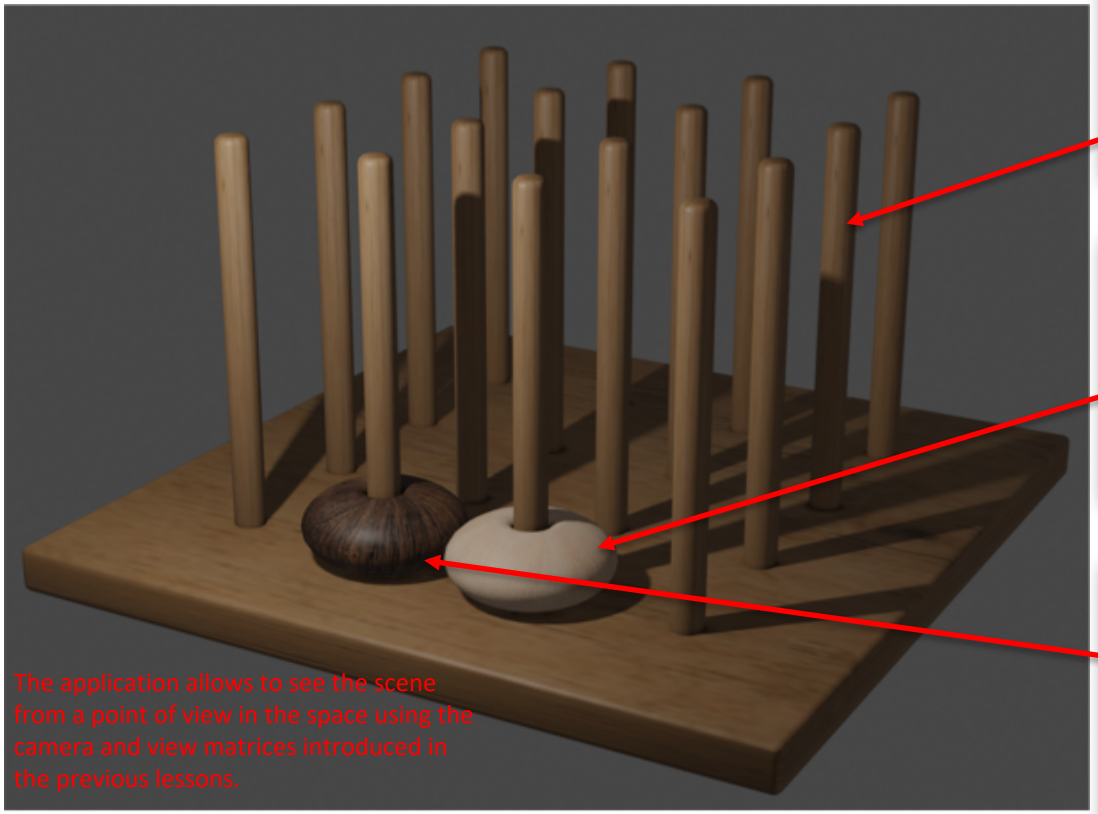
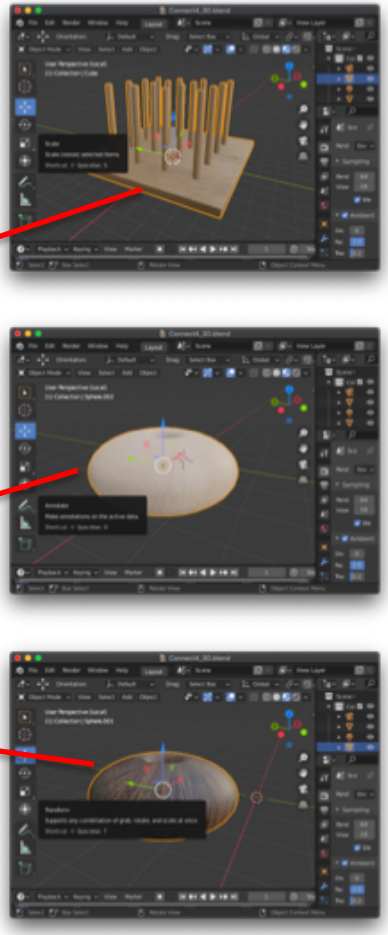
World matrix and quaternions

Application requirements

Example : a 3D tic-tac-toe application.



