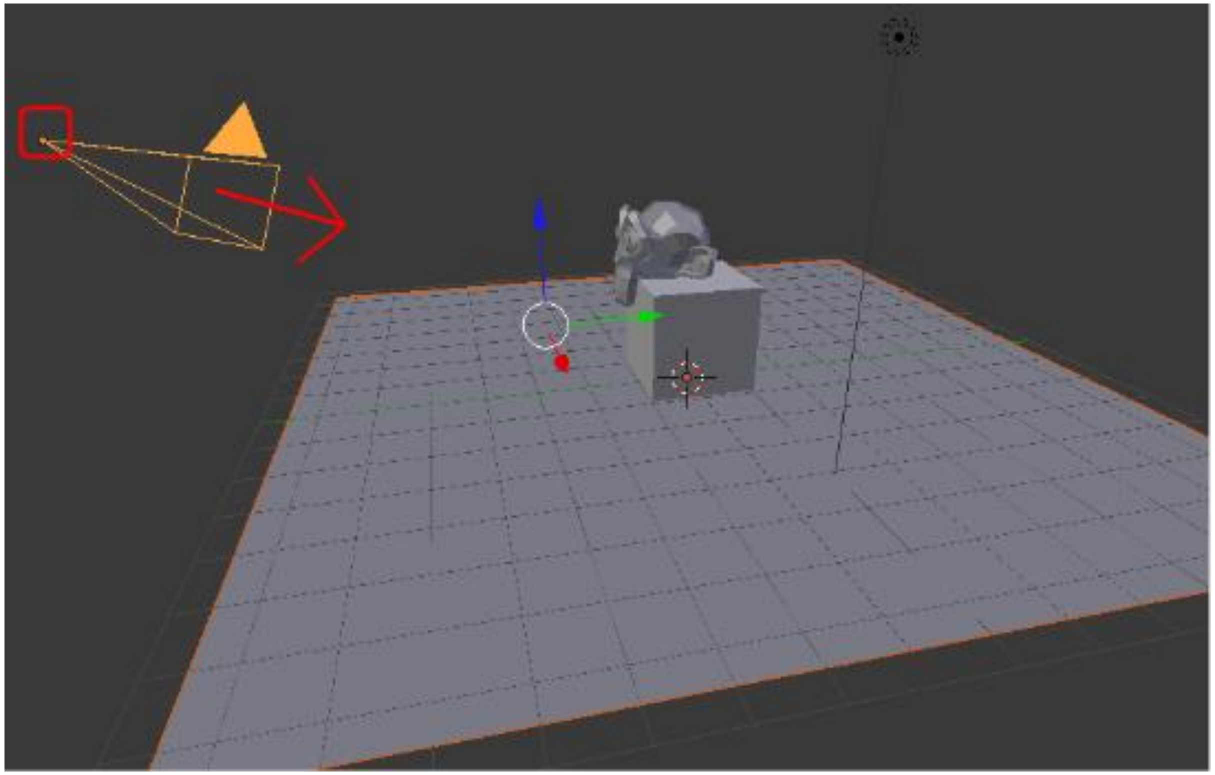
## The View Matrix

We will focus on perspective, but the same reasoning applies also to parallel projection.  The projection plane can be seen as the *sensor* of a *camera* that looks at the scene from the center of projection.



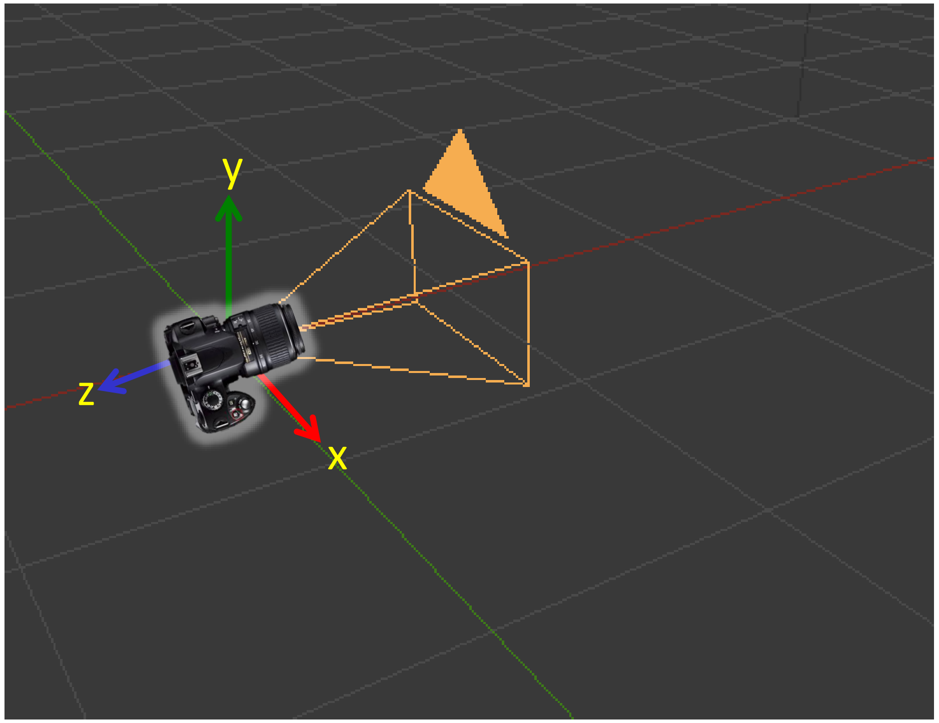
## The View Matrix

The camera is characterized by its position, the direction where it is aiming, and its angle around this direction.



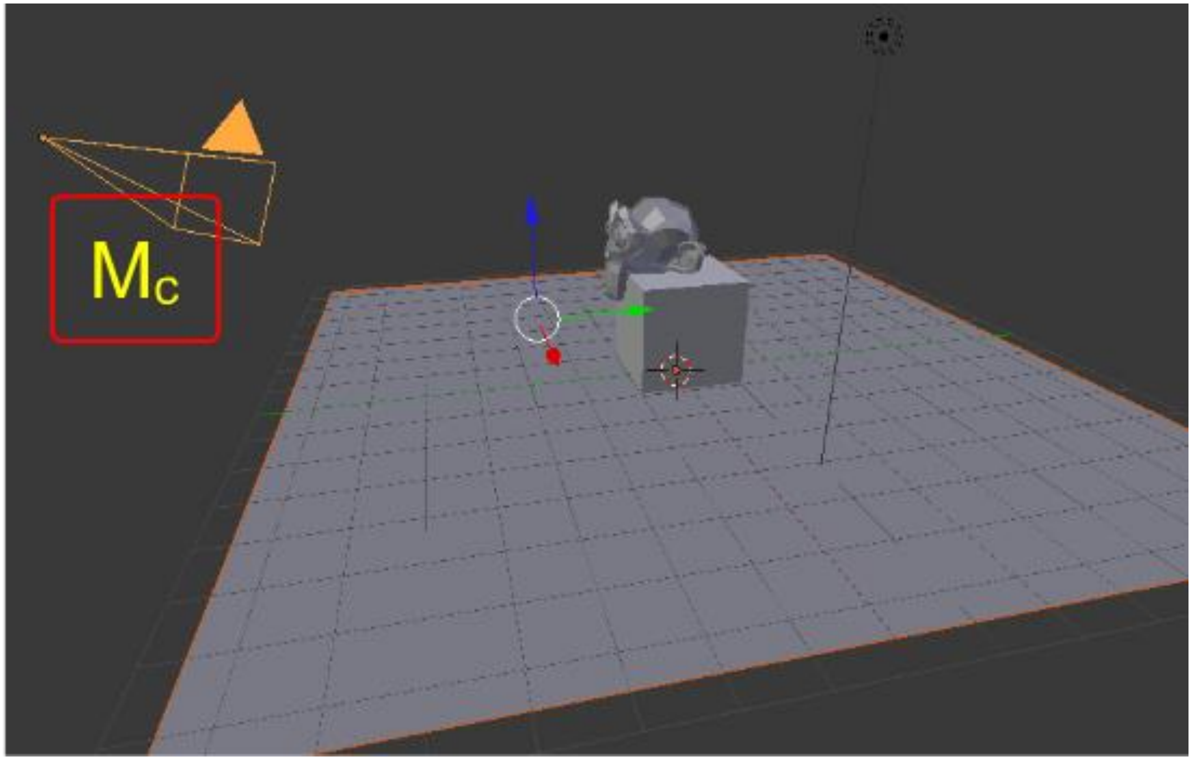
## The View Matrix

The projection matrices that we have seen, compute the view of a *camera* initially positioned in the origin, and aiming along the negative *z-axis*. We can consider this camera as a 3D object.



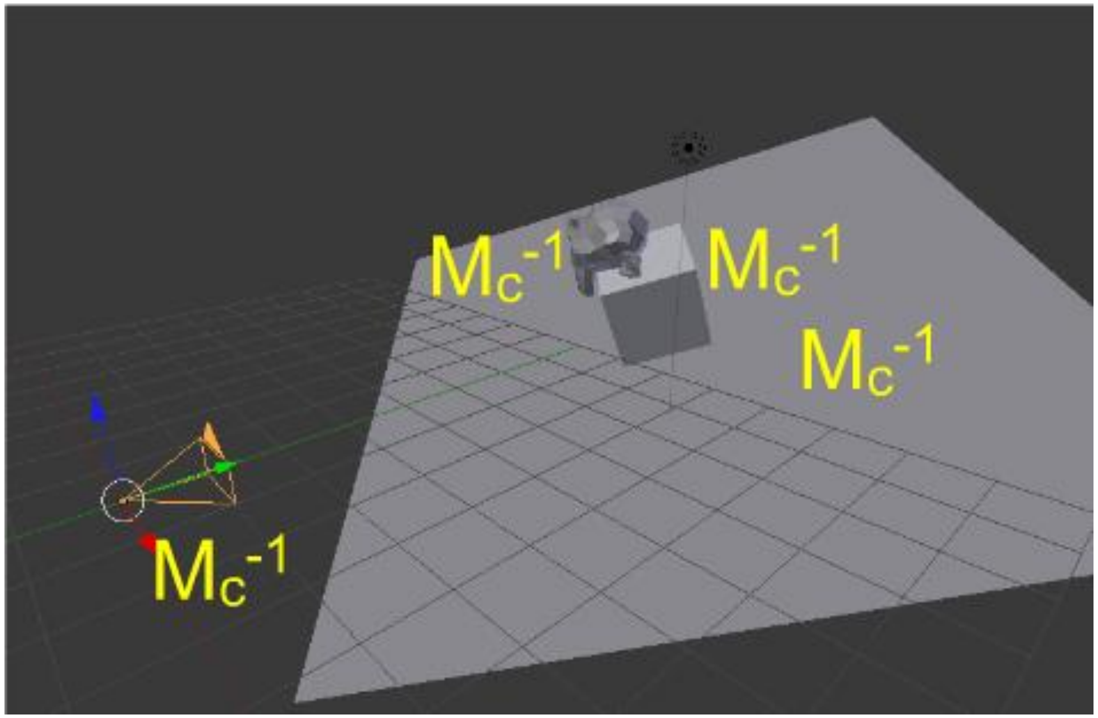
## The View Matrix

We can then define a transformation matrix $M_c$ that moves this camera object to its target position, orienting the negative *z-axis* along the desired direction. We will call this matrix the *Camera Matrix*.



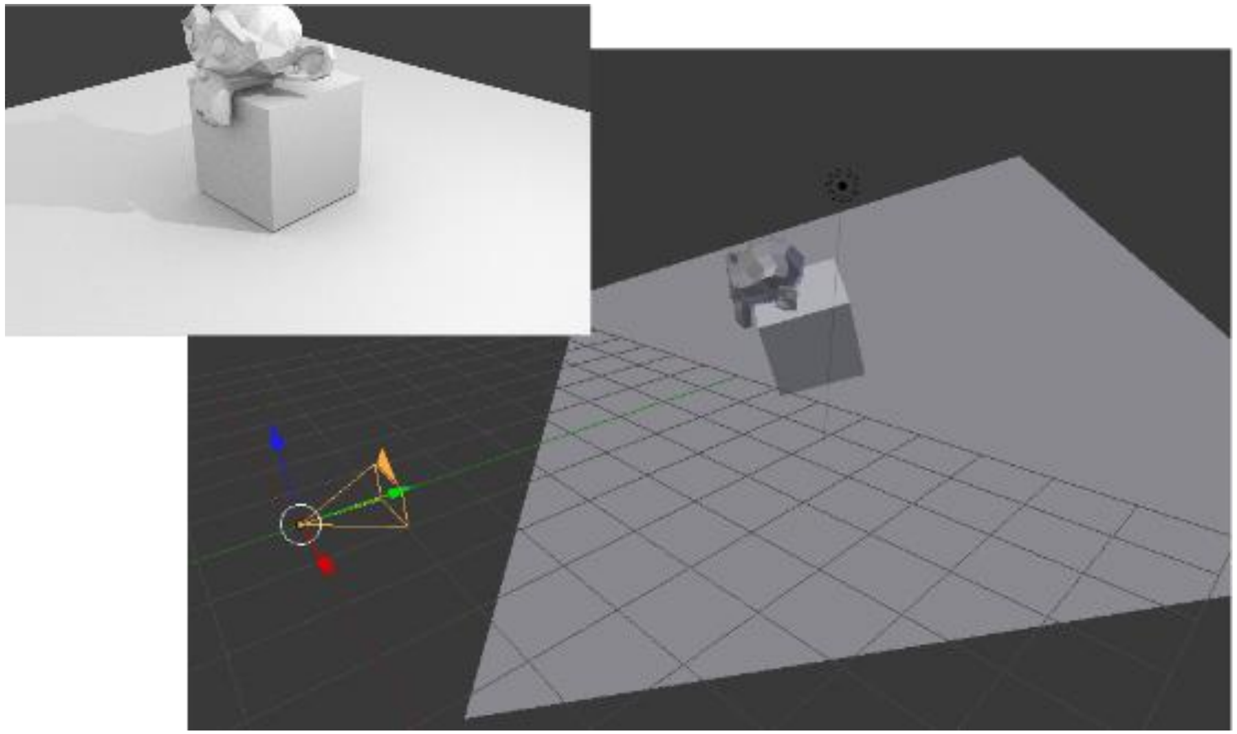
## The View Matrix

If we apply the inverse of $M_c$ to all the objects in the scene, we obtain a new 3D world where the projection plane is placed exactly as required.



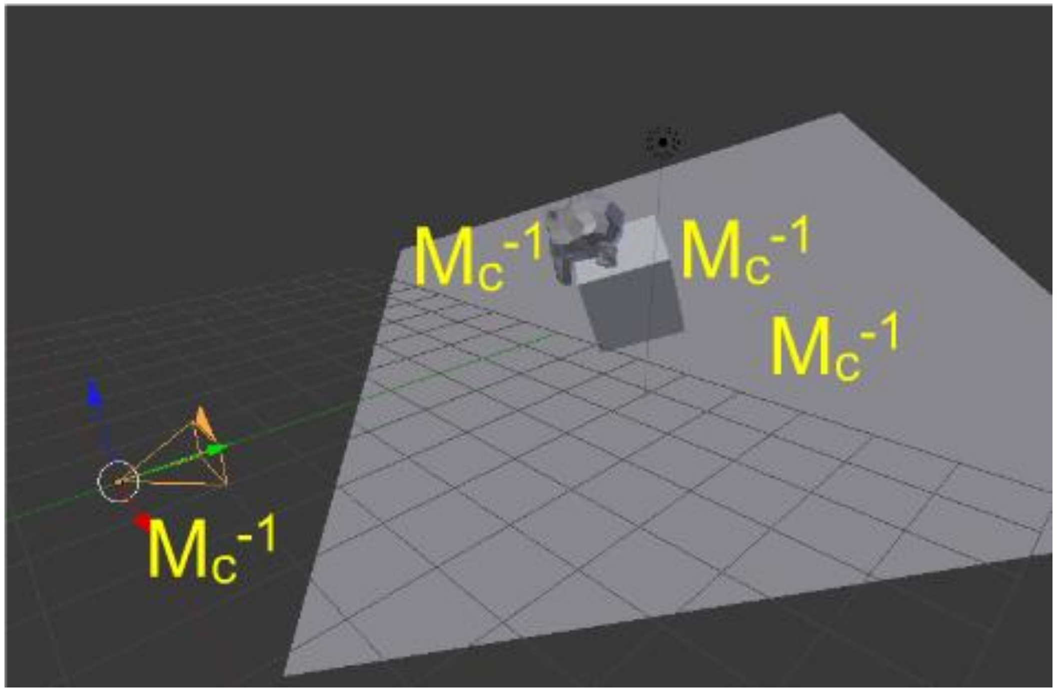
## The View Matrix

In this new space, the view of the 3D world as seen from the arbitrarily oriented camera can be computed by applying the projections matrices we have seen in previous lessons.



