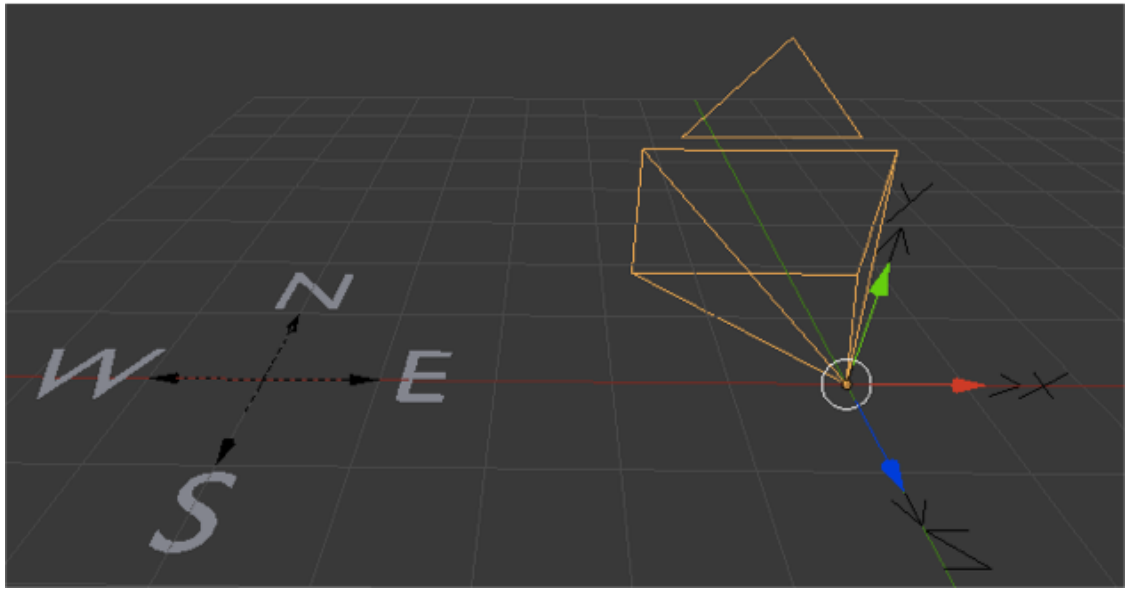
View matrix

Axis and orientation

In this lesson we will see how to view objects from different angles and positions in 3D.

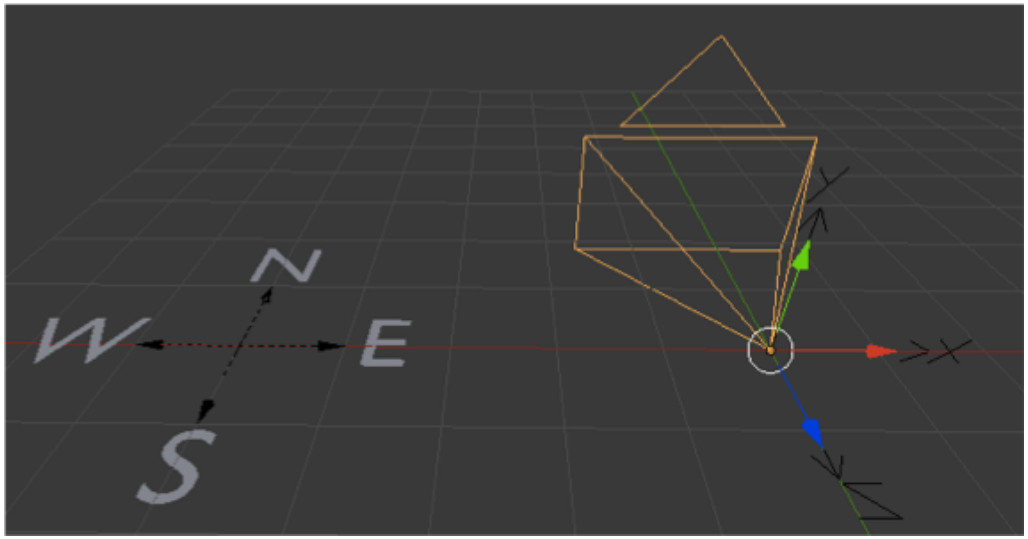
We have defined the *world coordinates* as a reference system to globally specify the positions of an object in the 3D space.

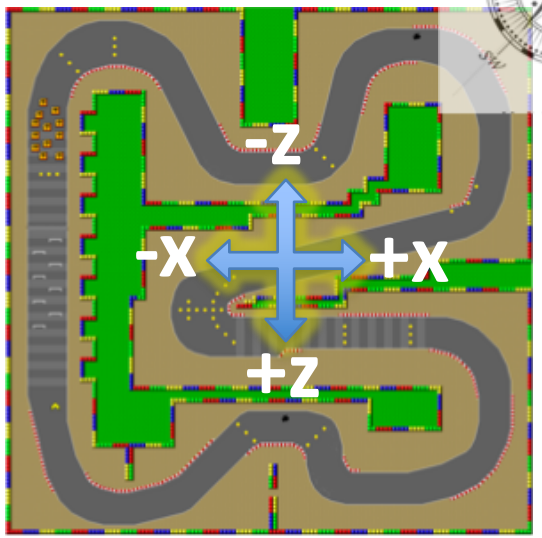


Axis and orientation

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

In particular, we add the convention that the *negative z-axis* corresponds to the *North* of the 3D world, and that the *positive x-axis* to the *East*.



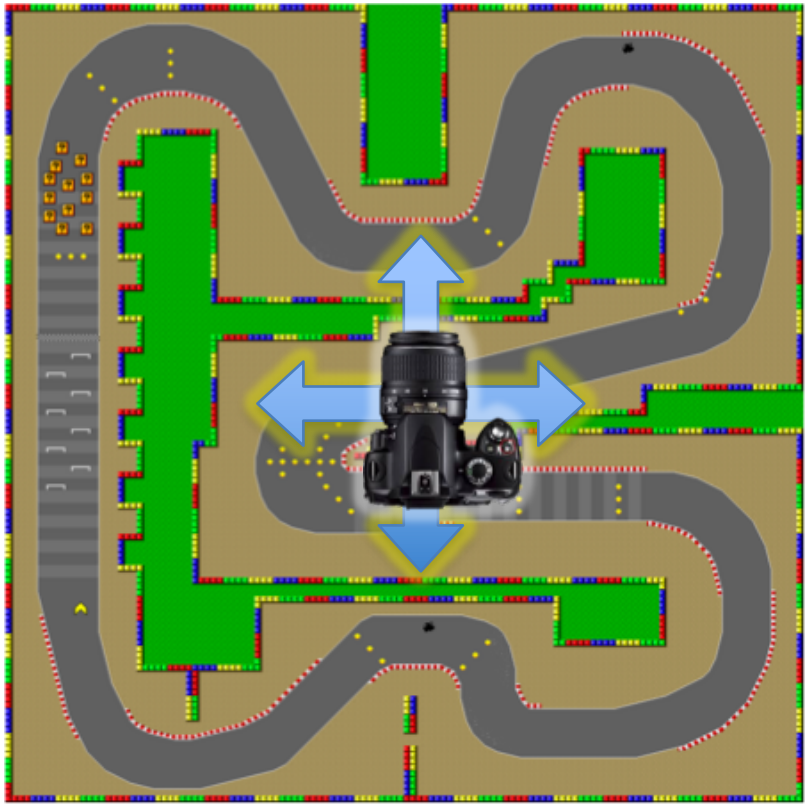


The View Matrix

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

For parallel projections, we also assume that the projection ray is oriented along the *z-axis*.

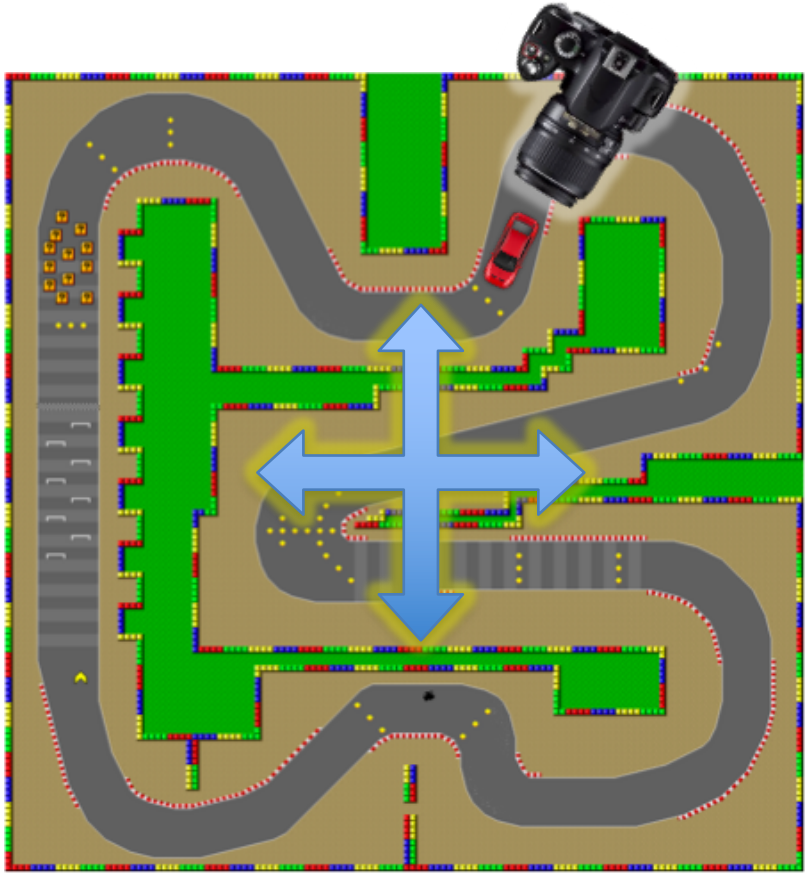
For perspective, we consider the center of projection in the *origin*.