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.

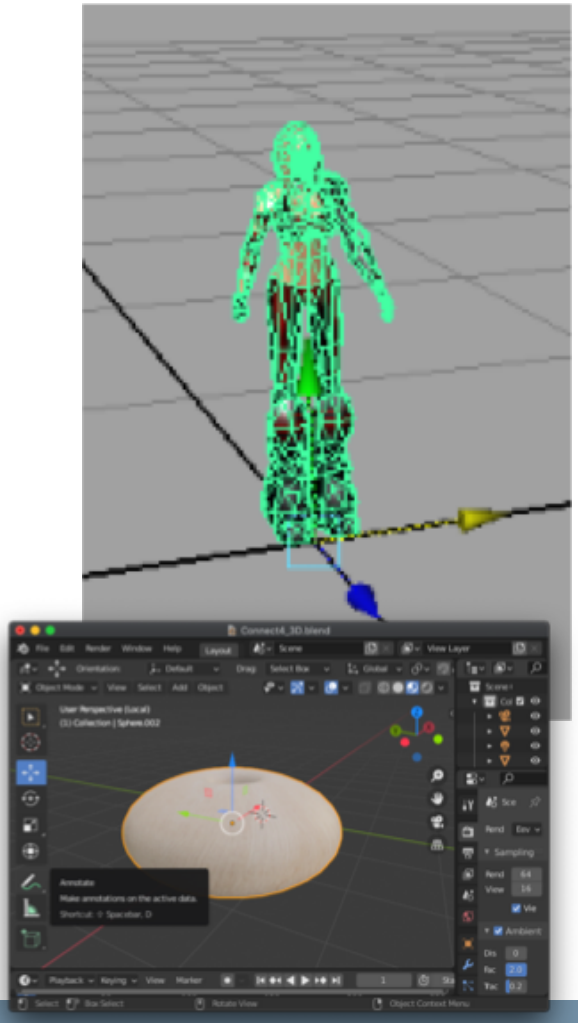
The application then replicates them and places the models in specific position to create the gameplay.

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.

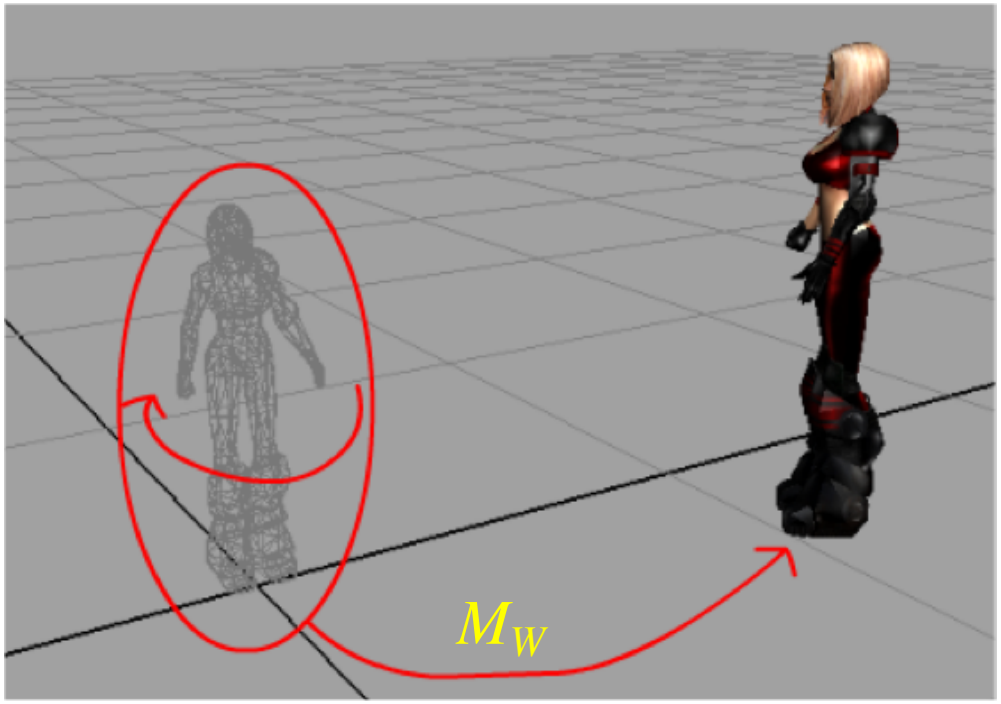
Every object is characterized by a set of *local coordinates*: the positions of the object's points in the space where it was created.