## The View Matrix

Matrix $M_c^{-1}$ is known as the *View Matrix*, and we will denote it as $M_v$ .



$$M_V = M_C^{-1}$$

## The View-Projection Matrix

The view matrix is placed before the projection matrix $M_{Prj}$ previously introduced: in this way, it allows to find the normalized screen coordinates of the points in space, as seen by the considered camera.

This combined matrix is sometimes known as the *view-projection matrix*.

$$M_{VP} = M_{Prj} \cdot M_V$$

It transforms a point from world coordinates (homogeneous coordinates), to 3D normalized screen coordinates (cartesian) as seen by a given camera characterized by matrix $M_{Prj}$, in a given location and direction in space according to matrix $M_V$.

## Creating a View Matrix

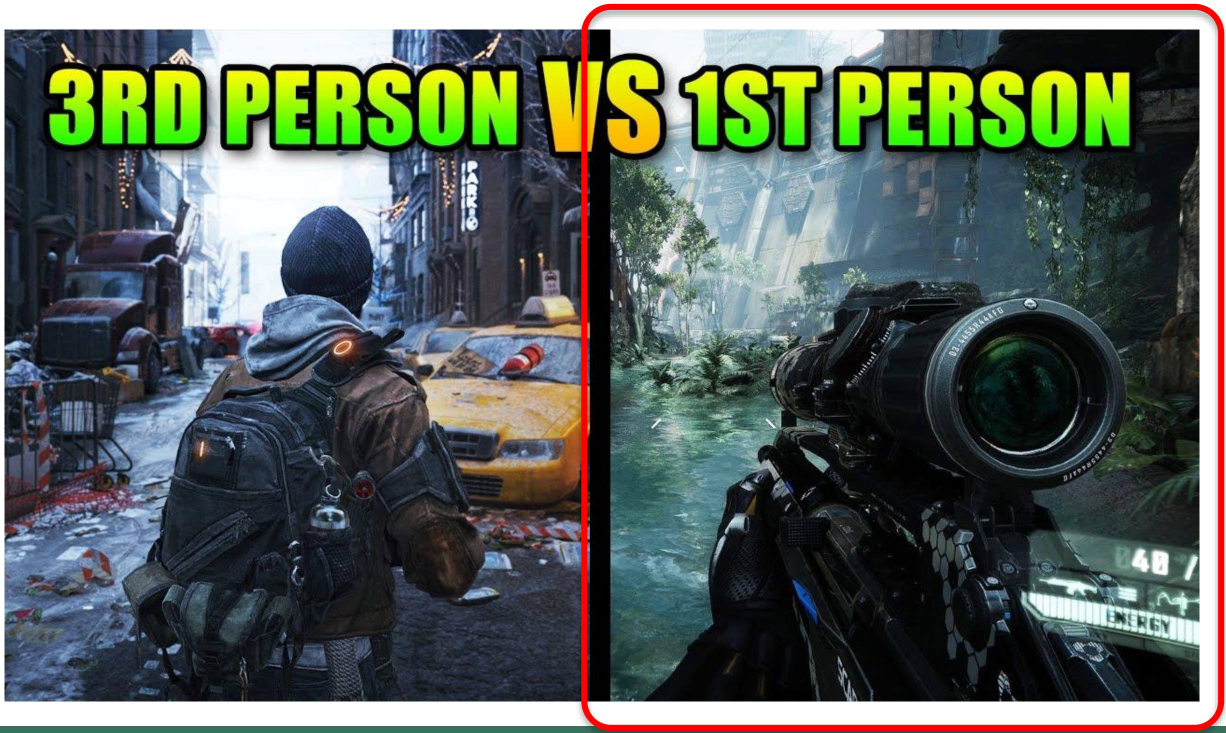
Several techniques exist to create a view matrix in a user-friendly way. The two most popular are:

- The *look-in-direction* technique
- The *look-at* technique

## View Matrix: look-in-direction

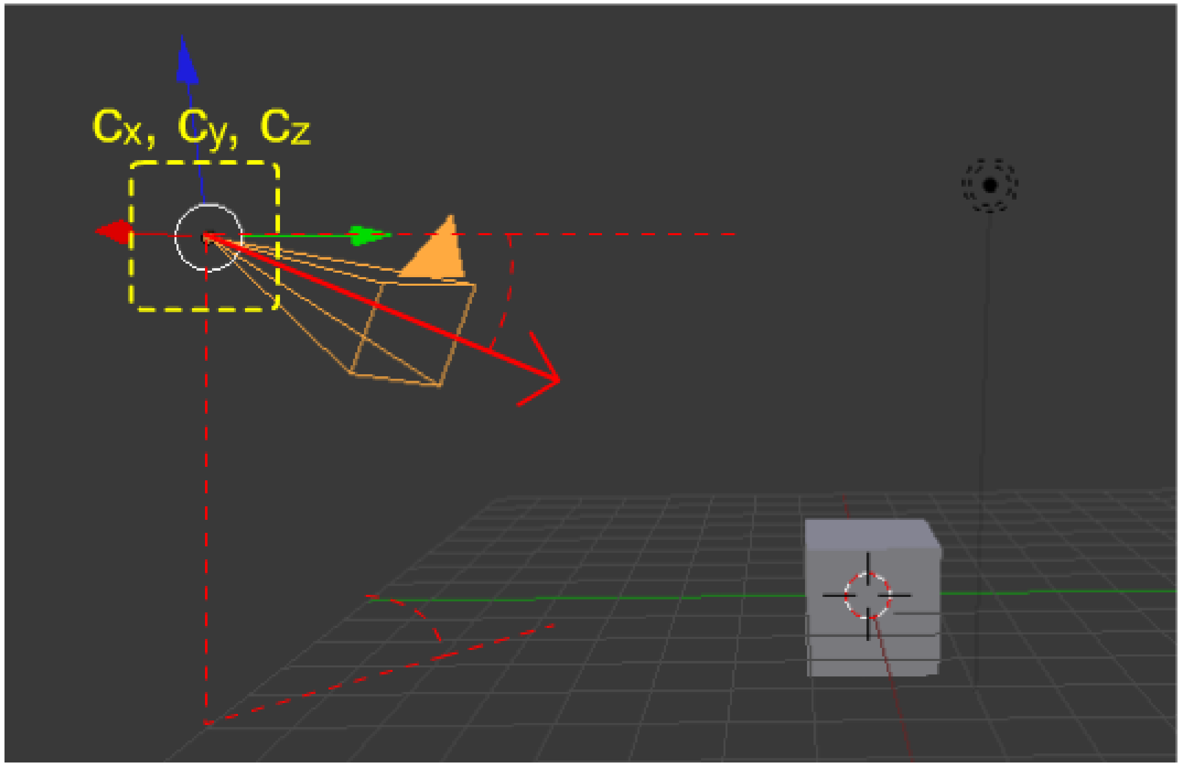
The *Look-in-direction* model is generally used for *first person applications*.

In particular, the user directly controls the camera position and the view direction.



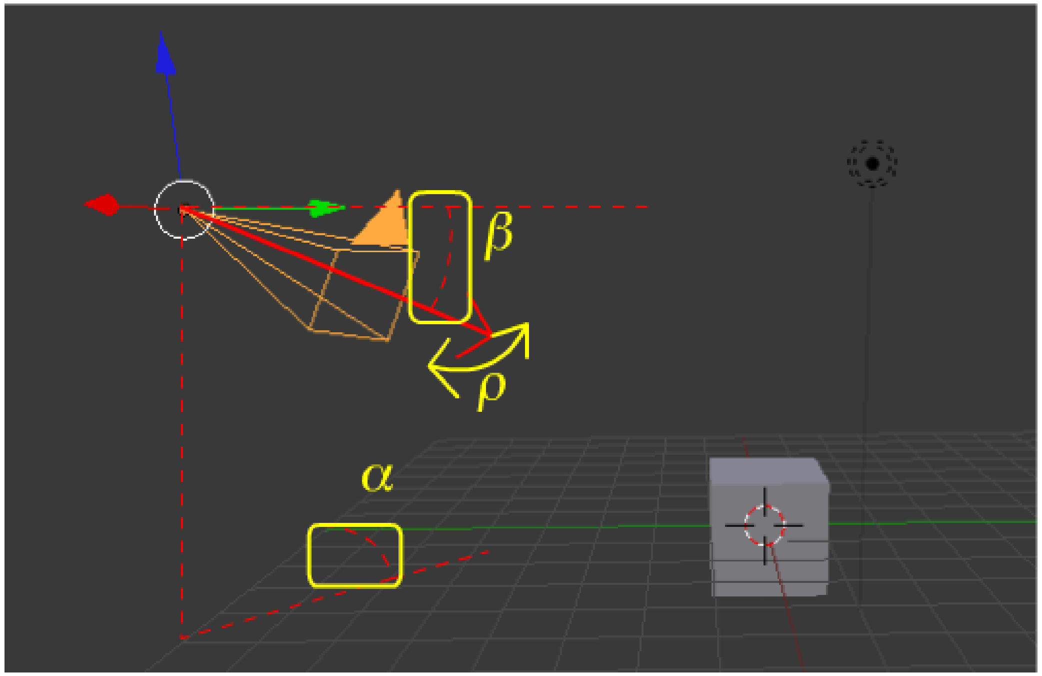
## View Matrix: look-in-direction

In the *Look-in-direction* model, the position *($c_x$, $c_y$, $c_z$)* of the camera is given in world coordinates.



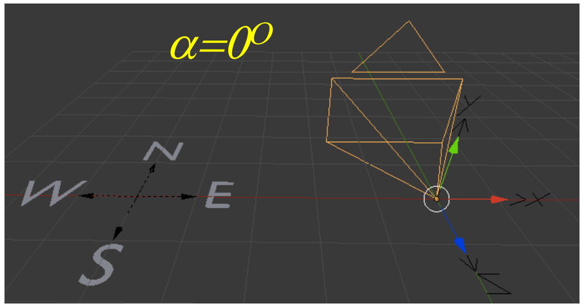
# View Matrix: look-in-direction

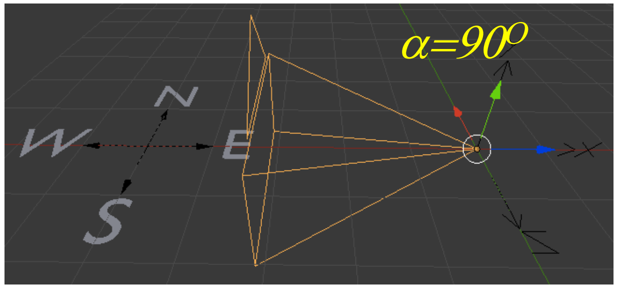
The direction where the camera is looking is specified with three angles: the “compass” direction (angle $\alpha$), the elevation (angle $\beta$), and the roll over the viewing direction (angle $\rho$). This last parameter however is rarely used.

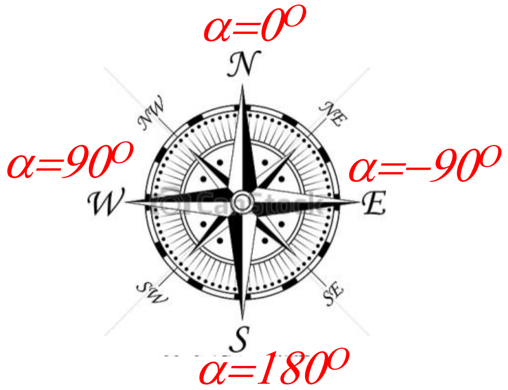


# View Matrix: look-in-direction

In particular, with $\alpha=0^O$ the camera looks *North*, while with $\alpha=90^O$ the camera looks *West*. *South* is $\alpha=180^O$ and *East* is $\alpha=-90^O=270^O$.

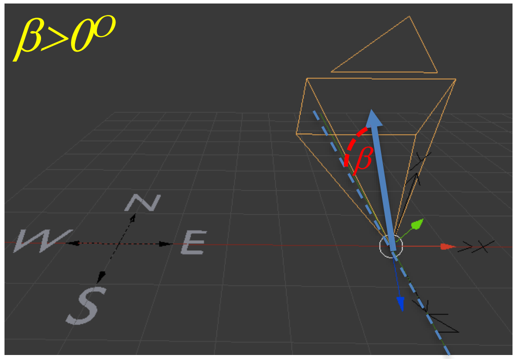


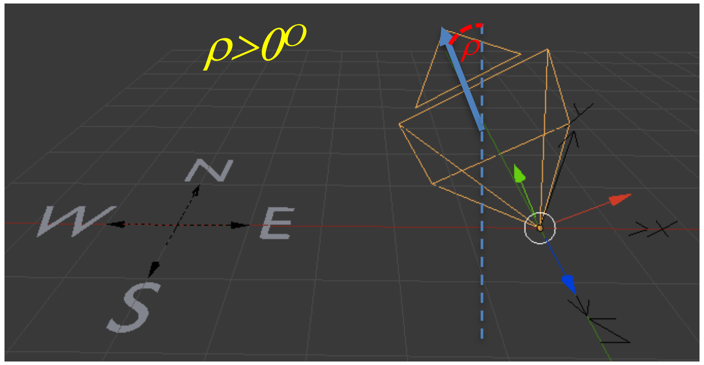




A positive angle $\beta>0^O$ makes the camera look up.

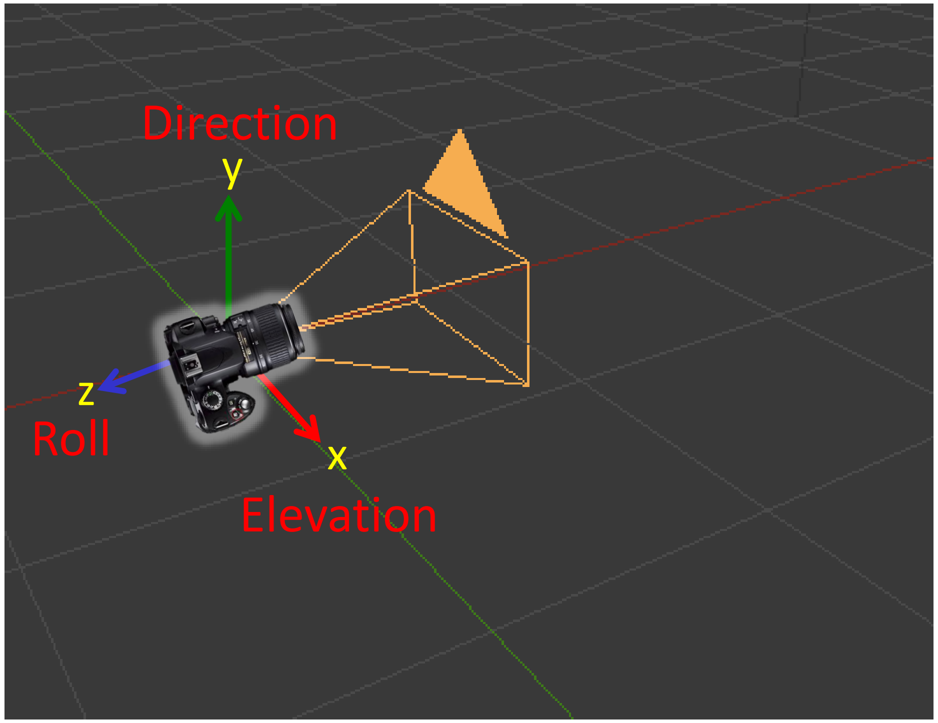
A positive angle $\rho>0^O$ turns the camera counterclockwise.