The View Matrix

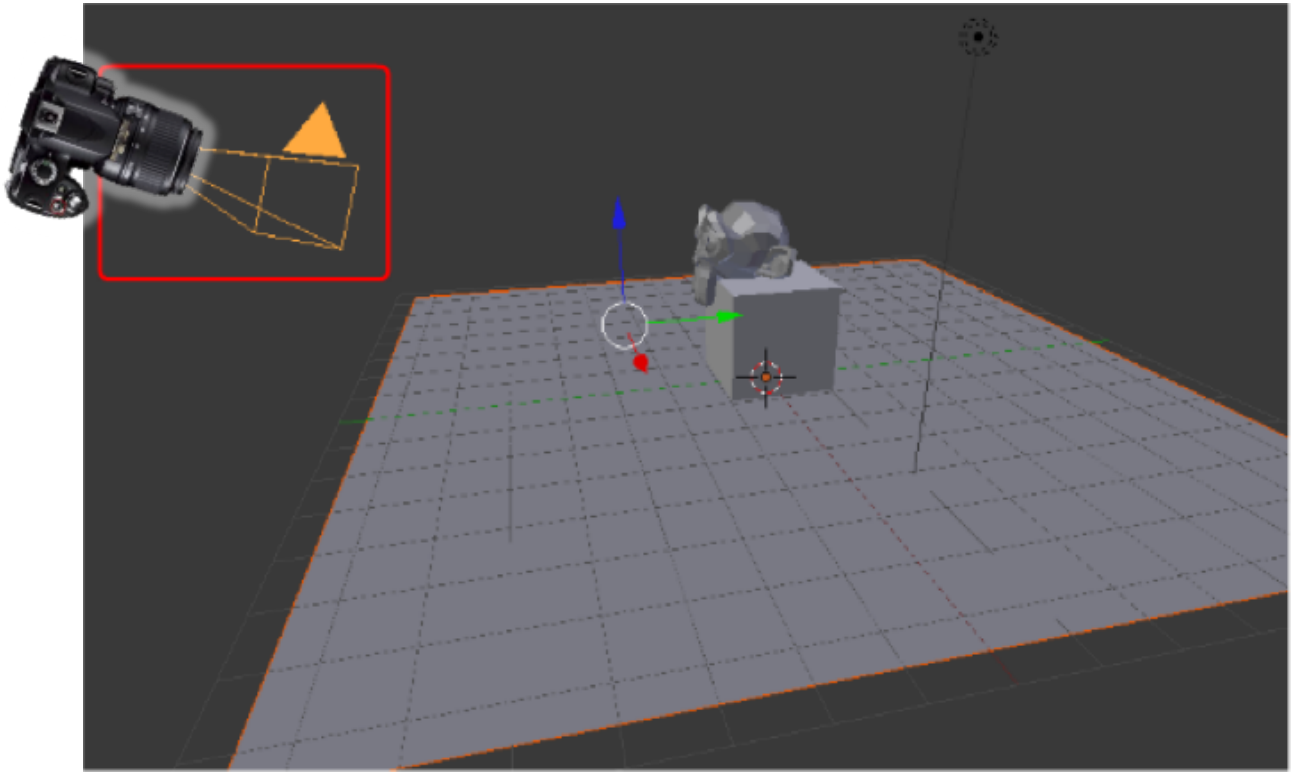
In actual applications, we are interested in generating a 2D view for a plane arbitrarily positioned in the space.

This is achieved adding extra transformations before the projection.



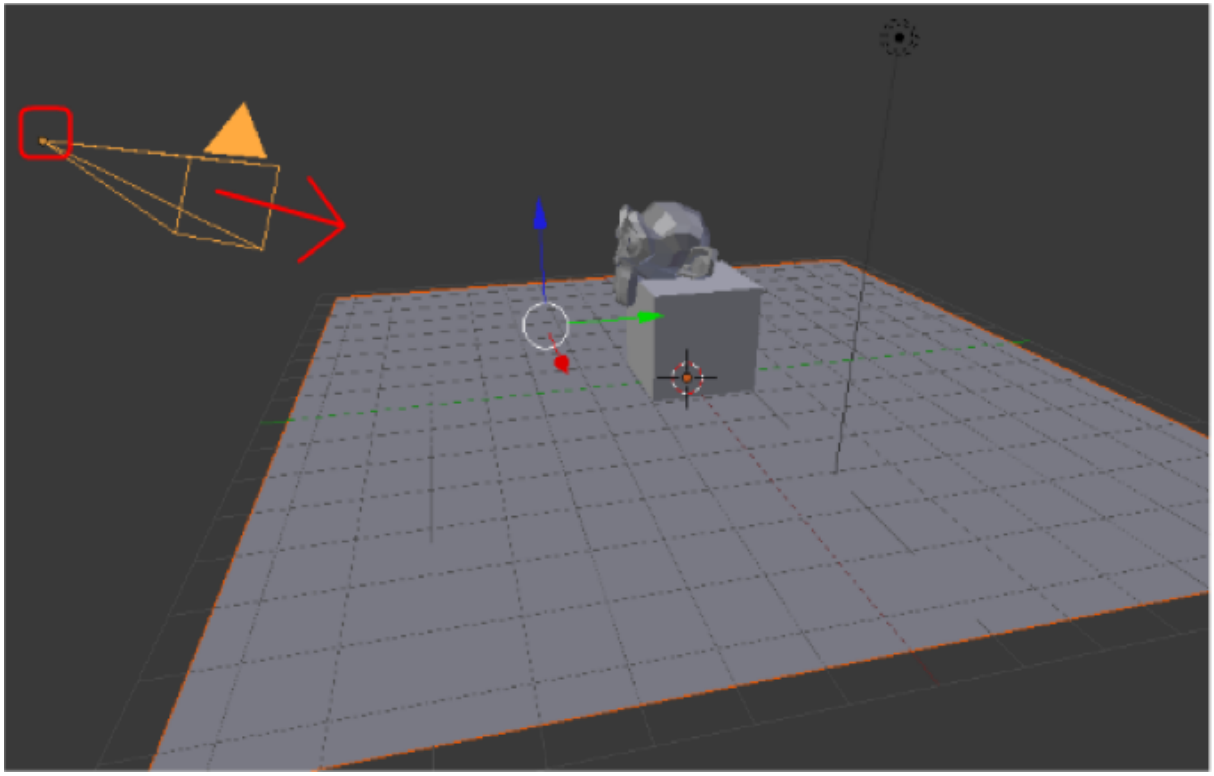
The View Matrix

With perspective, we can think the projection plane as a *camera* that looks at the scene from the center of projection.