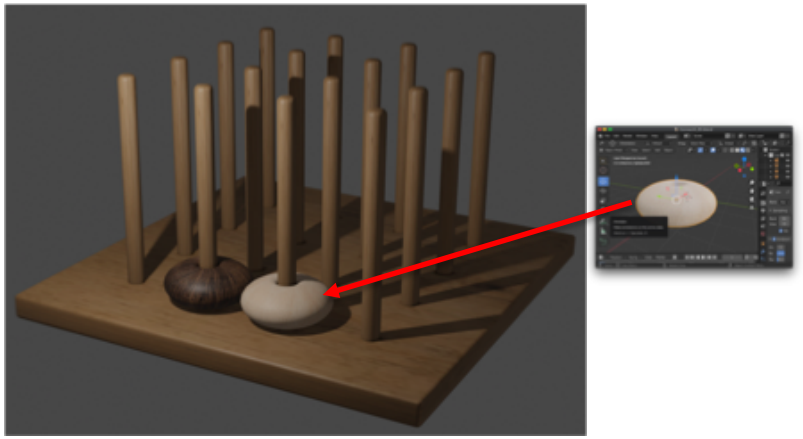


Local coordinates and World Matrix

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 and 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

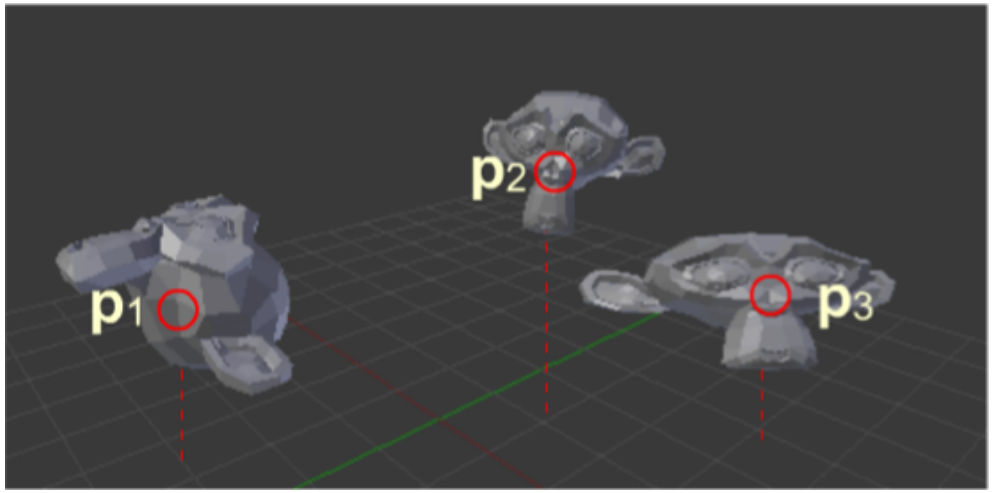
Shear and more advanced transformations, like scaling or rotations along arbitrary axes, are generally not considered since they cannot be easily generalized: should these transforms be necessary, custom procedures must be developed.

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.