## The View Matrix

Considering the camera object, roll corresponds to a rotation around the *z-axis*, elevation (also known as *pitch*) along the *x-axis*, and direction *(also known as yaw)* around the *y-axis*.



## View Matrix: look-in-direction

Rotations must be performed in a specific order, that will be discussed later. In particular, the roll must be performed first, then the elevation, and finally the direction. Translation is performed after the rotations. The *Camera Matrix* is then composed in this way:

$$M_C = T(c_x, c_y, c_z) \times R_y(a) \times R_x(b) \times R_z(r)$$

The *View Matrix*, is the inverse of the *Camera Matrix*. Remembering the rules and pattern for inverting a chain of transformations, we have:

$$M_V = \left(M_C\right)^{-1} = R_z(-r) \times R_x(-b) \times R_y(-a) \times T(-c_x, -c_y, -c_z)$$

# Look in matrix in GLM

GLM does not provide any special support to build a *Look-in-direction* matrix. However, due to its simplicity, it can be easily implemented using what we have seen in the previous lessons:

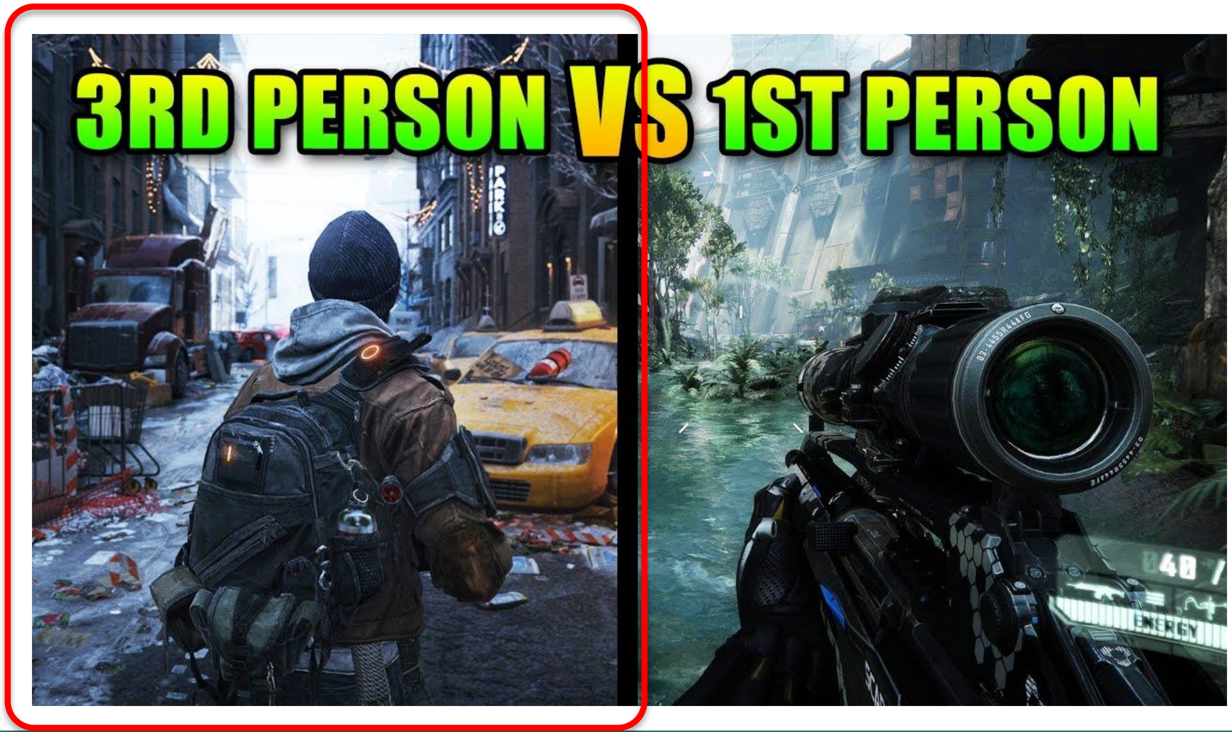
$$M_V = (M_C)^{-1} = R_z(-\rho) \cdot R_x(-\beta) \cdot R_y(-\alpha) \cdot T(-c_x, -c_y, -c_z)$$

```
glm::mat4 Mv =
    glm::rotate(glm::mat4(1.0),   -rho, glm::vec3(0,0,1)) *
    glm::rotate(glm::mat4(1.0),  -beta, glm::vec3(1,0,0)) *
    glm::rotate(glm::mat4(1.0), -alpha, glm::vec3(0,1,0)) *
    glm::translate(glm::mat4(1.0), glm::vec3(-cx, -cy ,-cz));
```

## View Matrix: look-at

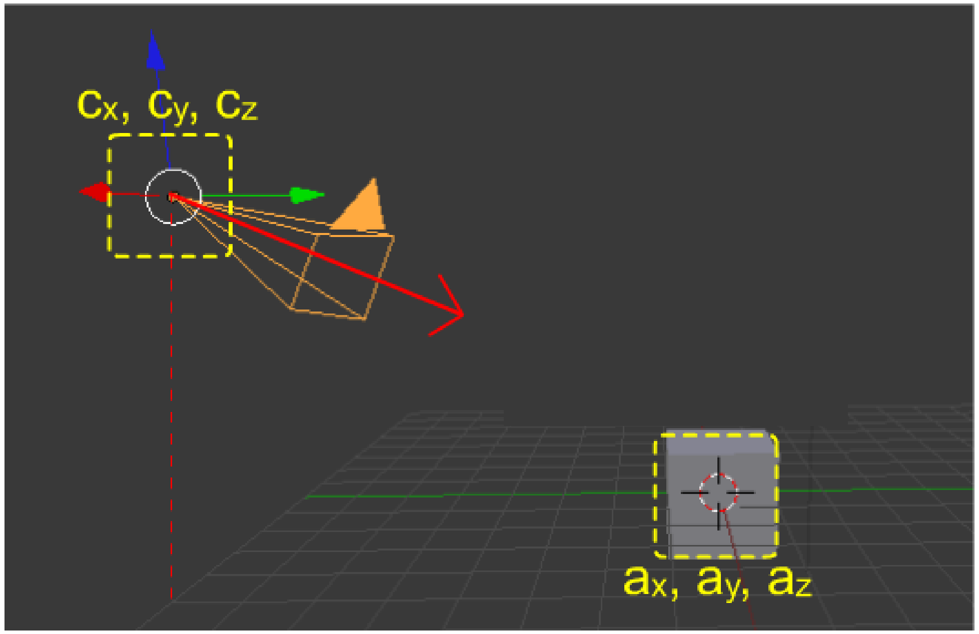
The *look-at* model is instead generally employed in *third person applications*.

In this case, the camera tracks a point (or an object) aiming at it.



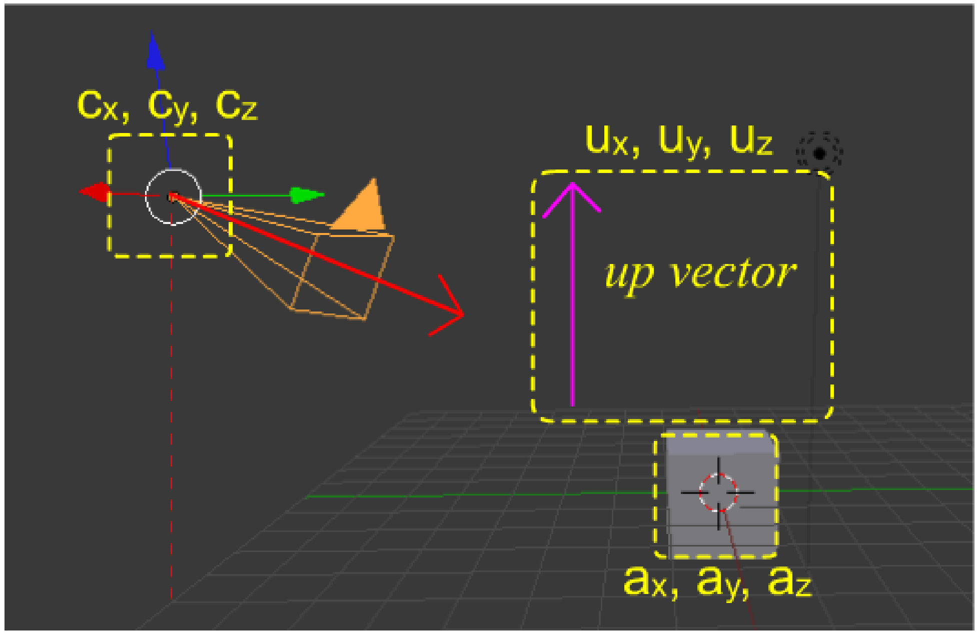
## View Matrix: look-at

In the *look-at* model, the center of the camera is positioned at *$c=(c_x, c_y, c_z)$*, and the target is a point of coordinates *$a=(a_x, a_y, a_z)$*.



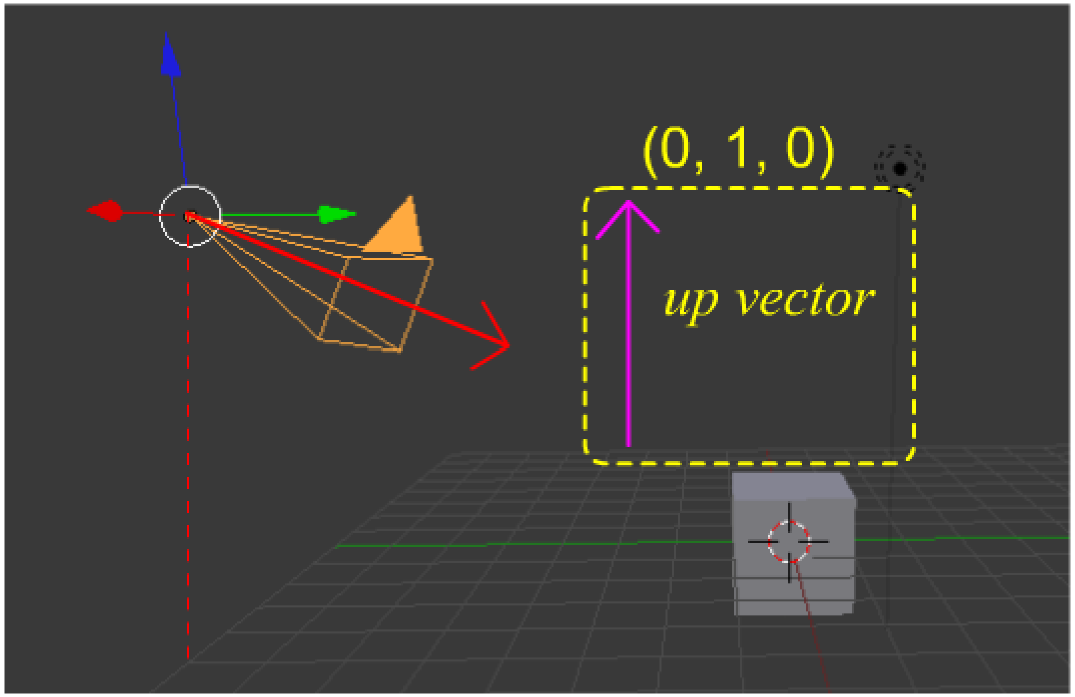
## View Matrix: look-at

The technique also requires the *up vector*: the direction $u=(u_x, u_y, u_z)$ perpendicular to "*the ground*" of the scene.



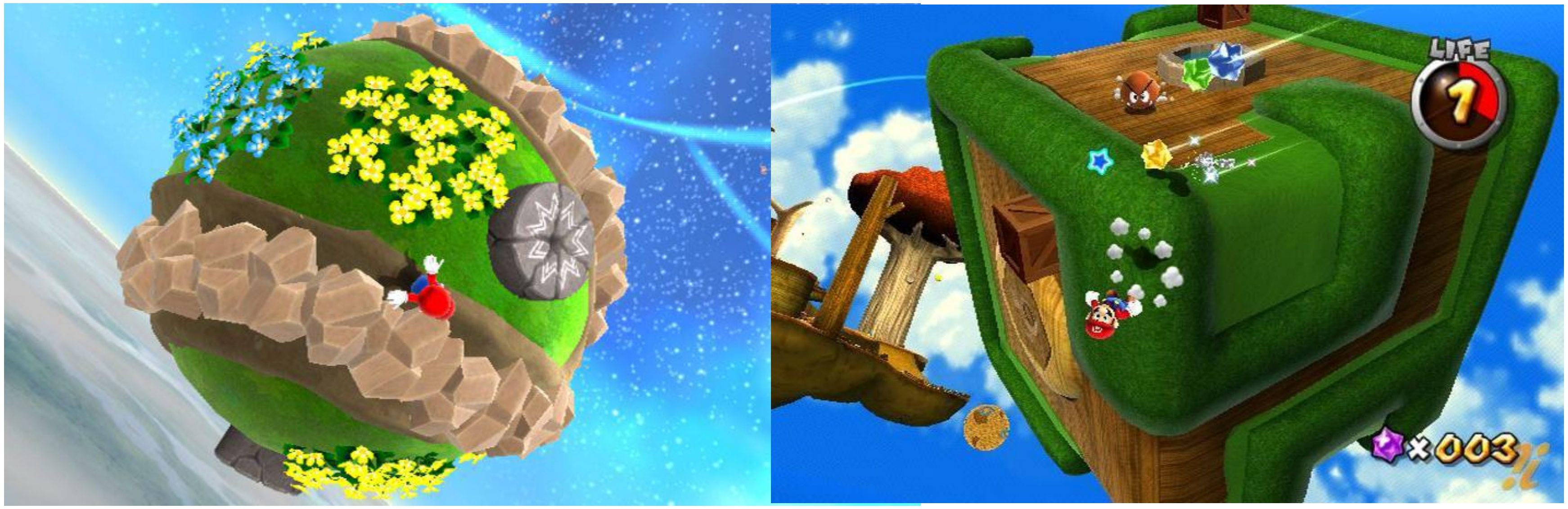
## View Matrix: look-at

In *y-up* coordinate systems, it is usually set to *u=(0, 1, 0).*
In this way, the camera oriented with the y-axis perpendicular to the horizon.



## View Matrix: look-at

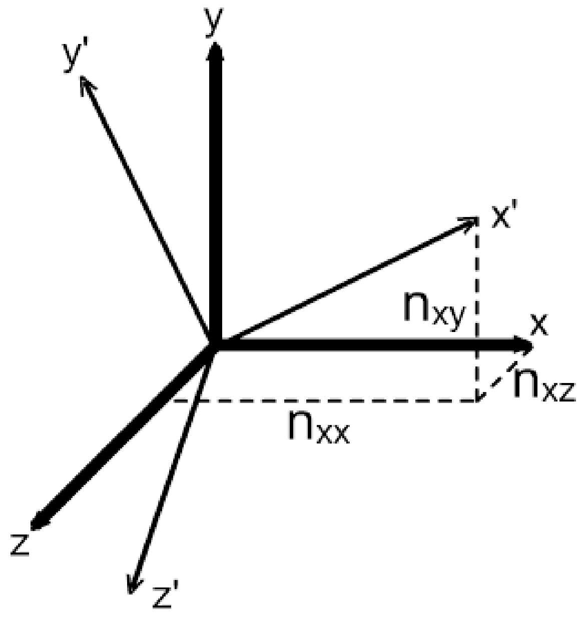
In some applications however, the direction of the up vector is changed to obtain interesting effects on the game plays.