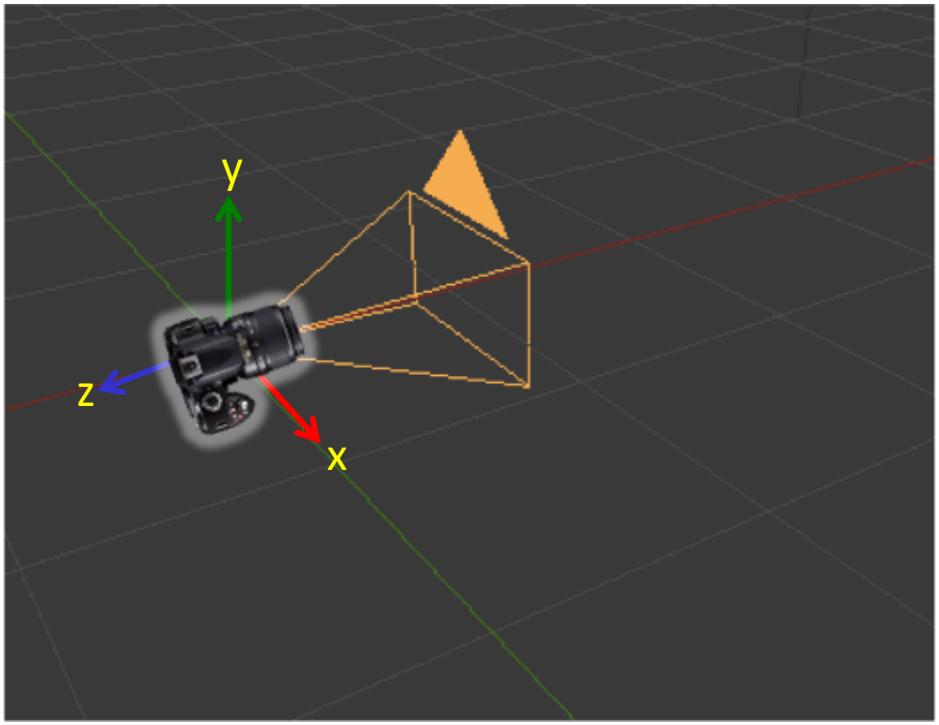
The View Matrix

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



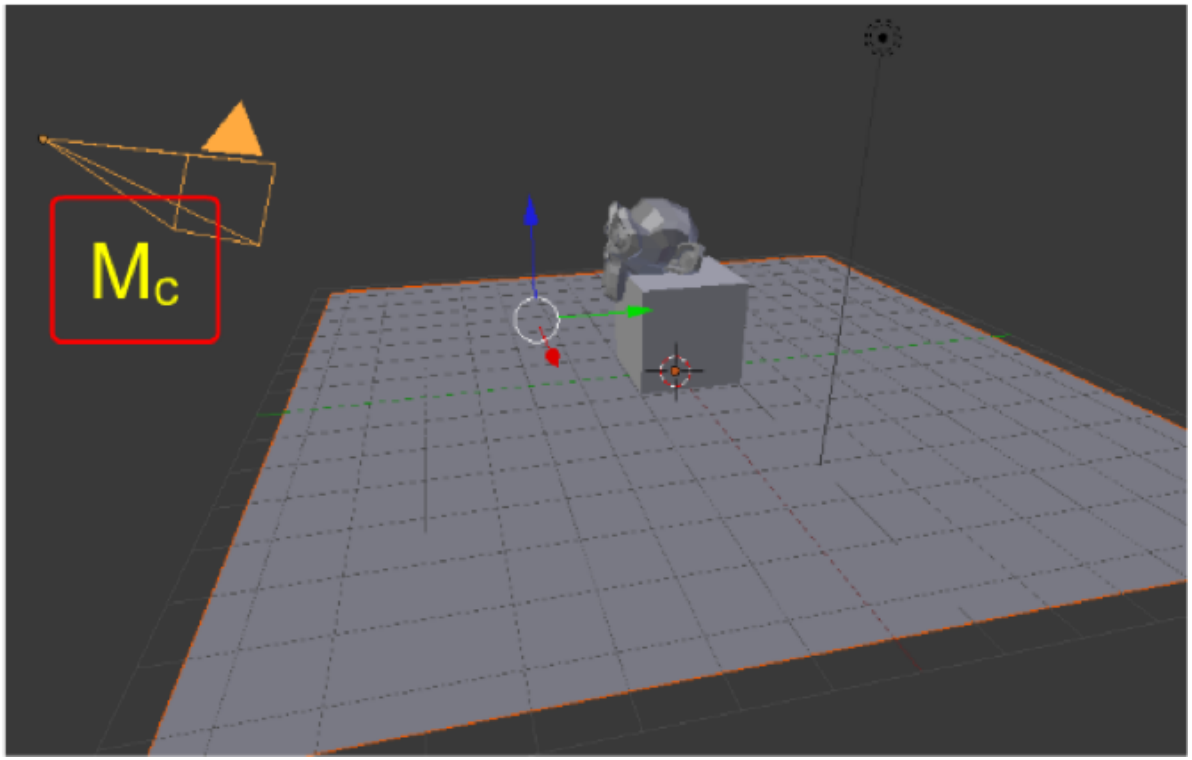
The View Matrix

The projection matrices we have seen, can 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.



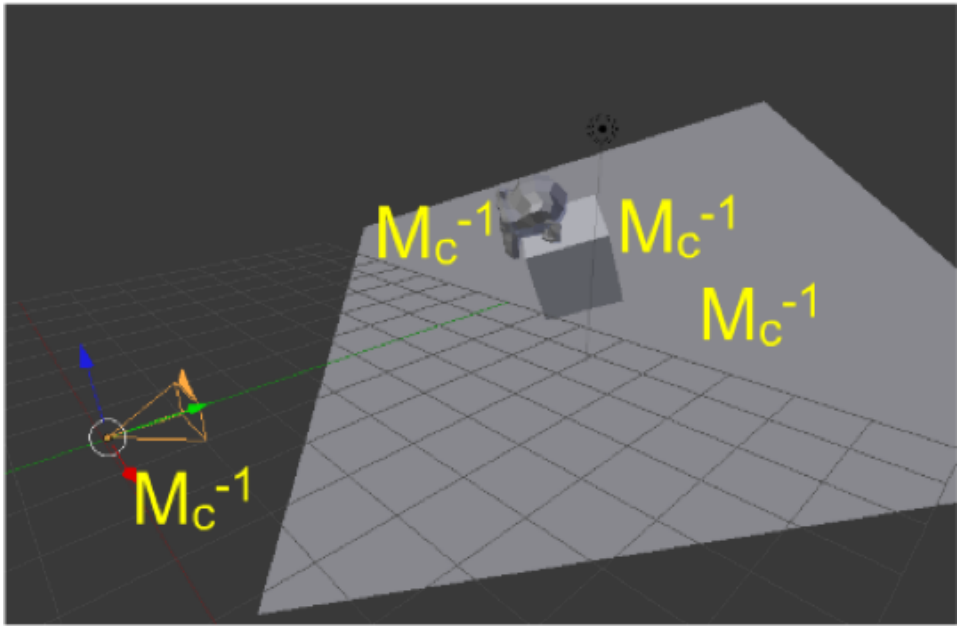
The View Matrix

We can then define a transformation matrix M_c that moves this camera object to its target position and direction. We will call this matrix the *Camera Matrix*.



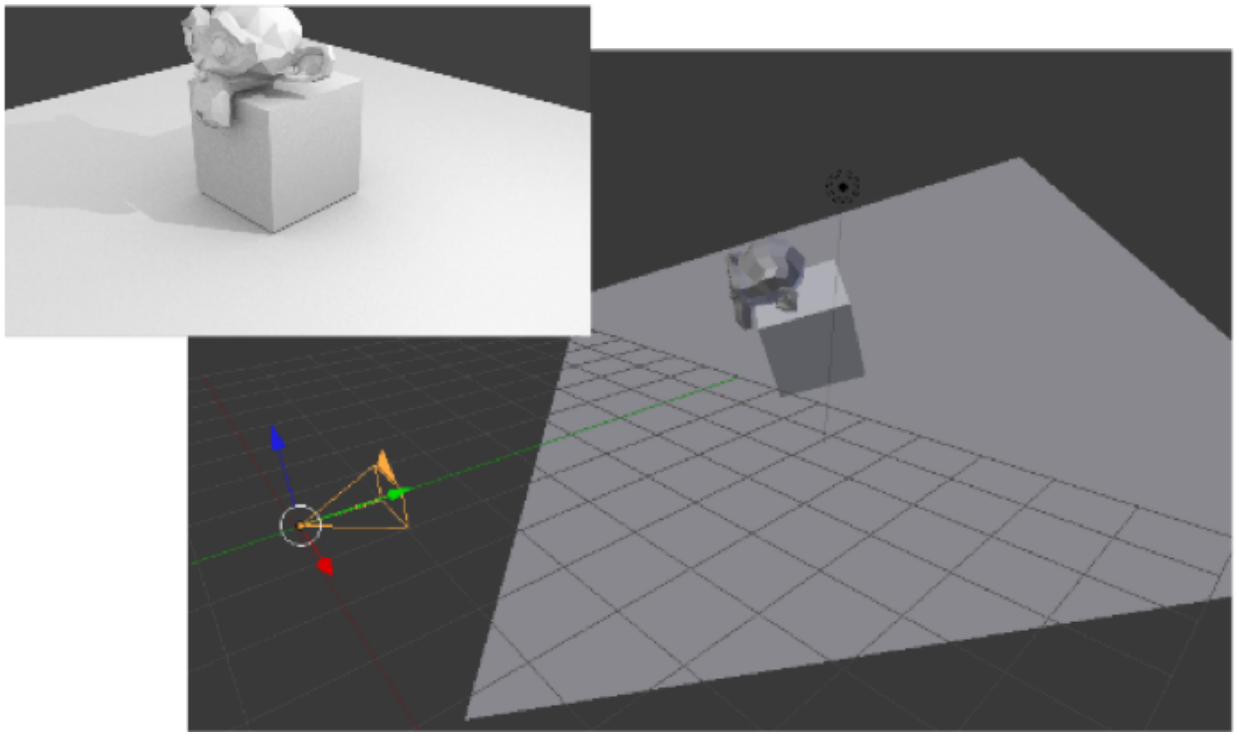
The View Matrix

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 parallel to the *xy-plane;* for perspective, the center of projection is in the origin; the projection rays (for parallel projection) or the central ray (for perspective) are aligned with the z-axis.