*Super Mario Galaxy, 2007, Nintendo Wii, or 2020 Nintendo Switch*

## View Matrix: look-at

The *View Matrix* is computed by first determining the direction of its axis in World coordinates, and then using the corresponding information to build the *Camera Matrix*.



$$M_R = \begin{vmatrix} \boxed{\begin{matrix} n_{xx} \\ n_{xy} \\ n_{xz} \end{matrix}} & \begin{matrix} n_{yx} \\ n_{yy} \\ n_{yz} \end{matrix} & \begin{matrix} n_{zx} \\ n_{zy} \\ n_{zz} \end{matrix} \end{vmatrix}$$

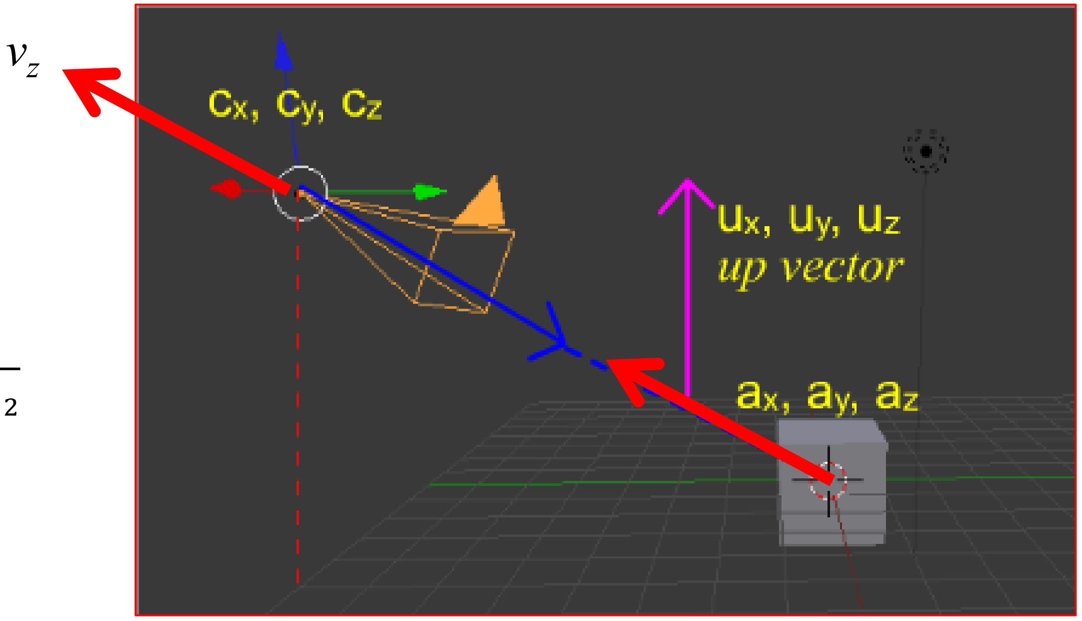
## View Matrix: look-at

We first determine the transformed (negative) *z-axis* as the normalized vector that ends into the camera center and that starts from the point that it is looking.

Normalization (unit size) is obtained by dividing for the length of the resulting vector.

$$v_z = \frac{c - a}{|c - a|}$$

$$|c-a| = \sqrt{(c_x - a_x)^2 + (c_y - a_y)^2 + (c_z - a_z)^2}$$



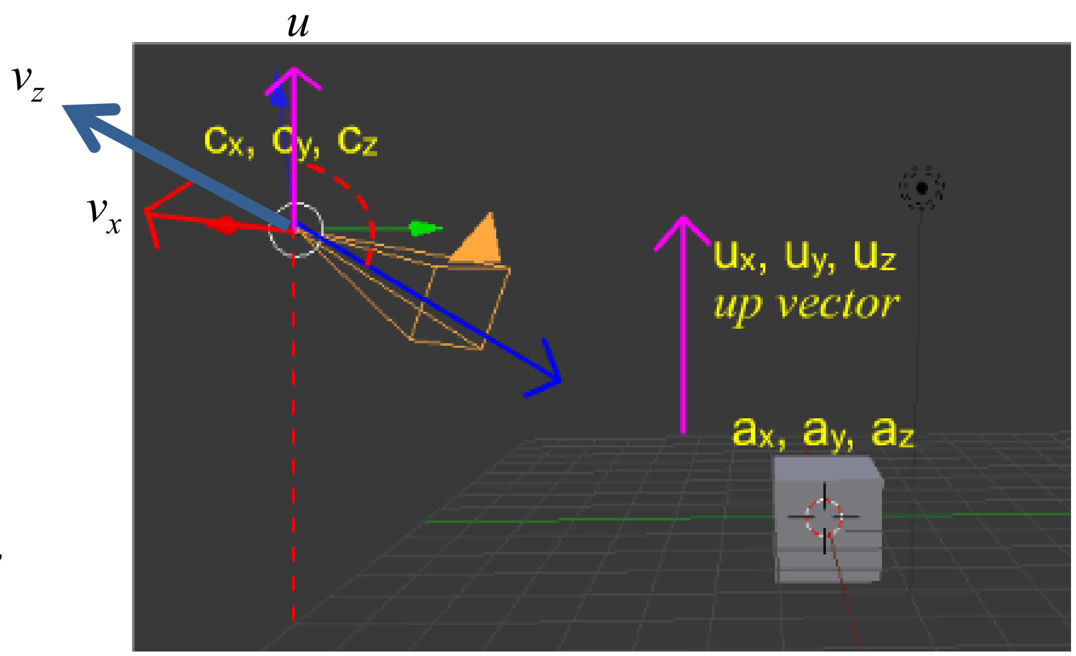
## View Matrix: look-at

The new x-axis must be perpendicular to both the new *z-axis*, and the *up-vector*: it can be computed via the normalized cross product of the two.

$$v_z = \frac{c - a}{|c - a|}$$

$$v_x = \frac{u \times v_z}{|u \times v_z|}$$

Even if both vectors are unitary, they are not always perpendicular. Without normalization, the result will be of a non-unitary size.



For convenience, the cross product of two 3 components vectors is defined as:

$$|u_x \quad u_y \quad u_z| \times |v_x \quad v_y \quad v_z| = |u_y v_z - u_z v_y \quad u_z v_x - u_x v_z \quad u_x v_y - u_y v_x|$$

## View Matrix: look-at

Note that the cross product returns zero if the two vectors *u* and $v_z$ are aligned.

This makes it impossible to determine vector $v_x$, and thus to find a proper camera matrix.

Such problem occurs when the viewer is perfectly vertical, and thus it is impossible to align the camera with the ground: the simplest solution is to use the previously computed matrix, or select a random orientation for the *x-axis*.

$$v_x = \frac{\boxed{u \times v_z}}{|u \times v_z|}$$

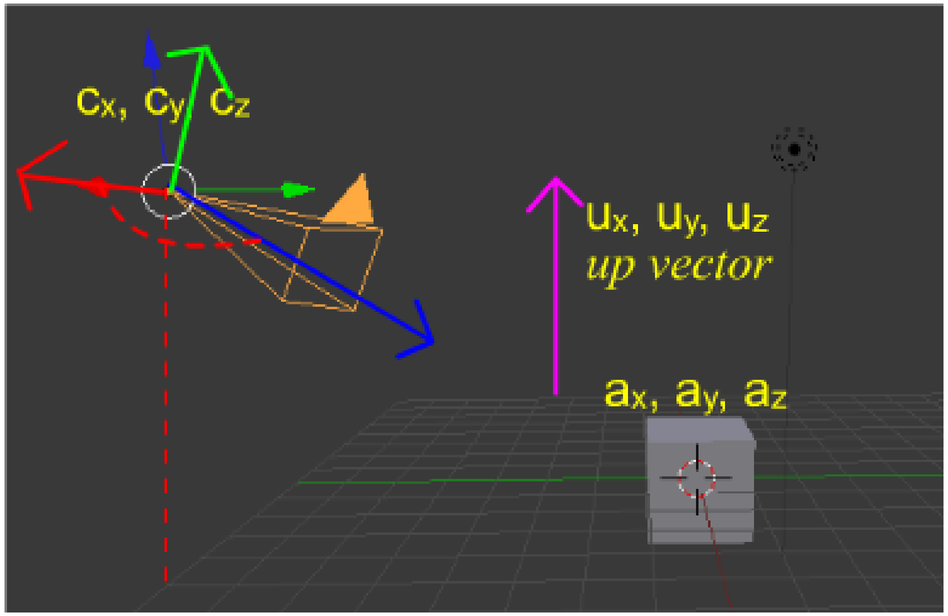
## View Matrix: look-at

Finally, the new *y-axis* should be perpendicular to both the new *z-axis* and the new *x-axis*. This could be computed via the cross product of the two vectors just obtained.

Since both the new *z-axis* and the new *x-axis* are by construction perpendicular and unit vectors, normalization is not required.

$$v_z = \frac{c - a}{|c - a|} \quad v_x = \frac{u \times v_z}{|u \times v_z|}$$

$$v_y = v_z \times v_x$$



## View Matrix: look-at

The *Camera Matrix* $M_C$ can then be computed by placing vectors $v_x$, $v_y$ and $v_z$ in the first three columns and the position of the center $c$ in the fourth.

$(c_x\ c_y\ c_z)$

$$v_z = \frac{c - a}{|c - a|}$$

$$v_x = \frac{u \times v_z}{|u \times v_z|}$$

$$v_y = v_z \times v_x$$

$$M_C = \left|\begin{array}{ccc|c} v_x & v_y & v_z & c \\ \hline 0 & 0 & 0 & 1 \end{array}\right|$$

## View Matrix: look-at

The *View Matrix* can be computed inverting $M_C$. Since the vectors are orthogonal, the inversion of a look-at camera matrix can be computed very easily with a transposition and a matrix-vector product:

$$v_z = \frac{c - a}{|c - a|}$$

$$v_x = \frac{u \times v_z}{|u \times v_z|}$$

$$v_y = v_z \times v_x$$

$$M_C = \left|\begin{array}{ccc|c} v_x & v_y & v_z & c \\ \hline 0 & 0 & 0 & 1 \end{array}\right| = \left|\begin{array}{c|c} R_C & c \\ \hline 0 & 1 \end{array}\right|$$

$$\boxed{M_V = [M_C]^{-1} = \left|\begin{array}{c|c} (R_C)^T & -(R_C)^T \times c \\ \hline 0 & 1 \end{array}\right|}$$

## Cross product, normalization and definition by column in GLM

GLM can compute the cross product of two vectors using the `cross()` function:

```
glm::vec3 uXvz = glm::cross(u, vz);
```

A vector can be made unitary with the `normalize()` function:

```
glm::vec3 vx = glm::normalize(uXvz);
```

Finally, there is a constructor to build a 4x4 matrix starting from 4 column vectors:

```
glm::mat4 Mc = glm::mat4(vx, vy, vz, glm::vec4(c.x,c.y,c.z,1));
```

## Cross product, normalization and definition by column in GLM

In particular, it can be implemented as:

```
glm::vec3 vz = glm::normalize(c - a);
glm::vec3 vx = glm::normalize(glm::cross(u, vz));
glm::vec3 vy = glm::cross(vz, vx);
glm::mat4 Mc = glm::mat4(glm::vec4(vx),
                         glm::vec4(vy),
                         glm::vec4(vz),
                         glm::vec4(c,1));
Glm::mat4 Mv = glm::inverse(Mc);
```

Where `c`, `a` and `u` are three `glm::vec3` representing respectively the position of the camera, the target point and the up vector, and `Glm::mat4 Mv` is the computed view matrix.

## Look at matrix in GLM

Moreover, GLM has the `lookAt()` functions that creates a Look-at matrix starting from three `glm::vec3` vectors representing respectively the center of the camera, the point it targets, and it up-vector:

```
glm::mat4 Mv = glm::lookAt(glm::vec3(cx, cy, cz),
                           glm::vec3(ax, ay, az),
                           glm::vec3(ux, uy, uz));

glm::mat4 Mv = glm::lookAt(c, a, u);
```

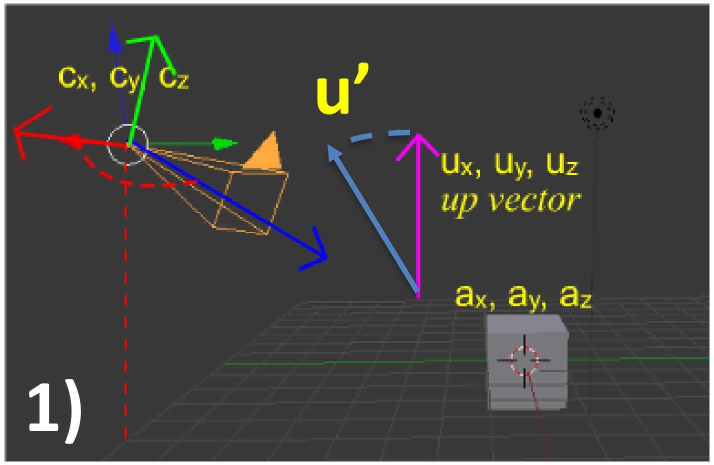
## Look at matrix in GLM

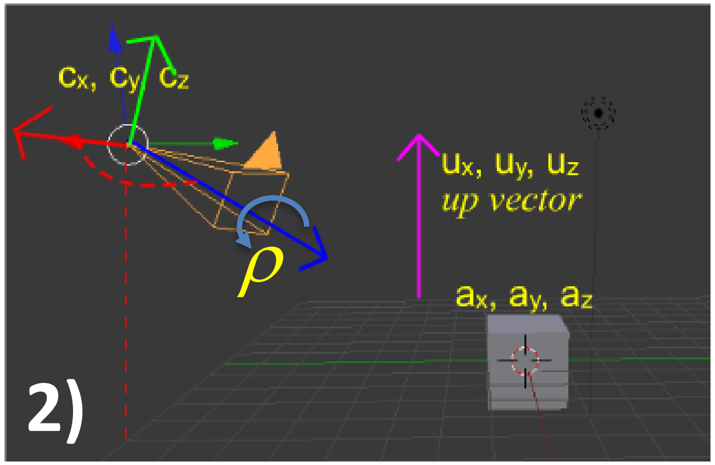
Roll in Look At matrices can be implemented in two ways:

1. Rotating the **u** vector
2. Adding a rotation of an angle $\rho$ around the z-axis.

In the last case, it can be implemented as follows:

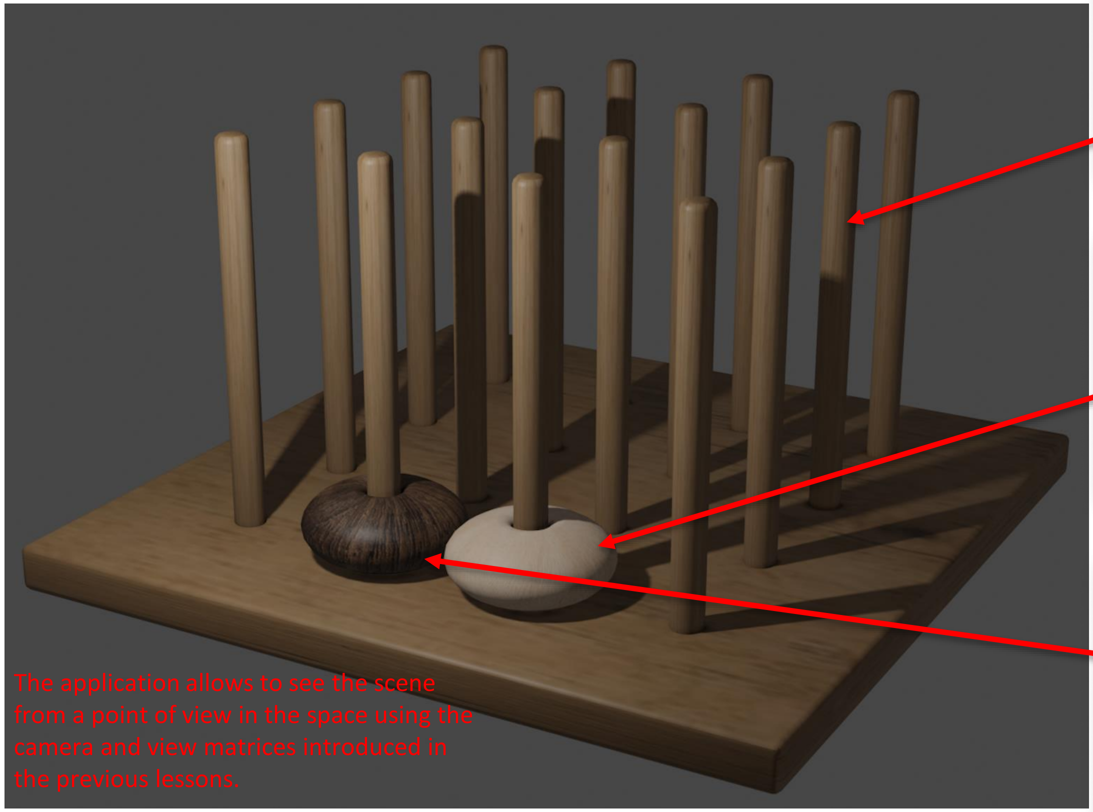
```
glm::mat4 Mv = glm::rotate(glm::mat4(1.0), -Roll, glm::vec3(0,0,1)) *
               glm::lookAt(c, a, u);
```



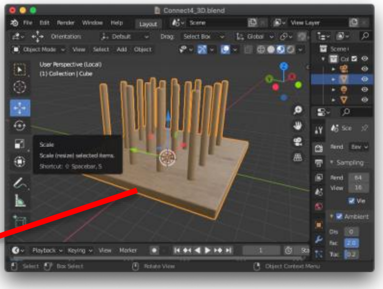


# Application requirements

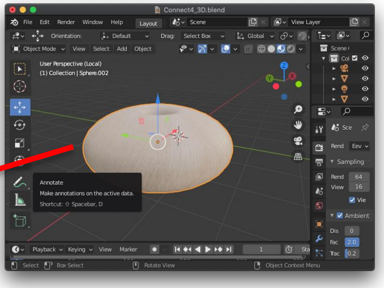
A typical application has to position objects in space.



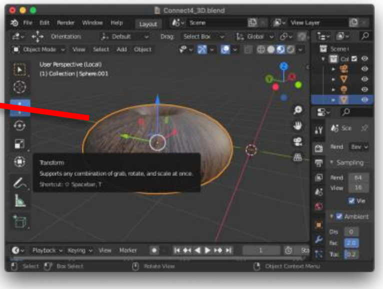
The application allows to see the scene from a point of view in the space using the camera and view matrices introduced in the previous lessons.



Assets are created in an external tool, such as Blender, and imported in the application.



The application then replicates and places the models in specific positions to create the gameplay.

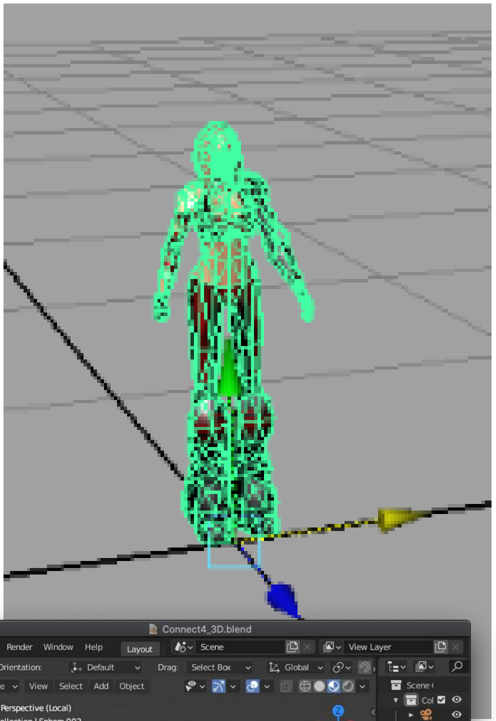

Connect4_3D.blend
User Perspective (Local)
(1) Collection | Sphere.001
Transform
Supports any combination of grab, rotate, and scale at once.
Shortcut: ⇧ Spacebar, T

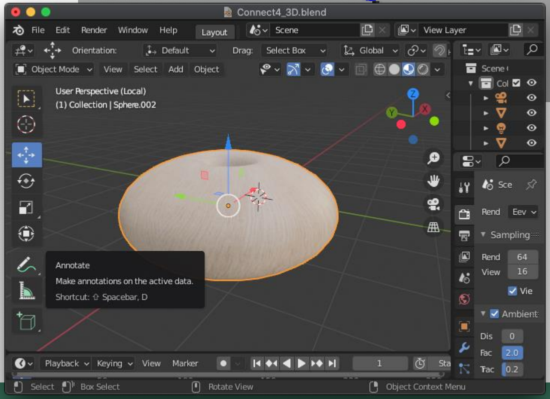
## Local coordinates and World Matrix

One of the main features of 3D computer graphics is the ability of placing and moving objects in the virtual space.

Movement of the objects is usually performed with a transformation matrix: the *World Matrix* $M_W$.

Every object is characterized by a set of *local coordinates*: the positions of the object's points in the space where it was created.

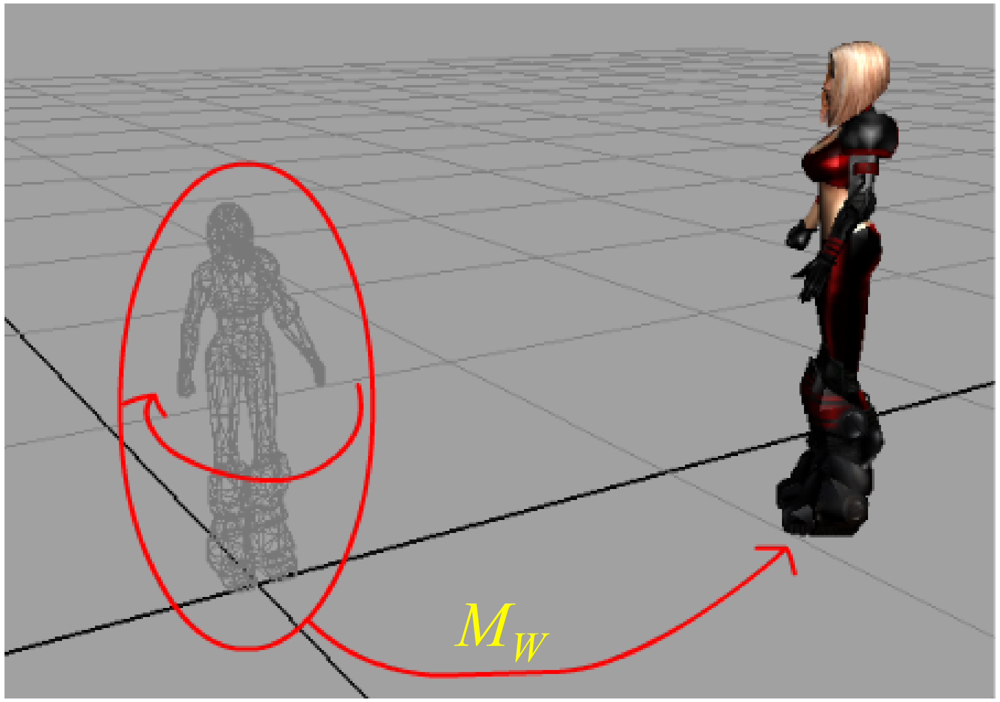


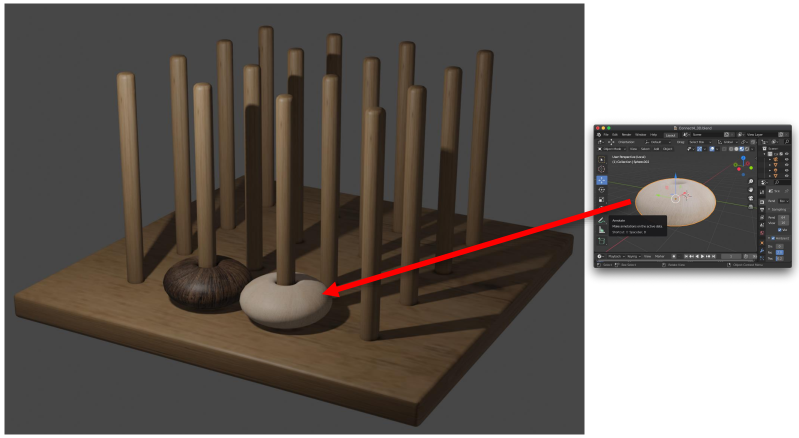


# Local coordinates and World Matrix

When a scene is composed, the object points are moved from the positions they were modeled, to the one where they are shown.

The coordinates reached after this transformation are the object's *global* (or *world*) *coordinates* previously introduced.





## Local coordinates and World Matrix

The *World Matrix* $M_w$ transforms the *local coordinates* into the corresponding *global ones*.

The world matrix applies a series of translations, rotations (and possibly scaling or shears) to the local coordinates.

As we have seen, the order of transformations is important since matrix product is not commutative.

Even if there is not a single solution, there are some best practices that usually yield the desired results.

## Creating a World Matrix

Given an object described in local coordinates the user generally wants to:

- Position the object
- Rotate the object
- Change the scale / Mirror the object

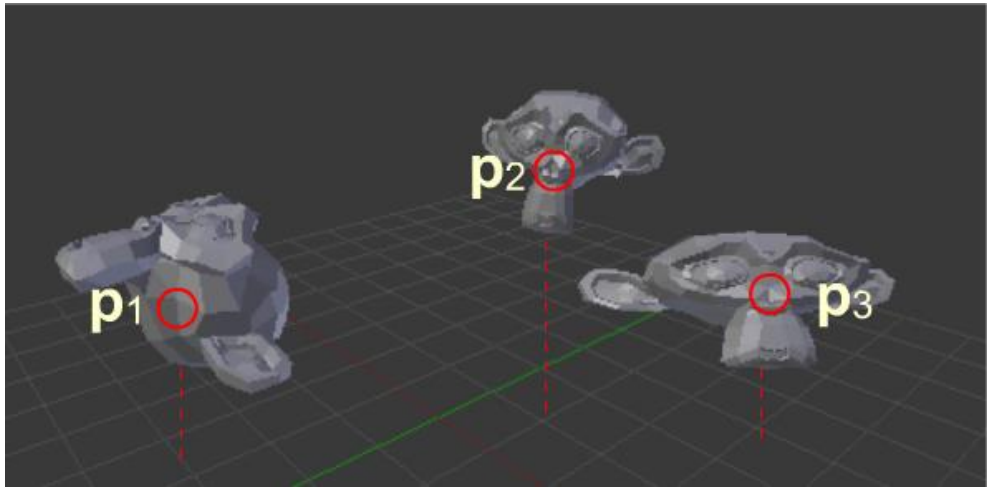
Shear and more advanced transformations, like scaling or rotations along arbitrary axes, are generally not considered since they cannot be easily generalized: should these transforms be necessary, custom procedures must be developed.

## Creating a World Matrix: positioning

When positioning an object, the user wants to specify the coordinates where it should be placed in the 3D space.

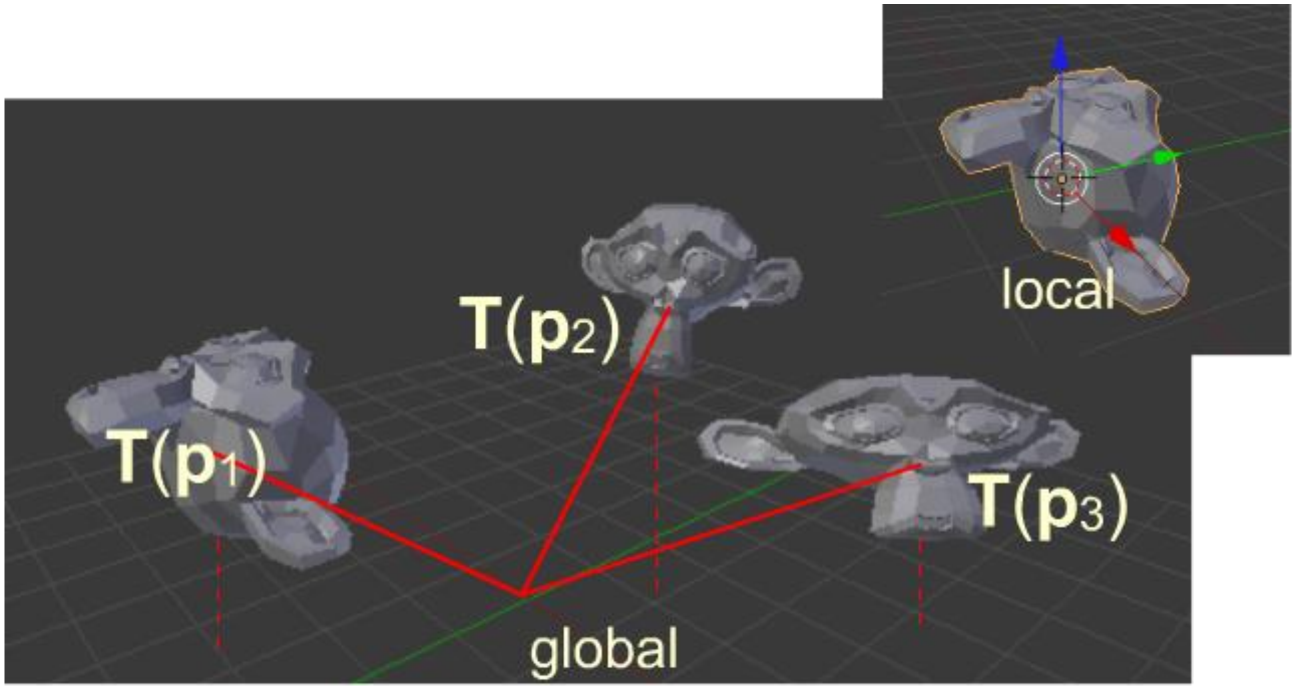
These coordinates should be independent of the size and on the orientation of the object.

Positioning at $p=(p_x,p_y,p_z)$ must then be performed by applying translation $T(p_x,p_y,p_z)$ as the **last** transform (otherwise the location would be changed during rotation or scaling).