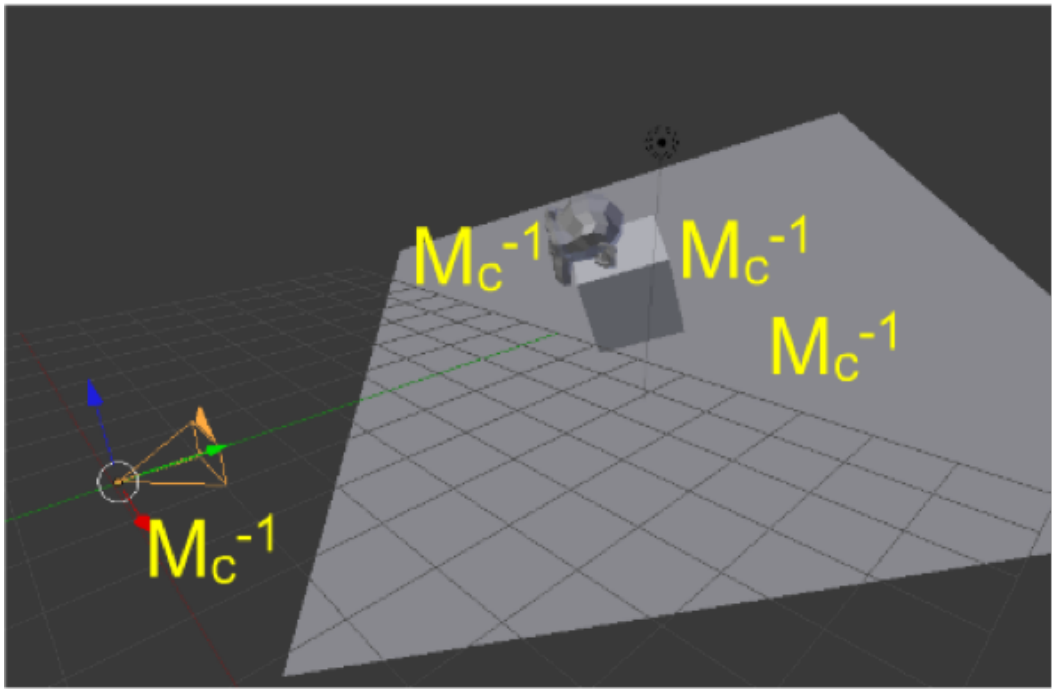
The View Matrix

In this new space, the view of the 3D world as seen from the camera can be computed by applying the projections given in the 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 us to find the normalized screen coordinates of the points in space, as seen by the considered camera.

This 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).

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* matrix
- The *look-at* matrix

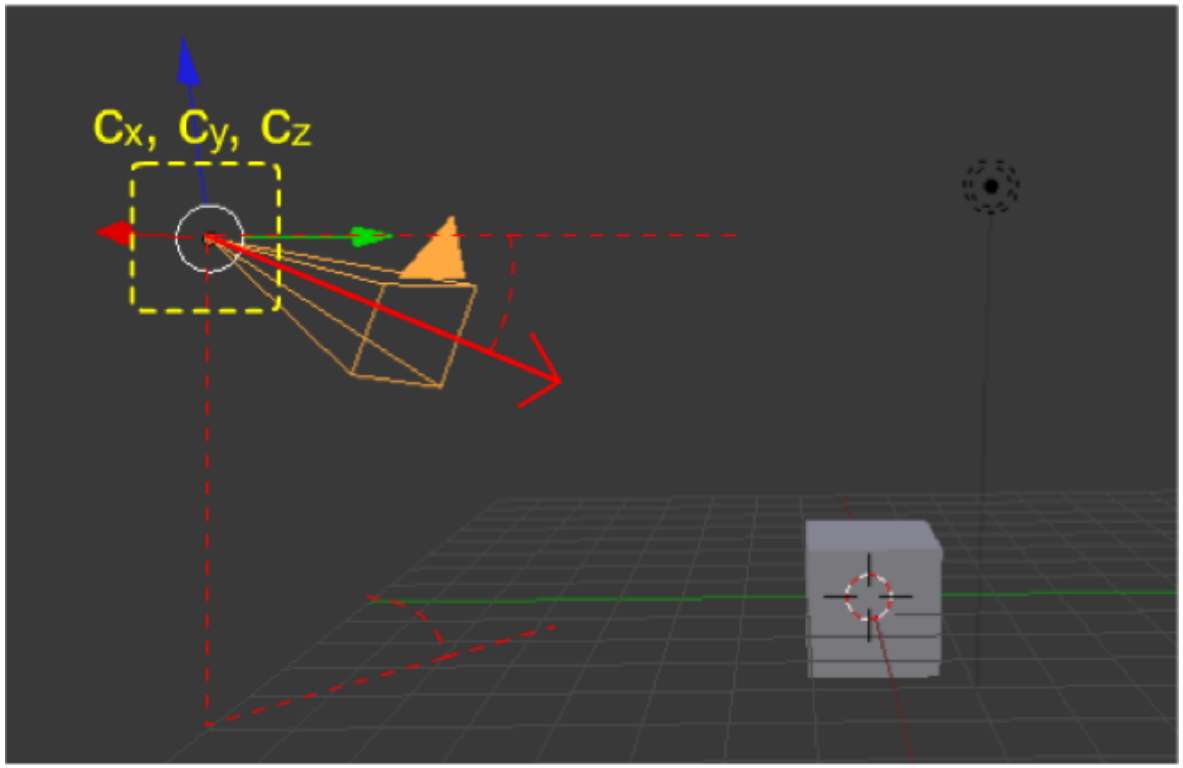
View Matrix: look-in-direction

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

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

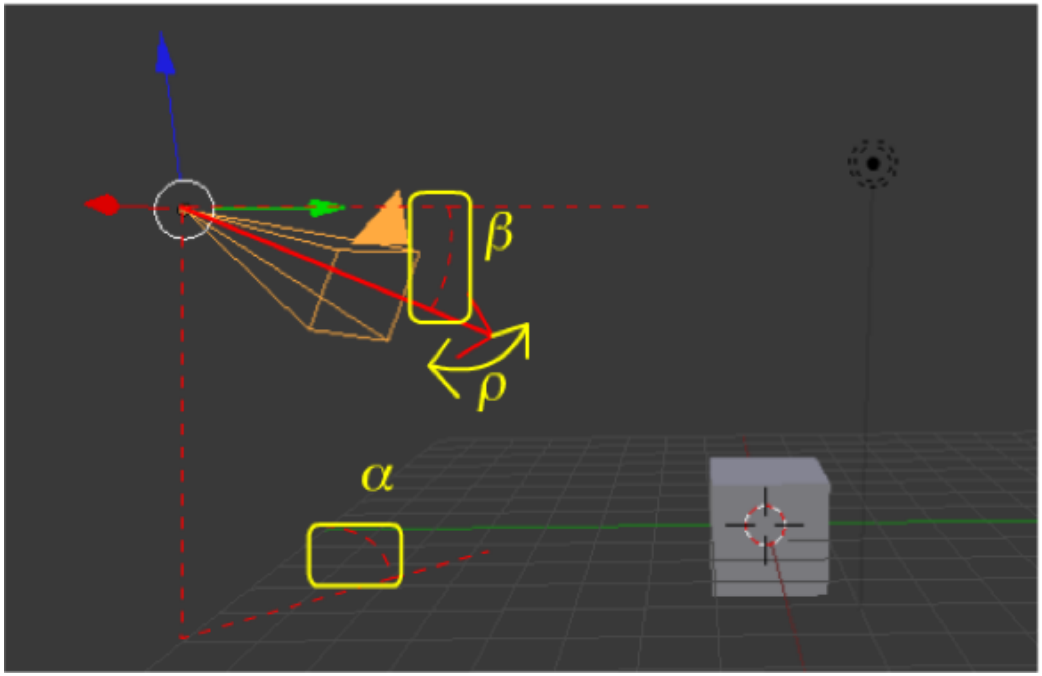
View Matrix: look-in-direction

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



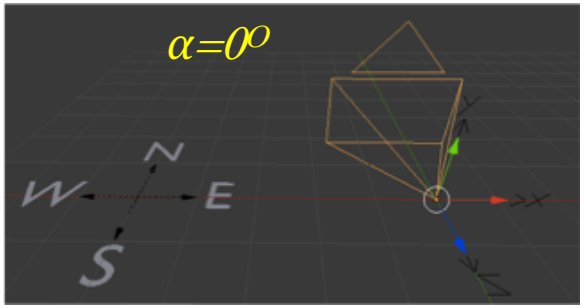
View Matrix: look-in-direction

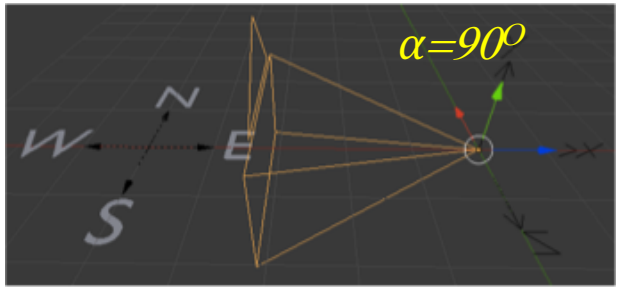
The direction where the camera is looking is specified with three angles: the “compass” direction (angle α), the elevation (angle β), and the roll over the viewing direction (angle ρ). This last parameter however is rarely used.

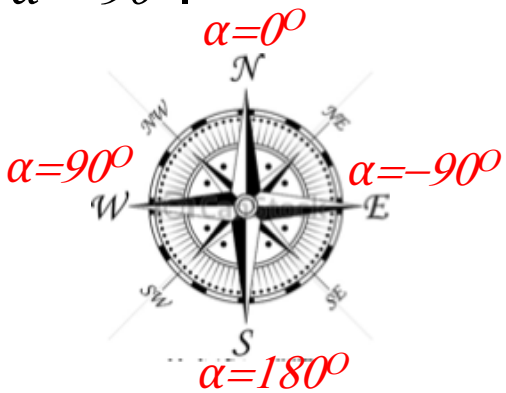


View Matrix: look-in-direction

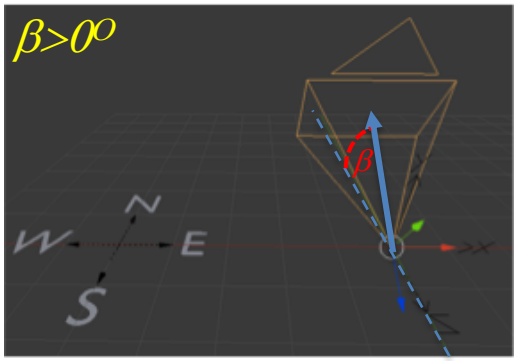
In particular, with $\alpha=0^o$ the camera looks *North*, while with $\alpha=90^o$ the camera looks *West*. *South* is $\alpha=180^o$ and *East* is $\alpha=-90^o$.