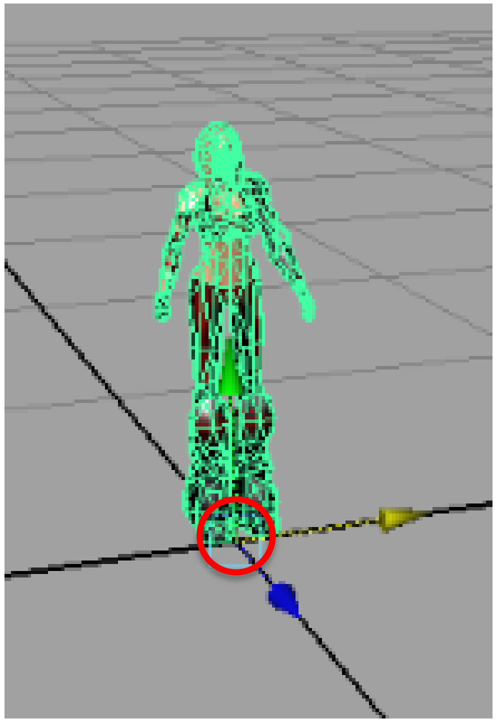
## Creating a World Matrix: positioning

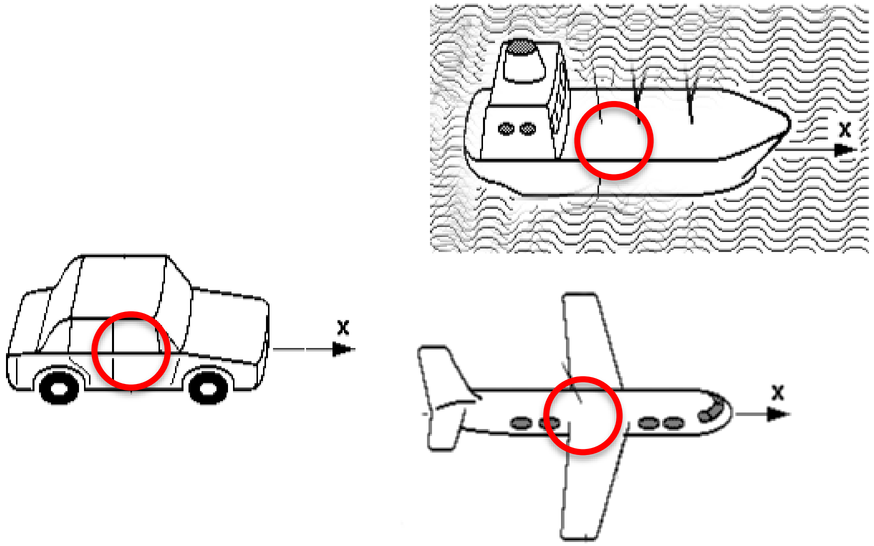
The parameters $p=(p_x,p_y,p_z)$ of the translation $T(p_x,p_y,p_z)$ defines the position that the origin of the object (in its local space) will occupy after being moved in the world space.



## Euler angles

It is important that the object being positioned has been modelled with the origin in the point that is most representative of its location. The exact position of this point varies from object to object.

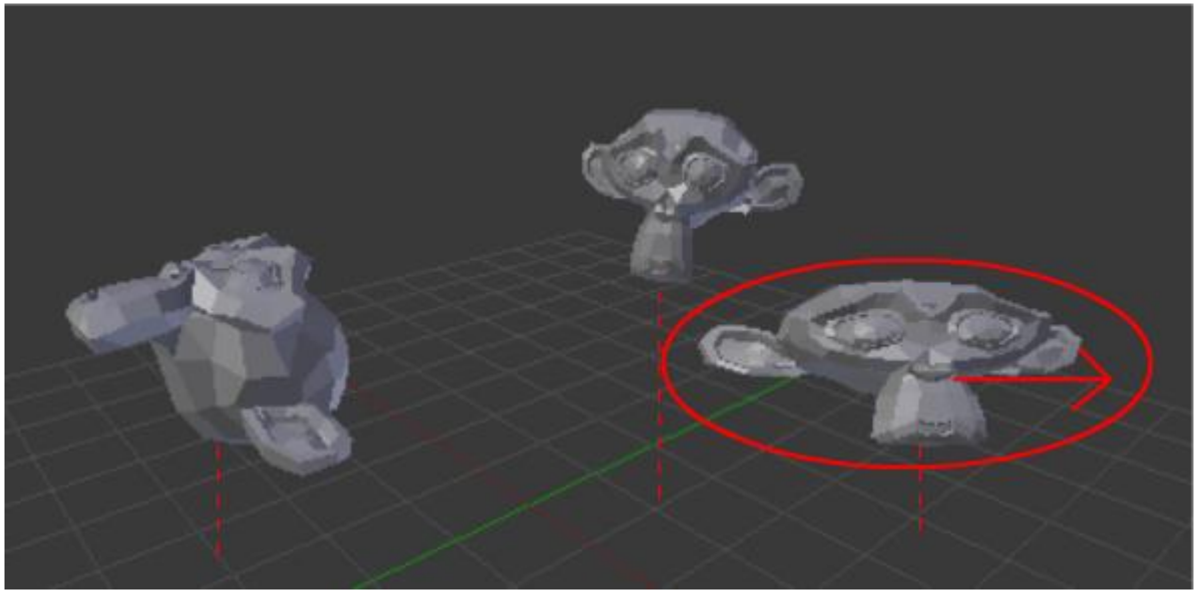




## Creating a World Matrix: scaling

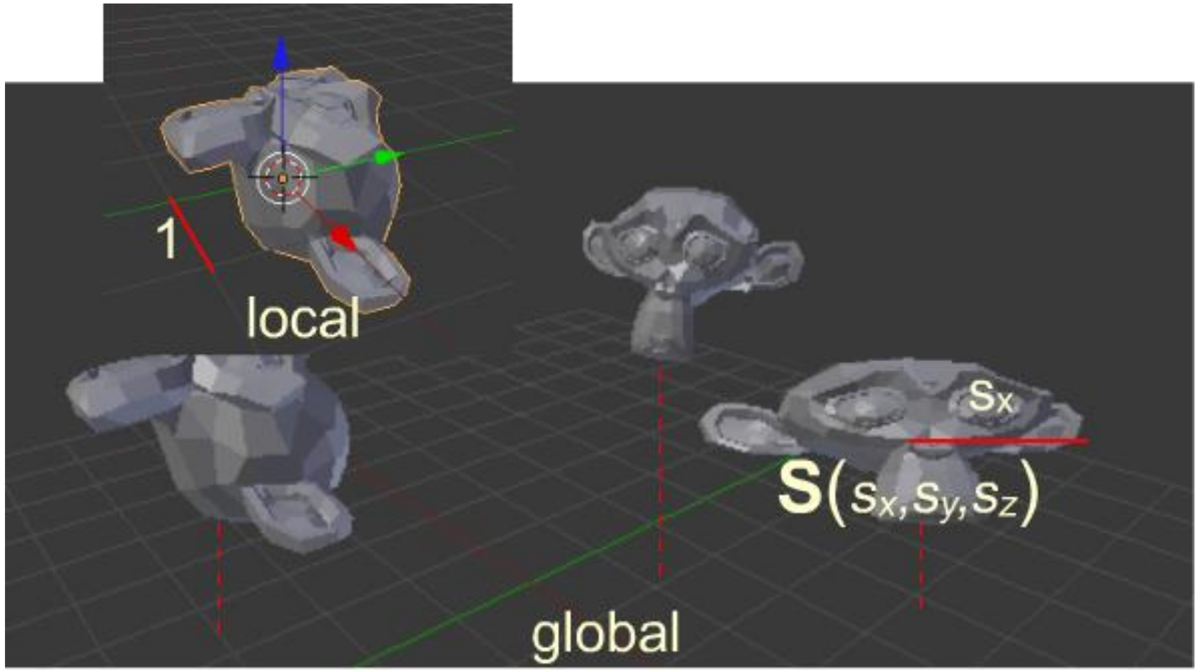
If the object should be of $s_x, s_y, s_z$, rotations must be applied after, otherwise scaling direction would be oriented differently.

Scaling of $s_x, s_y, s_z$ must then be performed by a transformation $S(s_x, s_y, s_z)$ that is applied **first**.



## Creating a World Matrix: scaling

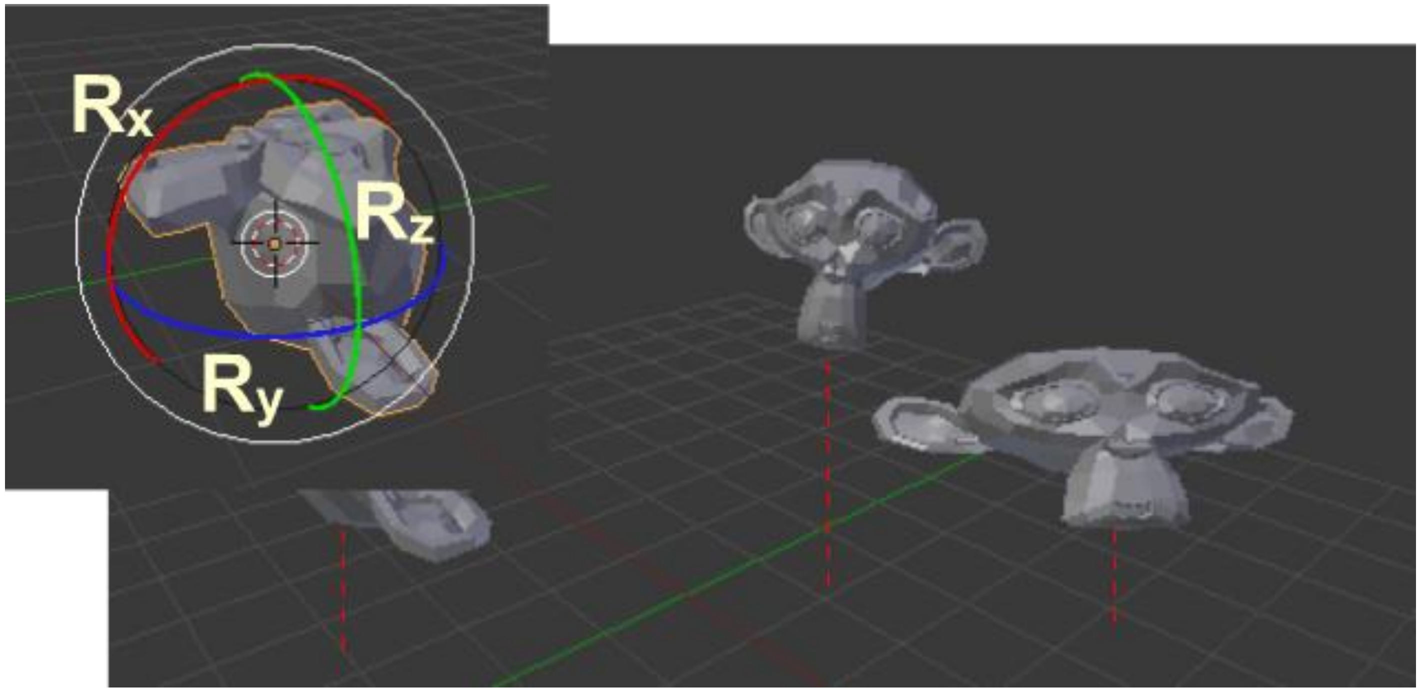
A scaling factor $s_x$ means that a unitary segment in local coordinates (along the *x-axis*), becomes $s_x$ units in global coordinates. The same applies also to factors $s_y$ and $s_z$.



## Creating a World Matrix: rotation

Rotation should then be performed **in between scaling and translation**.

To define an orientation in a 3D space, the basic rotations along the three axes *x*, *y* and *z*, should be combined.



## Euler angles

Several different ways to compute the rotation matrix for an object exist.

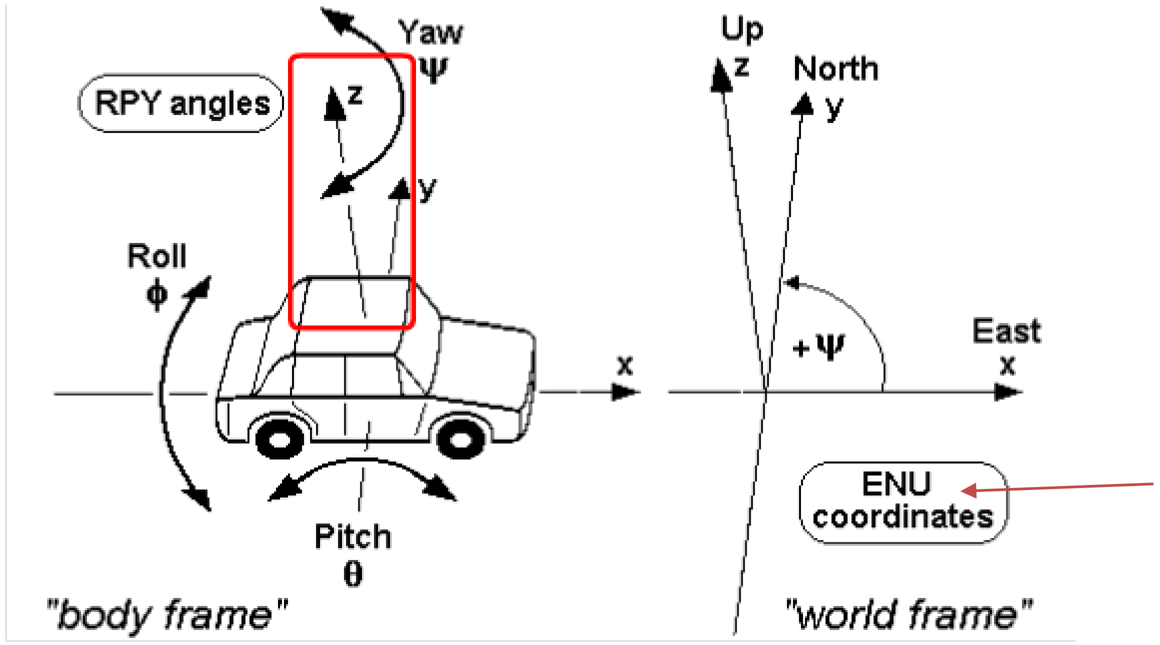
One of the goals of creating the rotation component of the world matrix is to simplify the specification of the parameters required by the application to define the direction of an object.

The *Euler Angles* are three parameters that define the orientation of an object in a 3D space, and they are usually addressed as:

- **Roll**
- **Pitch**
- **Yaw**

## Euler angles

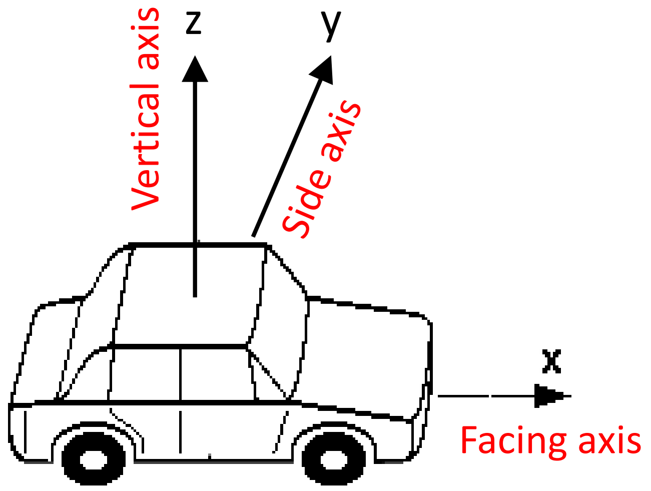
Euler angles were introduced for the *z-up* coordinates systems, and are much easier to remember in that case. For this reason, we will initially explain them in a *z-up* coordinate system: in this case the World's North will be aligned with the positive *y-axis*.



ENU means:
- East on the *x*
- North on the *y*
- Up on the *z*

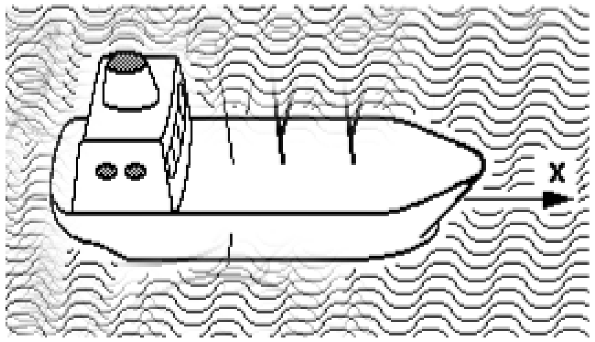
## Euler angles

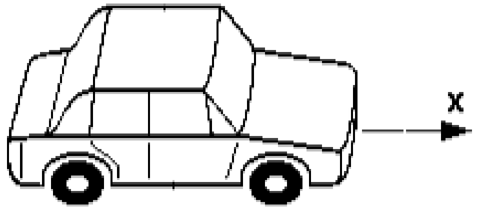
Objects must be modeled such that they are *facing* along the positive *x-axis*, their *side* is aligned with the *y-axis*, and their *vertical line* is oriented along the *z-axis*.

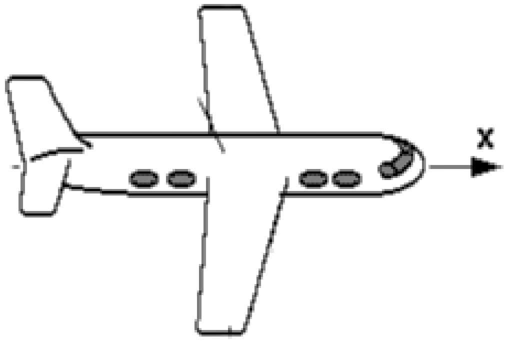


## Euler angles

The facing axis can be for example where the car has its front view, or where an airplane or a ship are directed.



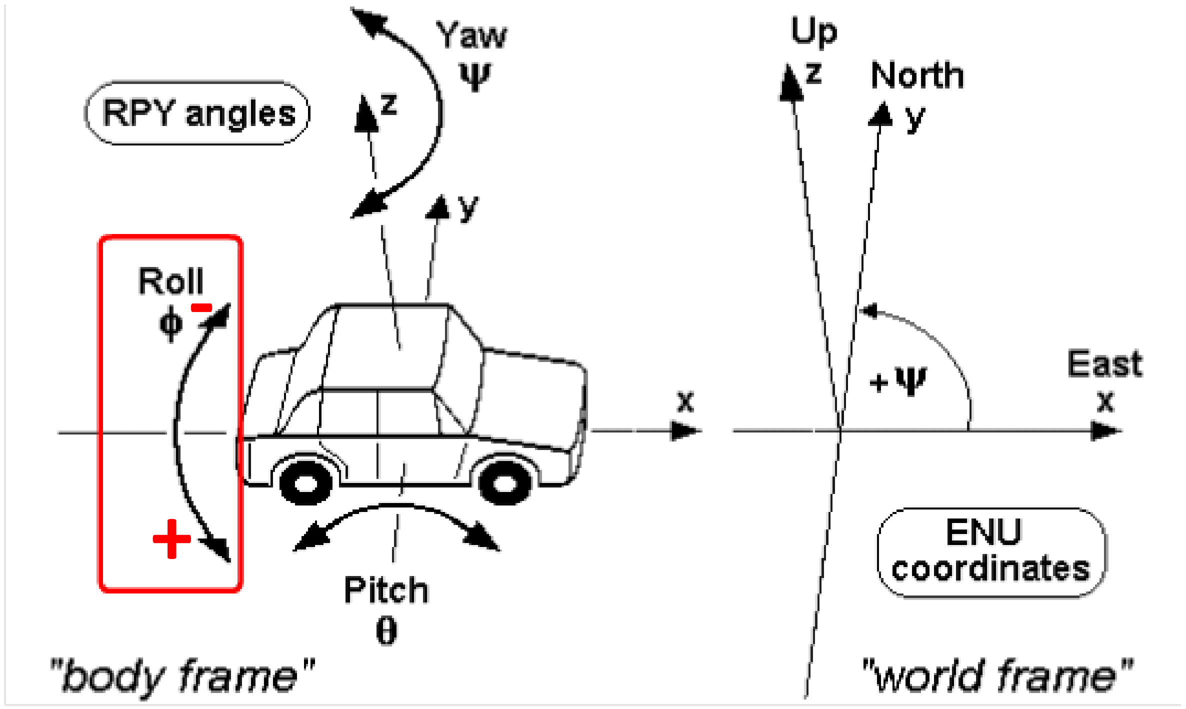




## Euler angles

The *Roll* $\phi$ identifies the rotation of the object along its facing axis (the x-axis).

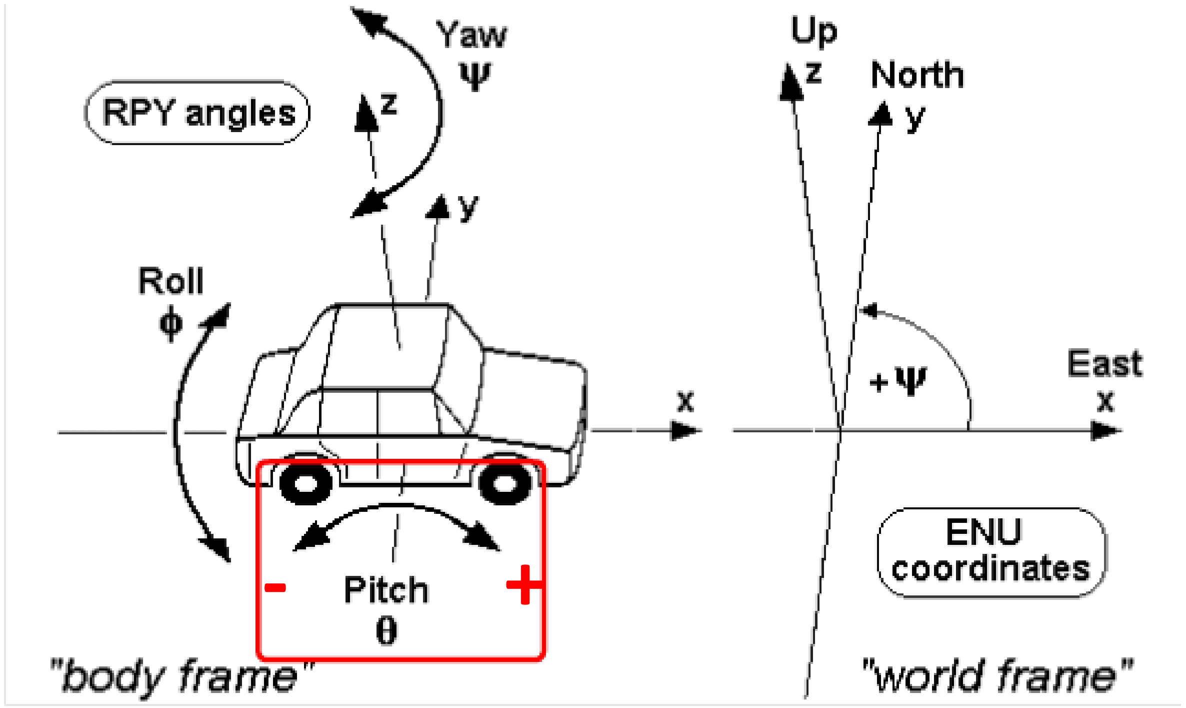
A positive roll, turns the object *clockwise*.



## Euler angles

The *Pitch* $\theta$ defines the elevation of the object and corresponds to a rotation around its *y-axis* (the *side axis*).

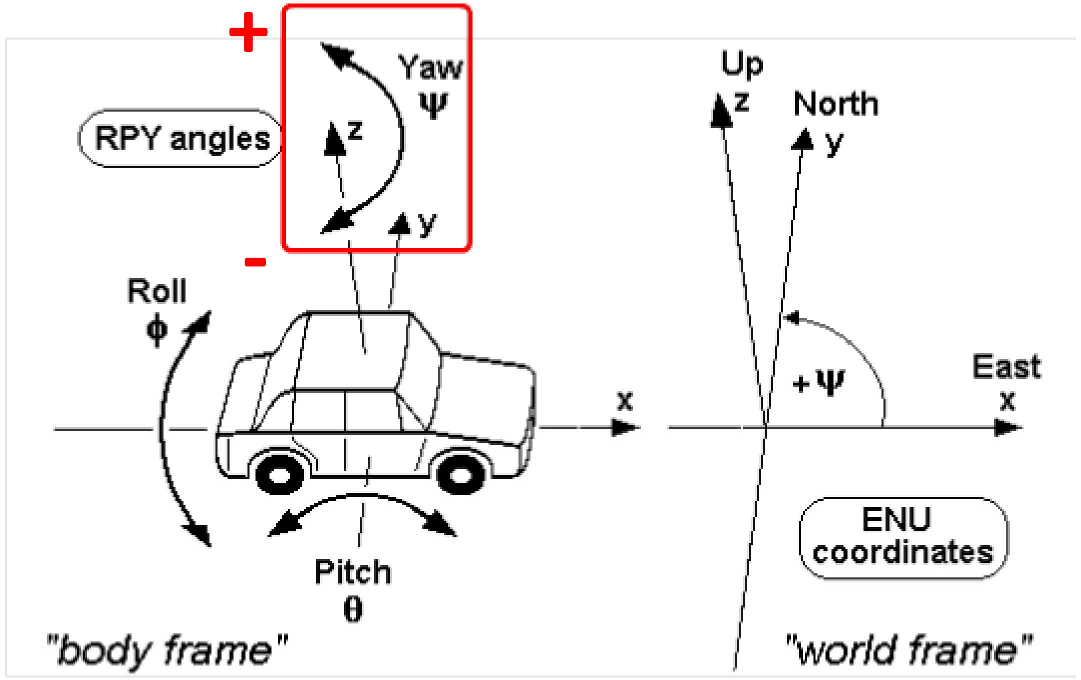
A positive pitch turns the head of the object *facing down*.



## Euler angles

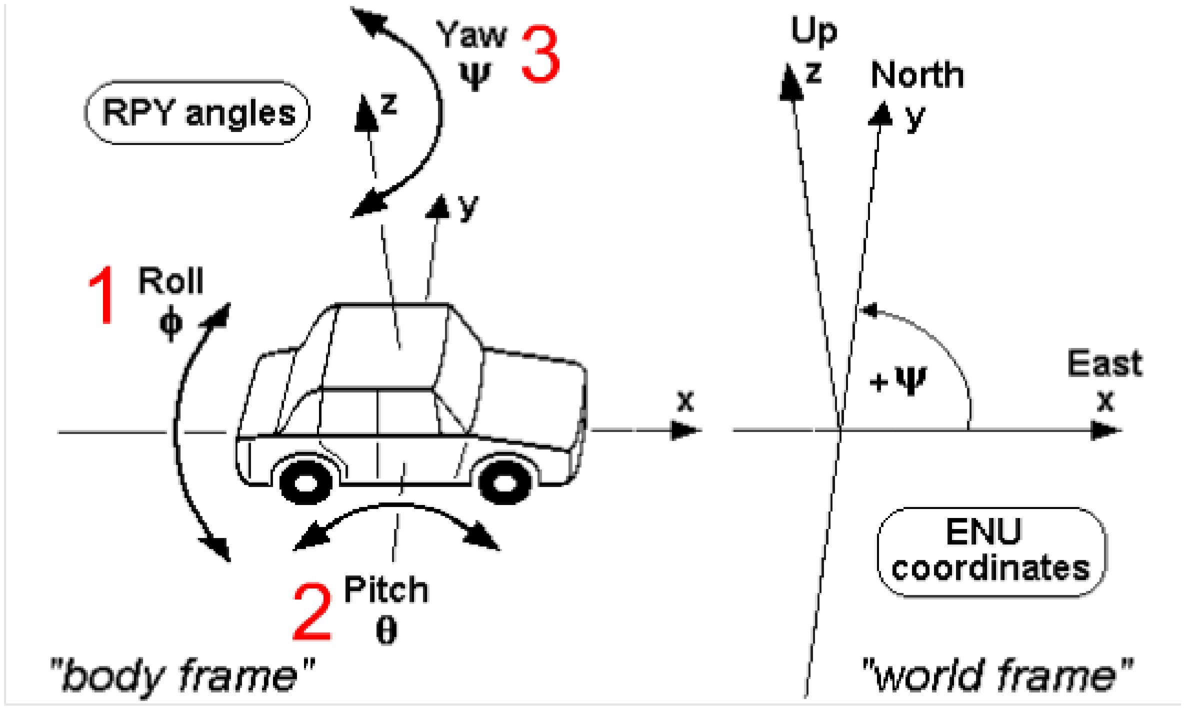
The *Yaw* $\psi$ defines the direction of the object, and corresponds to a rotation along the *z-axis* (the *vertical axis*).

It defines the direction where the object is heading: $\psi=0^o$ is East, $\psi=90^o$ corresponds to North.



## Euler angles

With this convention the rotations are performed in the alphabetic order: *x-axis*, *y-axis* and finally *z-axis*.

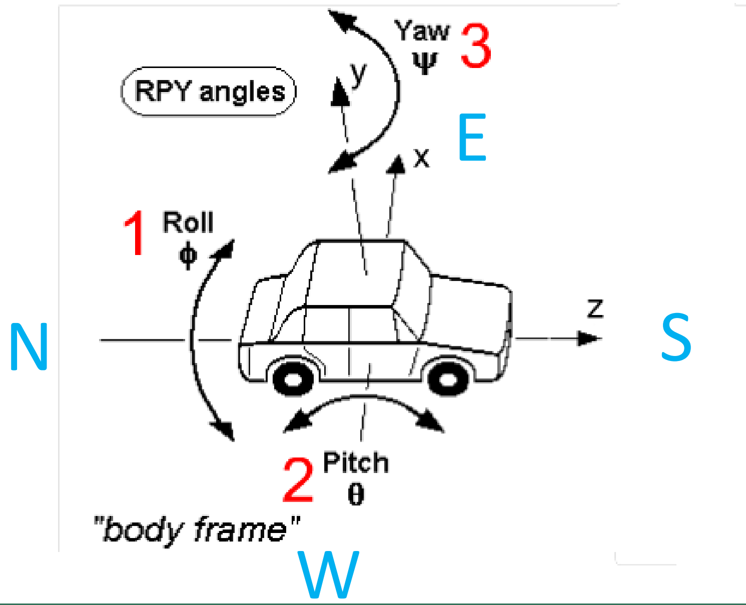


## Euler angles

For different conventions, the axes must always be rotated in the *roll*, *pitch*, *yaw* order (*RPW*).

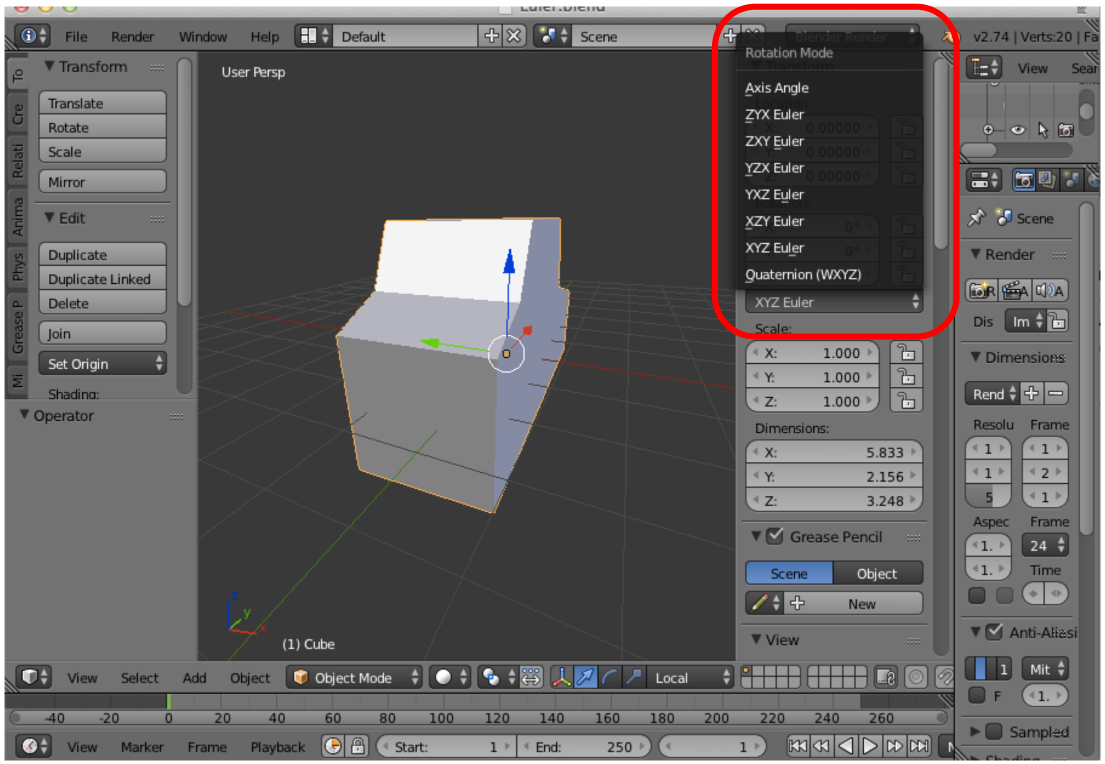
When considering a *y-up* system, where the object faces the positive *z-axis*, rotations should be performed in the *z*, *x* and *y* order.