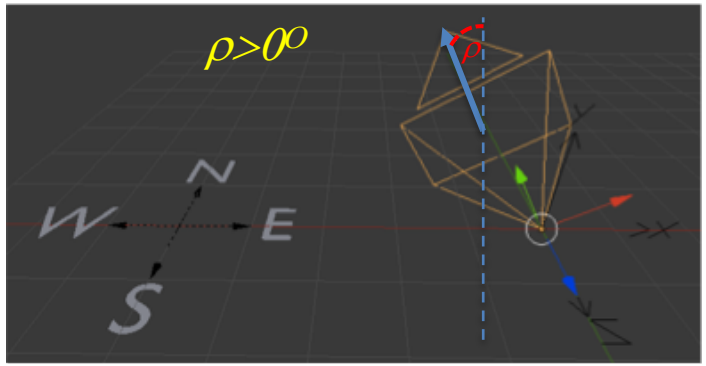




A positive angle $\beta>0^o$ makes the camera look up.

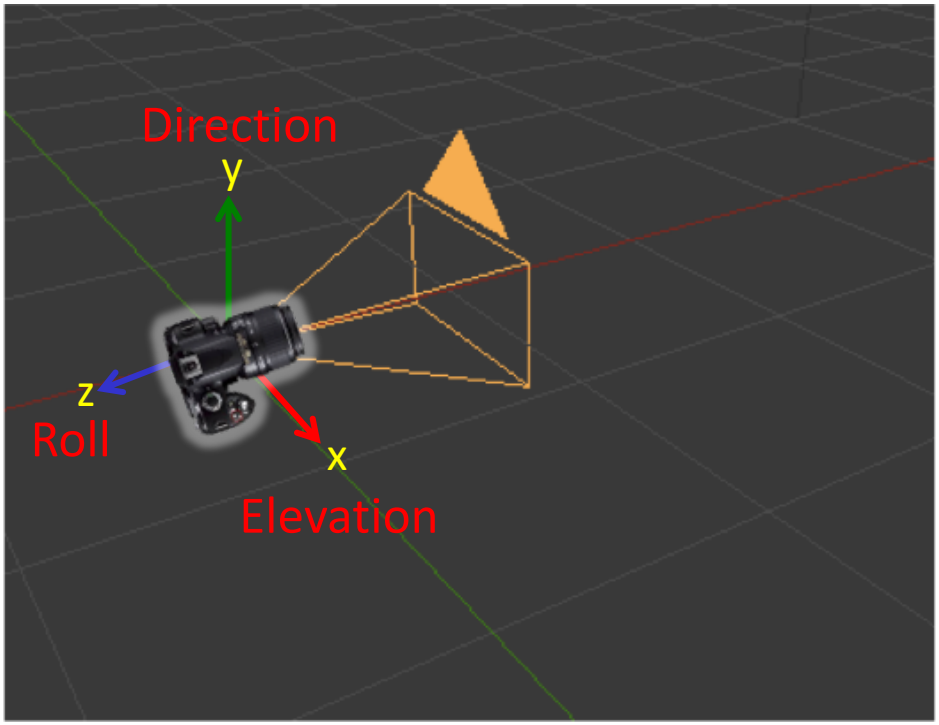


A positive angle $\rho>0^o$ turns the camera counterclockwise.



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)* with the *y-axis*.



View Matrix: look-in-direction

Rotations must be performed in a specific order (we will return on this next lesson). 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:

$$M_C = T(c_x, c_y, c_z) \cdot R_y(\alpha) \cdot R_x(\beta) \cdot R_z(\rho)$$

The *View Matrix*, is the inverse of the *Camera Matrix*.
remembering the inverse of the composition, we have:

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

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:

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

```
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));
```

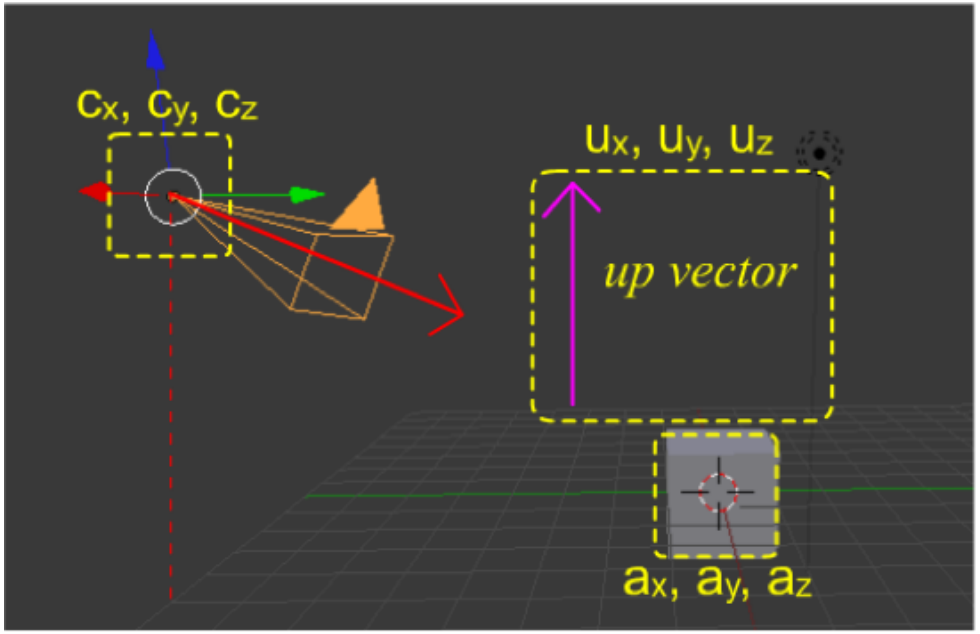
View Matrix: look-at

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.

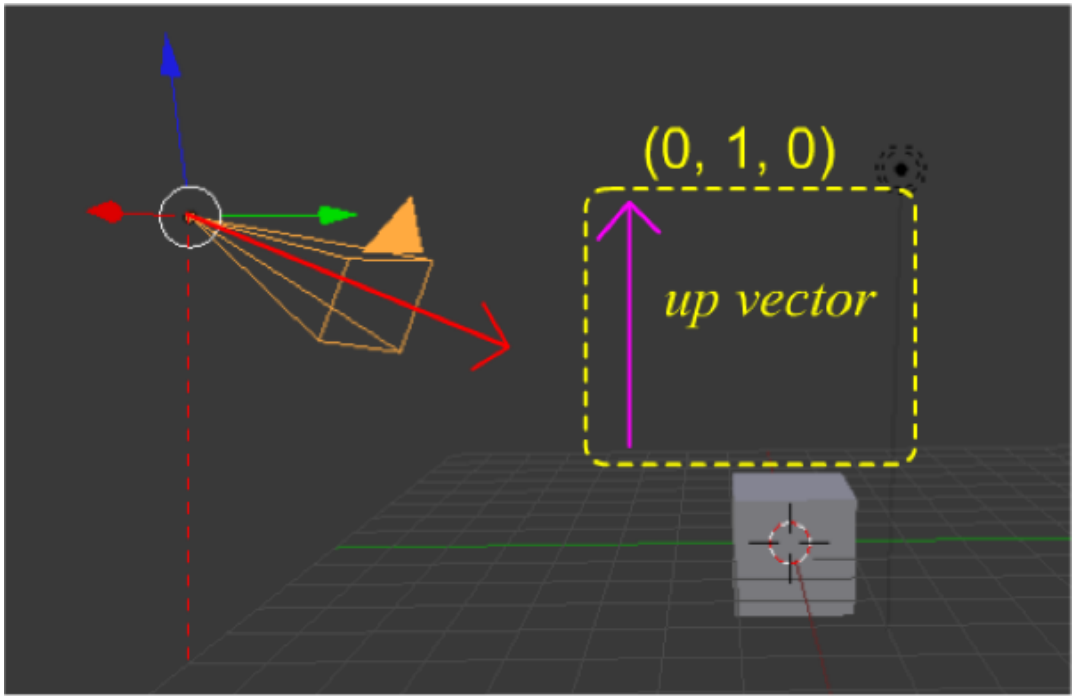
View Matrix: look-at

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 $a=(a_x, a_y, a_z)$. The technique also requires the *up vector*: the direction $u=(u_x, u_y, u_z)$ to which the vertical side of the screen is aligned.



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.



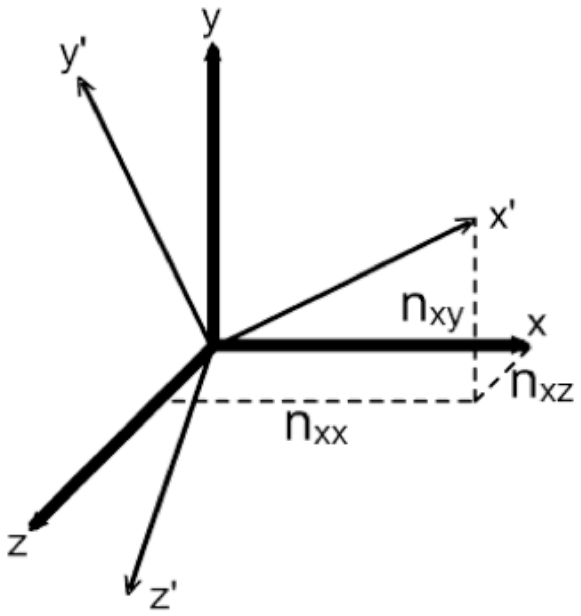
View Matrix: look-at

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*.



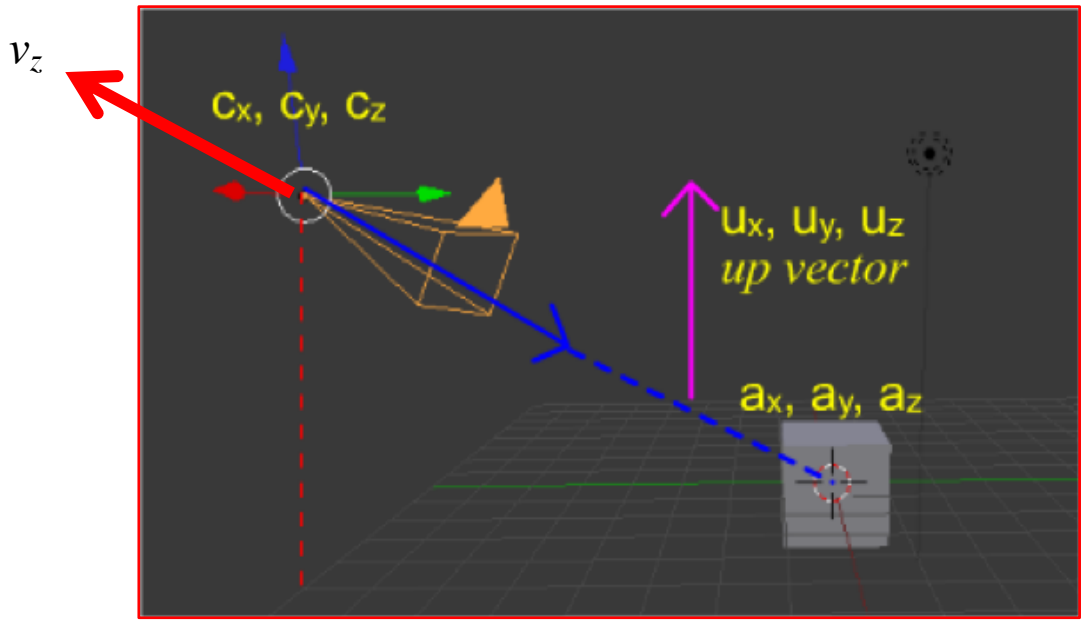
$$M_R = \begin{vmatrix} n_{xx} & n_{yx} & n_{zx} \\ n_{xy} & n_{yy} & n_{zy} \\ n_{xz} & n_{yz} & n_{zz} \end{vmatrix}$$

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.

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



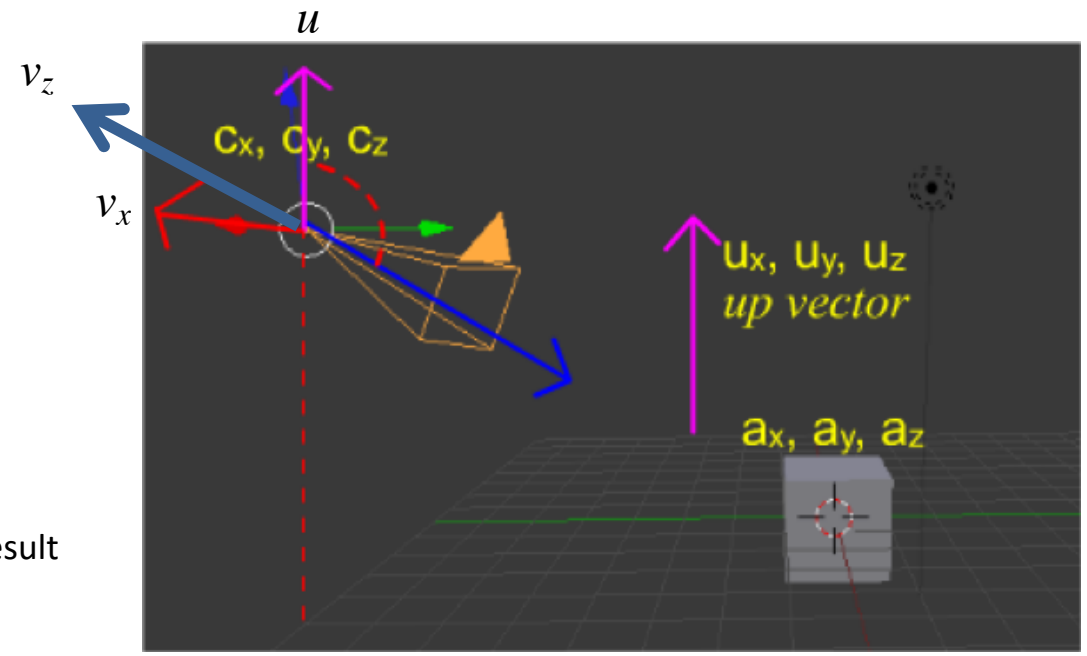
View Matrix: look-at

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 perpendicular. Without normalization, the result will be of a different 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*.

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

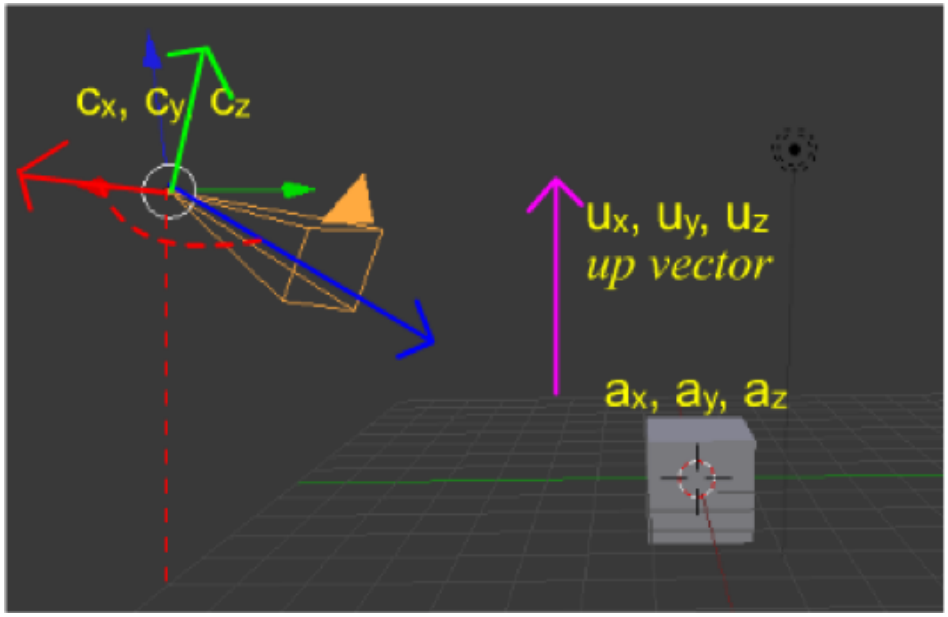
View Matrix: look-at

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 already perpendicular unit vectors, no further normalization is required.

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

$$v_y = v_z \times v_x$$



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.

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

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

$$v_y = v_z \times v_x$$

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

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:

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

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

$$v_y = v_z \times v_x$$

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

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

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(0,0,0,1));
```

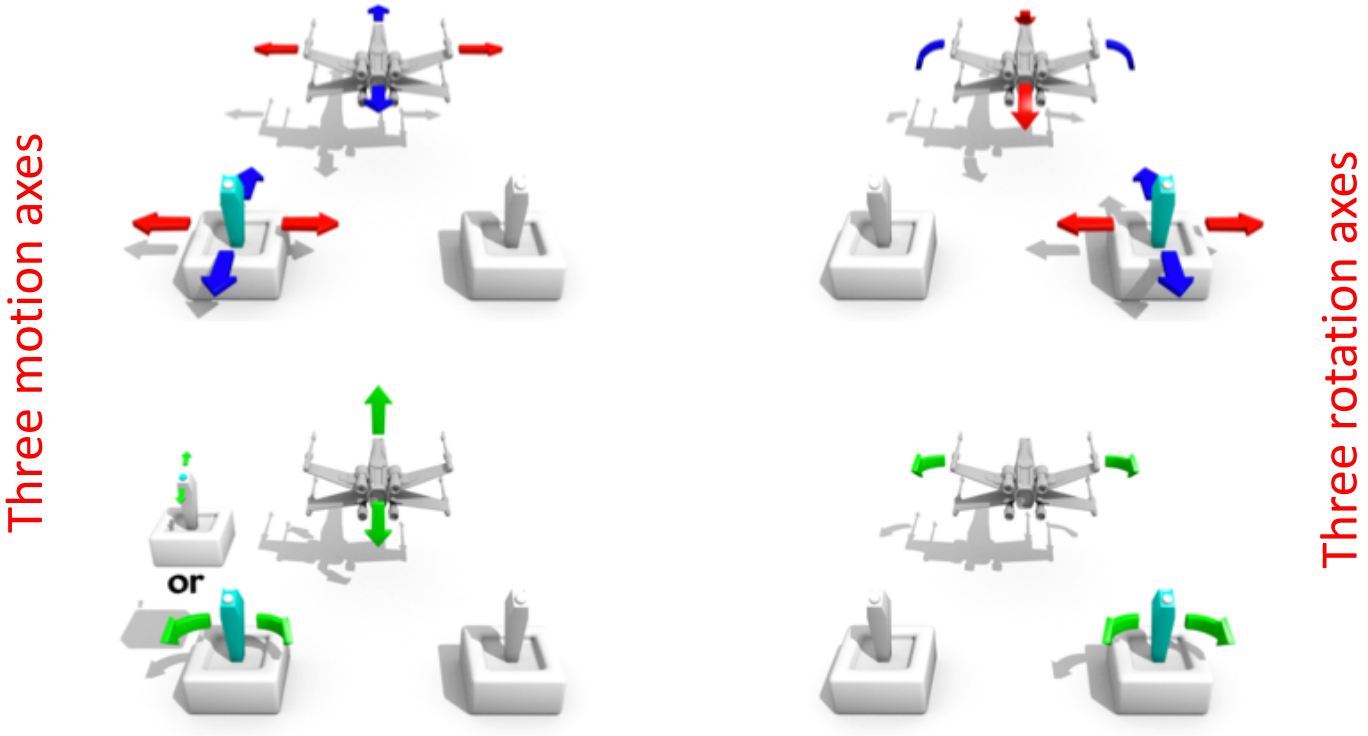
Look at matrix in GLM

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));
```