In this case, a yaw $\psi=0^o$ makes the object face South, and $\psi=90^o$ East.



# Euler angles

Since different conventions can be used, 3D tools (such as Blender) might support several rotation orders and allow the user to choose the one that better suits her needs.

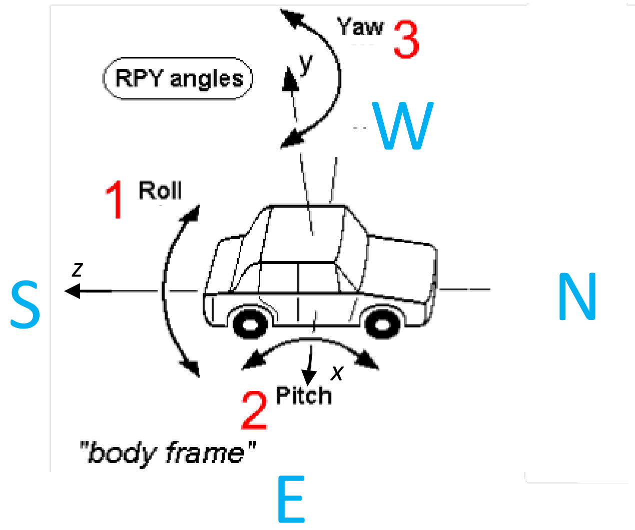


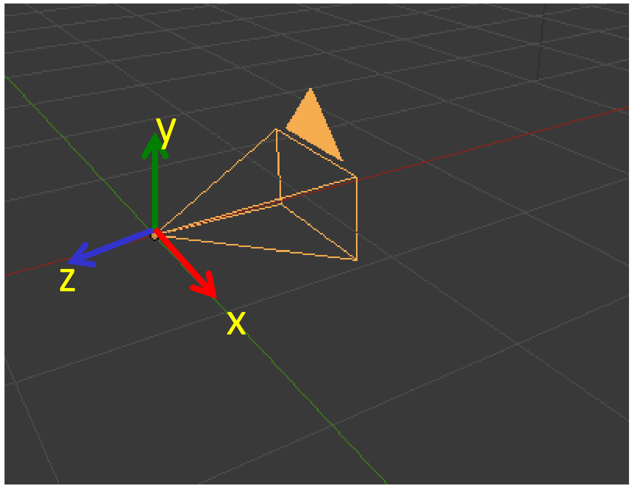
## Euler angles

For example, the *"Look-in-direction"* camera model introduced earlier, uses a *y-up Euler angle* orientation system.

The camera however is oriented along the *negative z-axis* (and not the positive one as introduce here).

For this reason, *Roll* $\phi$ and *Pitch* $\theta$ works in the opposite direction with respect to $\rho$ and $\beta$, and direction $\alpha=0^O$ corresponds to the camera looking *North* instead of *South* for yaw $\psi=0^O$. Rotations are however performed in the same order.

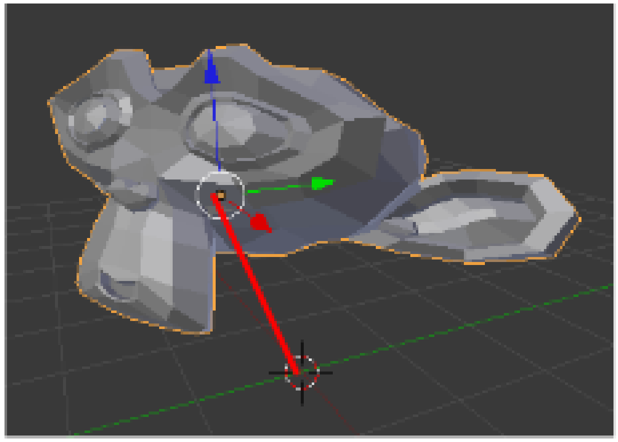




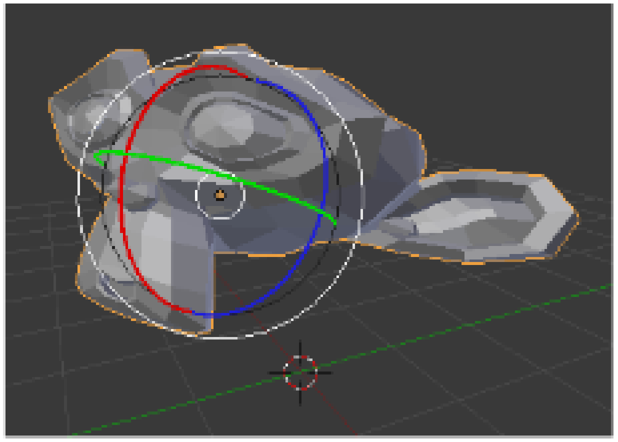
## Creating a World Matrix: final result

With this convention, an object can be positioned in the space using nine parameters: *three positions*, *three Euler angles*, and *three scaling factors*.

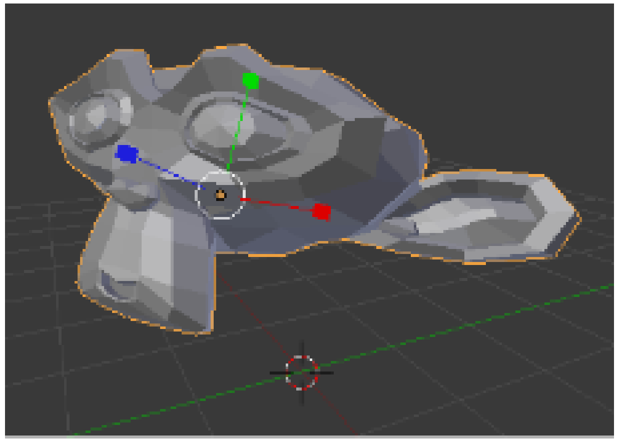
All the parameters have an intuitive "physical" meaning.



$p_x, p_y, p_z$



$\varphi, \psi, \theta$

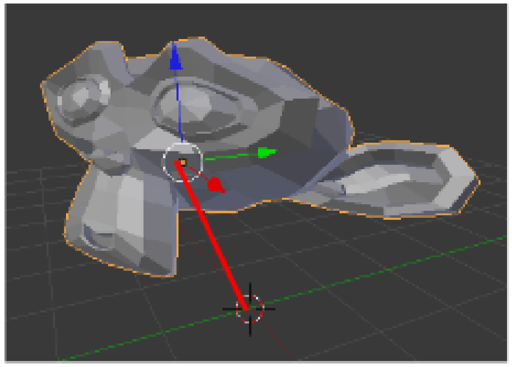


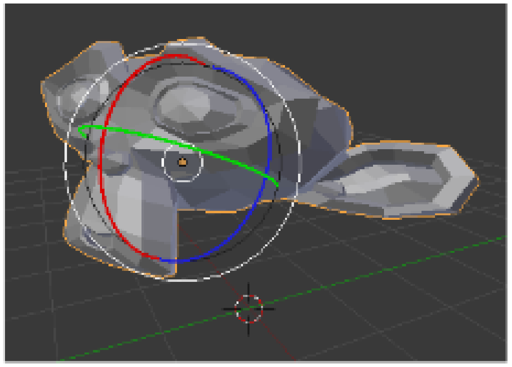
$s_x, s_y, s_z$

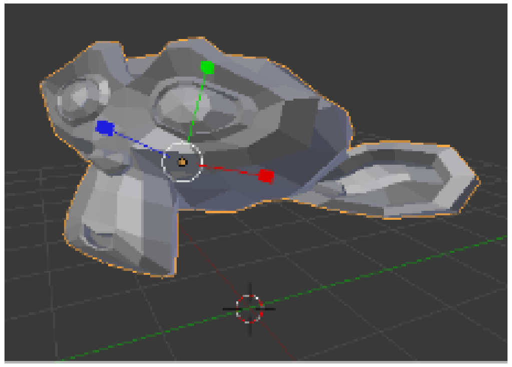
## Creating a World Matrix: final result

The *World Matrix* $M_w$ is then be computed by factorizing the five transformations in the described order:

$$M_W = T(p_x, p_y, p_z) \times R_y(y) \times R_x(q) \times R_z(j) \times S(s_x, s_y, s_z)$$


$p_x$, $p_y$, $p_z$


$\varphi$, $\psi$, $\theta$


$s_x$, $s_y$, $s_z$

## World matrix in GLM

GLM does not have special functions for creating a World matrix starting from basic position, scales and rotations.

However, a specific package of the library has a special function for creating a rotation matrix from Euler angles:

```
#include <iostream>
#include <cstdlib>

#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#define GLM_FORCE_RADIANS

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/euler_angles.hpp>
…
glm::mat4 MEa = glm::eulerAngleYXZ(glm::radians(yaw),
                                   glm::radians(pitch),
                                   glm::radians(roll));
```

## World matrix in GLM

As seen for the view matrix with the Look-In-Direction model, it is generally more practical to define the entire transform sequence ourselves:

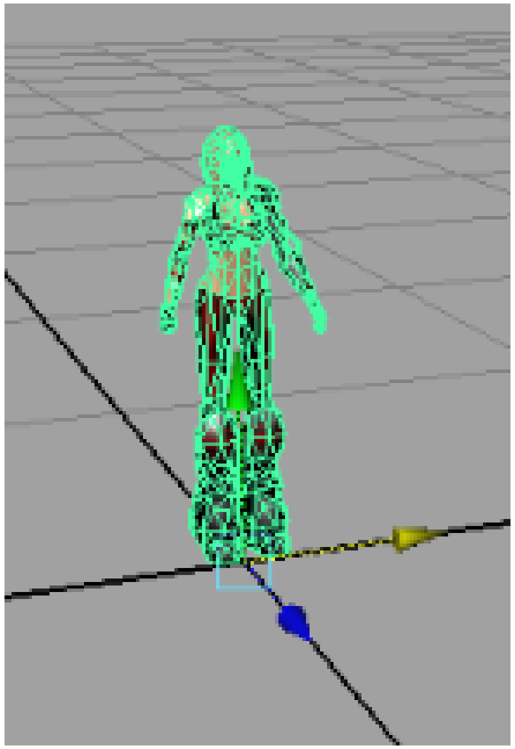
```
glm::mat4 Mw =
   glm::translate(glm::mat4(1.0), glm::vec3(px, py, pz)) *
   glm::rotate(glm::mat4(1.0), yaw   , glm::vec3(0,1,0)) *
   glm::rotate(glm::mat4(1.0), pitch , glm::vec3(1,0,0)) *
   glm::rotate(glm::mat4(1.0), roll  , glm::vec3(0,0,1)) *
   glm::scale(glm::mat4(1.0), glm::vec3(sx, sy, sz));
```

## Conventions

In a real application, the modelers and the developers defines which convention should be used to create proper assets that can be integrated in the project.

An adaptation transform matrix $M_A$, can be chained before the world matrix to adapt the convention followed by third party assets, to the one required by the application.

$$M_W = T(p_x, p_y, p_z) \cdot R_y(\psi) \cdot R_x(\theta) \cdot R_z(\varphi) \cdot S(s_x, s_y, s_z) \cdot M_A$$

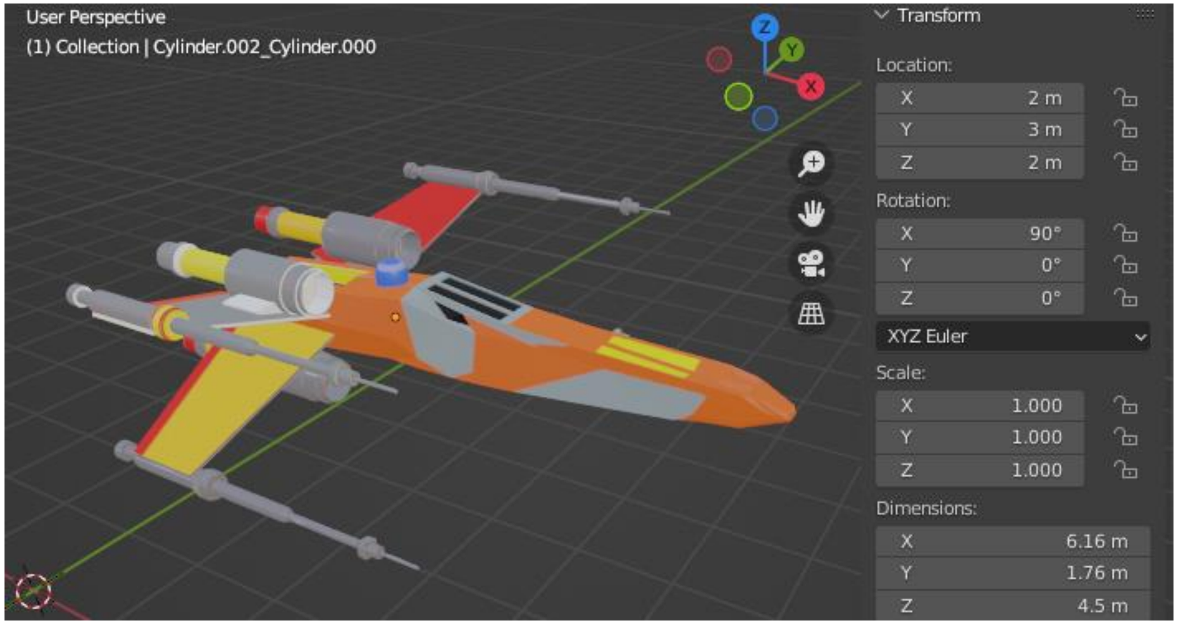


**Asset Conventions:**

*y-up*, middle of the feet in the origin, character facing the negative z-axis. Rotation uses Euler angles in zxy order, character face south with a zero yaw.

## Conventions

For example, this starship has been modelled off-centered (2,3,2), using the *z-up* convention, and two times bigger than required.



User Perspective
(1) Collection | Cylinder.002_Cylinder.000

Transform

Location:

| | |
|---|---|
| X | 2 m |
| Y | 3 m |
| Z | 2 m |

Rotation:

| | |
|---|---|
| X | 90° |
| Y | 0° |
| Z | 0° |

XYZ Euler

Scale:

| | |
|---|---|
| X | 1.000 |
| Y | 1.000 |
| Z | 1.000 |

Dimensions:

| | |
|---|---|
| X | 6.16 m |
| Y | 1.76 m |
| Z | 4.5 m |

It can be corrected with the following adaptation matrix:

$$M_A = S(0.5, 0.5, 0.5) \cdot R_y(90) \cdot R_x(-90) \cdot T(-2, -3, -2)$$

The way in which the adaptation matrix can be defined is very application and asset dependent, and cannot be generalized. It must be custom defined in a different way for each case.

## Marco Gribaudo

*Associate Professor*

CONTACTS

Tel. +39 02 2399 3568
marco.gribaudo@polimi.it
https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)