Camera navigation models

The motion of the camera is characterized by six axes:



Camera navigation models

Given the possibility of receiving input from each of these axis, there are two main navigation models:

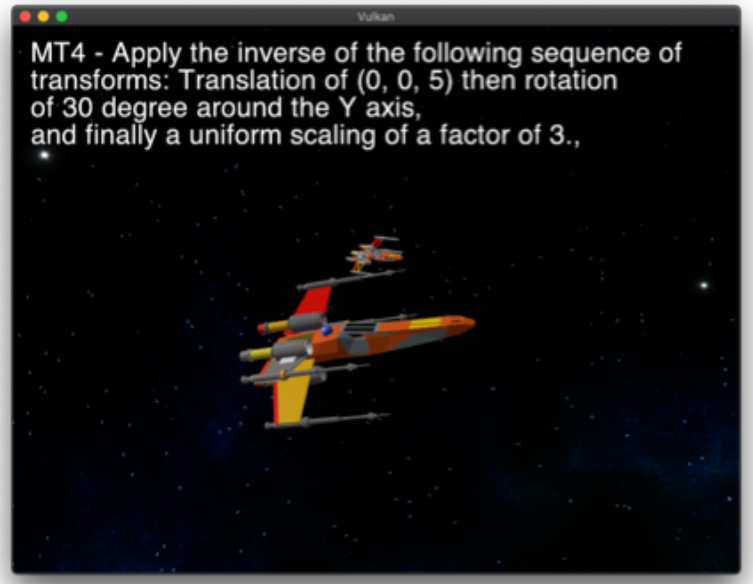
- Walk
- Fly

As the name suggests, the first works best in environments where there is a reference “ground” and some gravity that anchors the user over it, while the other is better when the viewer can fly in an open space without specific reference points.

Camera navigation models

For example, *Assignment0* uses the *Walk* model, while *Assingment03* uses the *Fly* one.

Walk model



Fly model

Camera navigation models

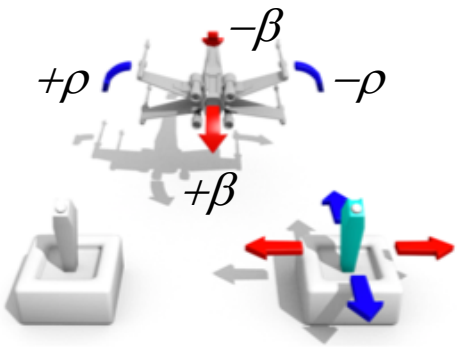
Please note that the camera navigation model is independent from the first – third person view:

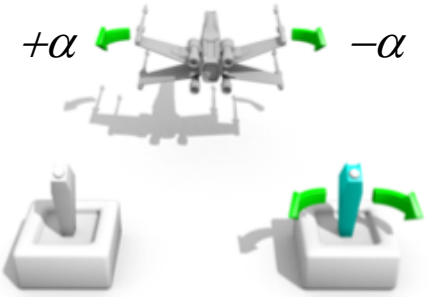
- All four combinations of camera model and person view are possible and regularly used in many applications.

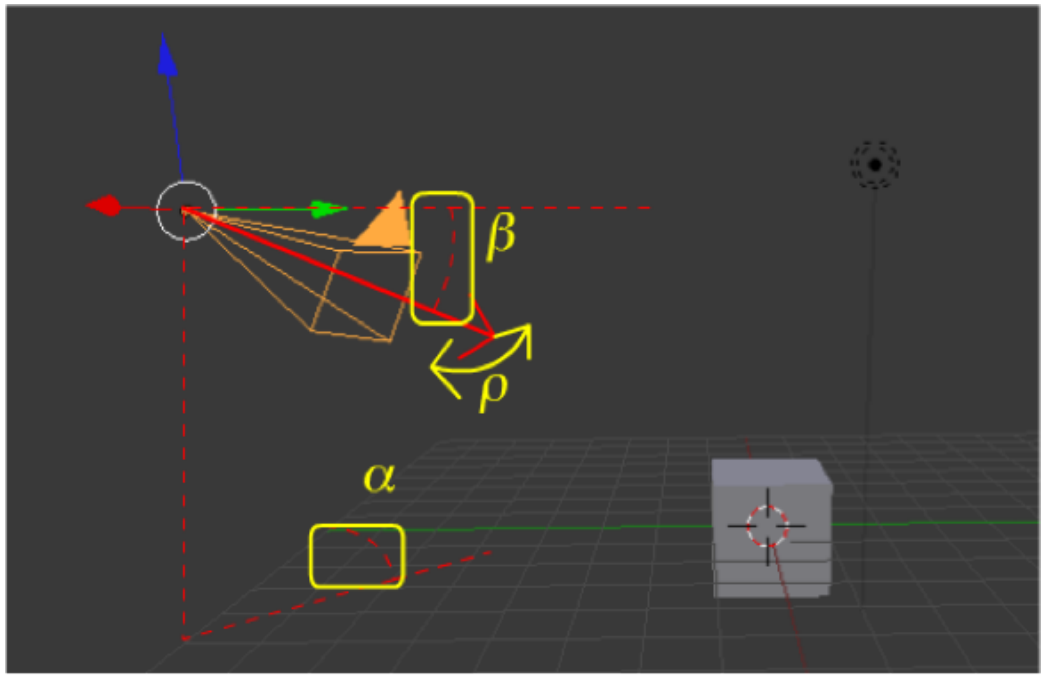
We focus only on the *look-in-direction* camera model, giving some hint on how to consider the *look-at* case.

The Walk navigation model

In the Walk navigation model, rotations around the three axis are directly mapped to the α, β and ρ parameters of the camera.







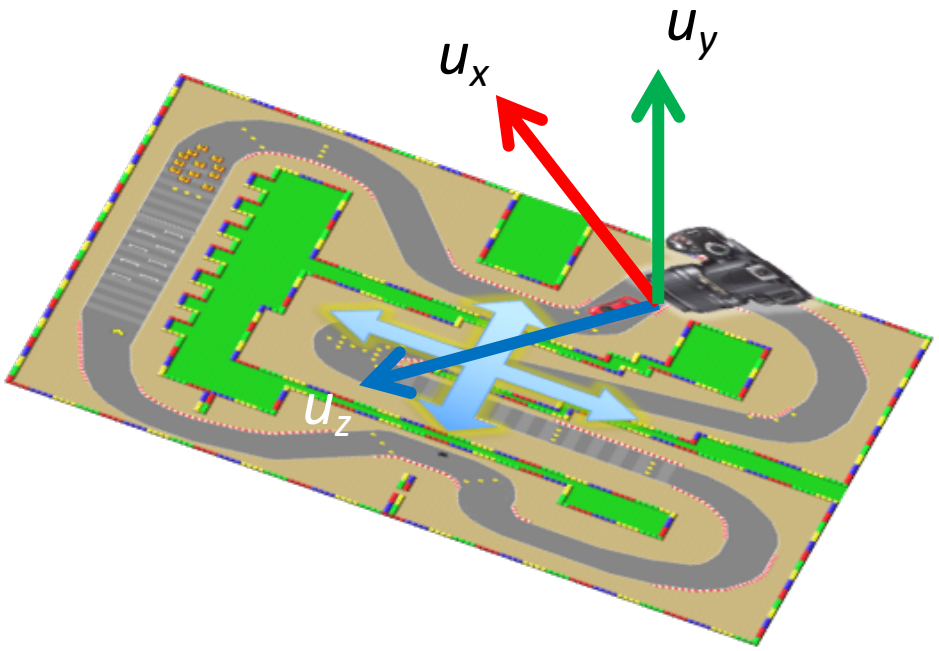
The Walk navigation model

For the motion, three vectors u_x, u_y and u_z, that represents the unitary movement in each axis, are computed.

$$u_x = \left[R_y(\alpha) \cdot |1 \quad 0 \quad 0 \quad 1|\right].xyz$$

$$u_y = |0 \quad 1 \quad 0|$$

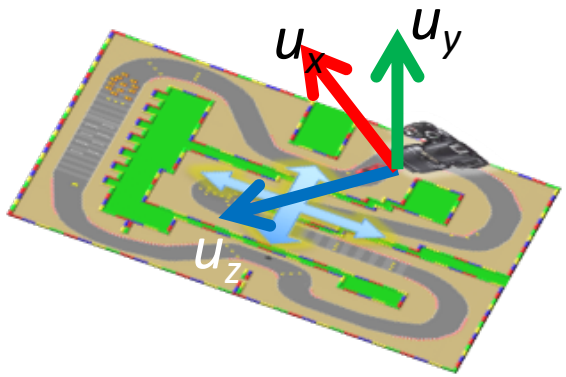
$$u_z = \left[R_y(\alpha) \cdot |0 \quad 0 \quad -1 \quad 1|\right].xyz$$

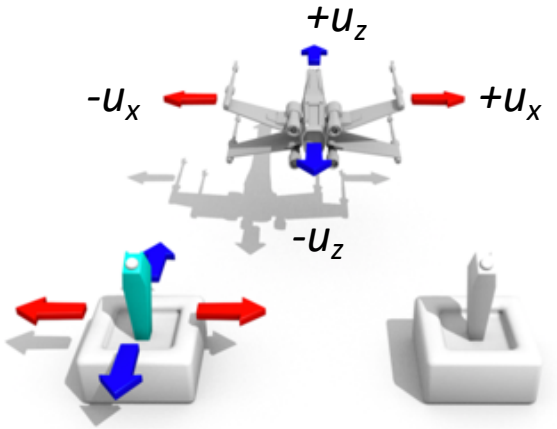


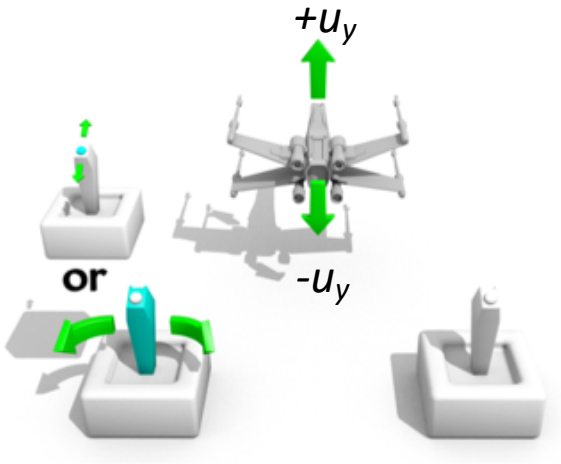
Here the *[…].xyz* notation is used to denote the cartesian coordinate corresponding the the homogeneous one.

The Walk navigation model

Motion is the performed updating the position of the center of the camera *c*, adding or subtracting one of the three vectors u_x, u_y and u_z.



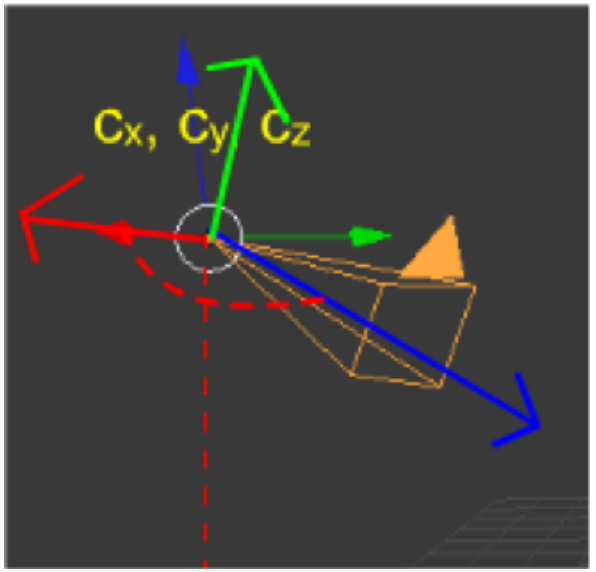




The Fly navigation model

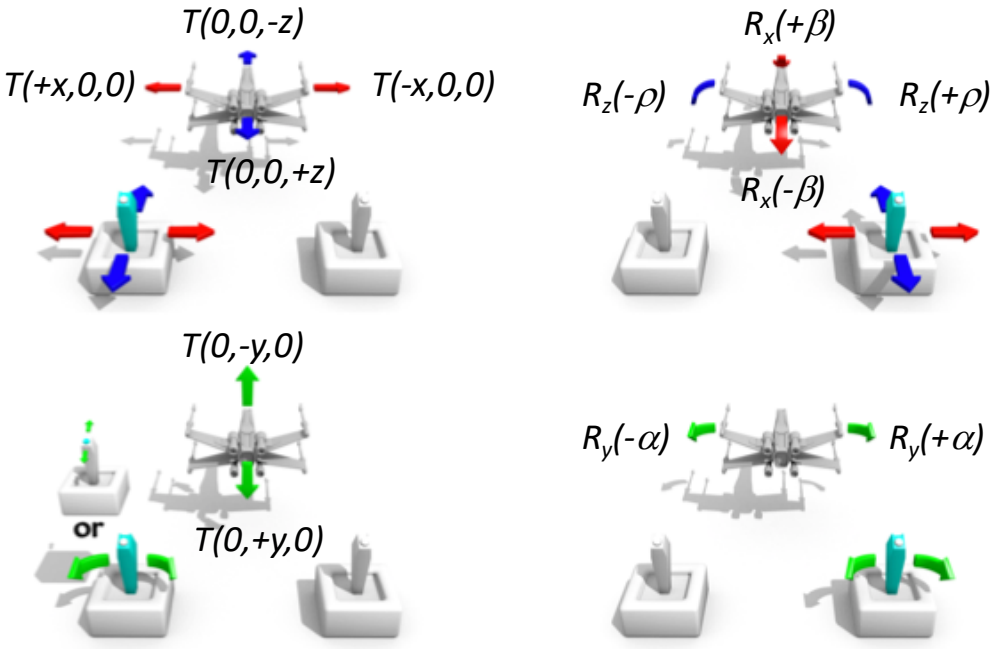
In the fly navigation models, displacements and rotations are along the axis of the camera space.

Since the view matrix brings the three camera axes along the x, y and z-axes, and the camera center in the origin, things are much simpler, and the view matrix can be updated directly.



The Fly navigation model

It is enough to either translate or rotate the view matrix in the opposite direction to the one of the movement. The opposite is required since the view matrix is the inverse of the camera matrix.



Speed

In order to properly animate the navigation a linear and an angular speed must be set:

- μ is the linear speed, expressed in world units per second. It is used to update the positions.
- ω is the angular speed, defined in radians per second. It is used to update the rotations.

More over, since updates occurs every time a frame is shown on screen, the fraction of time passed since last update *dt*, must be known. *dt* is measured in seconds.

Controls

All controls – keyboard, joysticks, gamepads and mouse – always returns a value between -1 and +1 per axis:

+1 : one direction along the axis is selected
0 : this axis is not being changed
-1 : the opposite direction along the axis is selected.

Discrete sources such as keyboard, buttons, hat-swtich or DPad return exactly one of these three value per axis.

Continuous sources such as joystick, thumbstick or mouse pointer return instead a floating point value in the range, depending on the intensity of the pressure / motion.

Controls

A navigation model update procedure receives then up to six floating point values in the [-1, 1] range:

- m_x : control along the horizontal axis for the movement
- m_y : control along the vertical axis for the movement
- m_z : control along the depth axis for the movement
- r_x : rotation control around the horizontal axis
- r_y : rotation control around the vertical axis
- r_z : rotation control around the depth axis

Controls

Examples:

Update

The update cycle for a Walk navigation model has then the following pseudo-code:

```
// external variables to hold
// the camera position
float alpha, beta, rho;
float cx, cy, cz;

...

// The Walk model update procedure
glm::mat4 ViewMatrix;

alpha += omega * rx * dt;
beta  += omega * ry * dt;
rho   += omega * rz * dt;
cx += mu * mx * dt;
cy += mu * my * dt;
cz += mu * mz * dt;

ViewMatrix = MakeLookAt(cx, cy, cz,
                        alpha, beta, rho);
```

In the Walk model, it is easier to have variables containing the position and direction of the camera, and use them to recreate a new view matrix at each frame update.

Update

The update cycle for a Fly navigation model has then the following pseudo-code:

```
// external variable to hold
// the view matrix
glm::mat4 ViewMatrix;

...

// The Fly model update proc.
ViewMatrix = glm::rotate(glm::mat4(1), omega * rx * dt,
                         glm::vec3(1, 0, 0)) * ViewMatrix;
ViewMatrix = glm::rotate(glm::mat4(1), omega * ry * dt,
                         glm::vec3(0, 1, 0)) * ViewMatrix;
ViewMatrix = glm::rotate(glm::mat4(1), omega * rz * dt,
                         glm::vec3(0, 0, 1)) * ViewMatrix;
ViewMatrix = glm::translate(glm::mat4(1), glm::vec3(
                          mu * mx * dt, mu * my * dt, mu * mz * dt));
```

Please note that (as introduced in the beginning of the course) since matrix product is not commutative, the order of transformations matters. However, in this particular case, it usually does not have a visible impact because:

1. In most practical case, only one of the six axes variables *mx*, *my*, *mz*, *rx*, *ry*, or *rz*, is different from zero at each time, making most of transformation matrices the identity matrix.
2. Rotations and displacement are always almost infinitesimal: when movements or rotations are very small, the influence of the order of transformations is less appreciable.