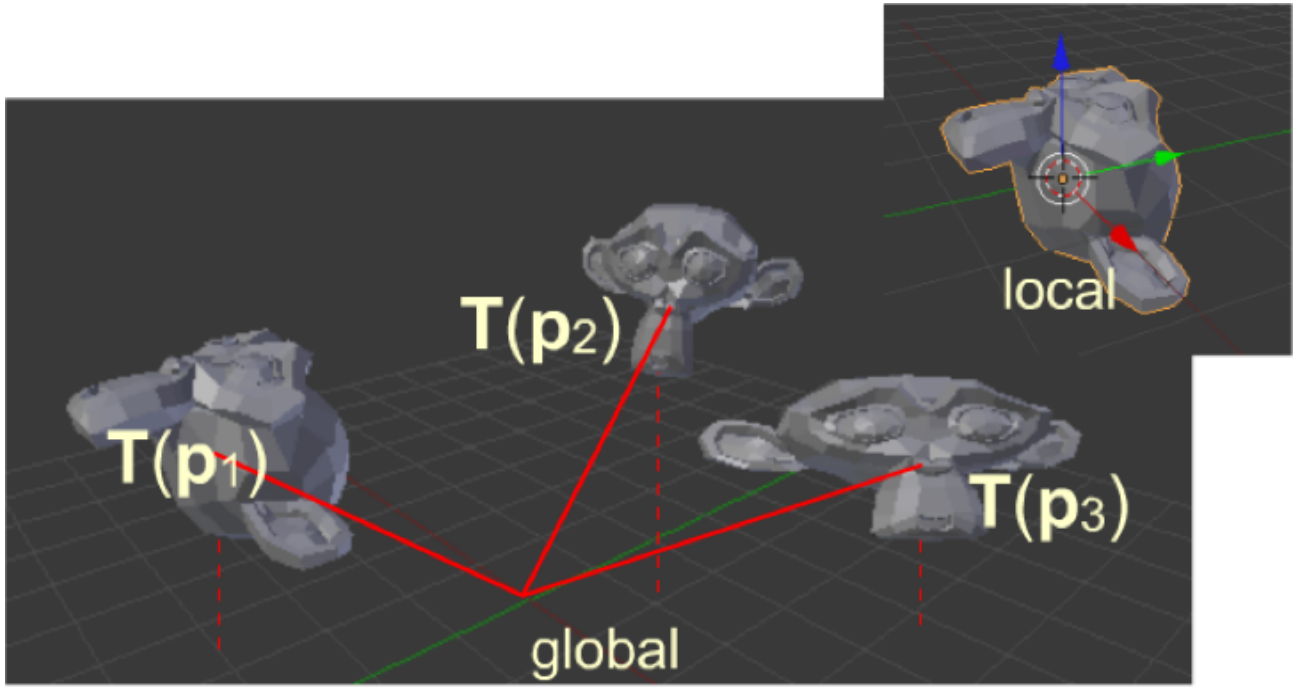
These coordinates should be independent of the size and on the orientation of the object.

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).



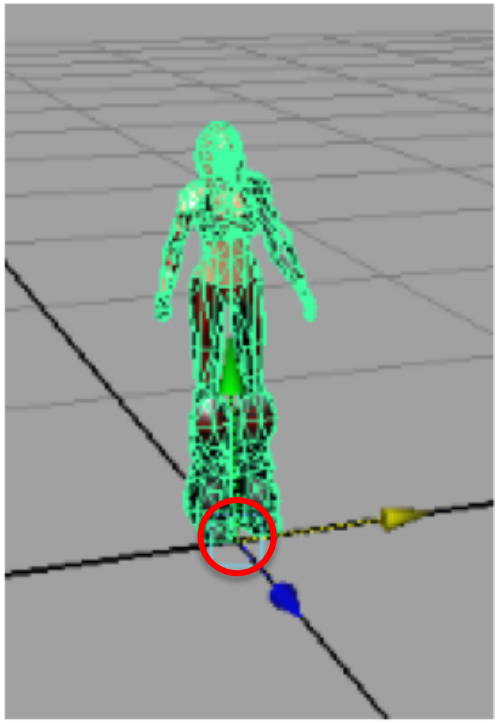
Creating a World Matrix: positioning

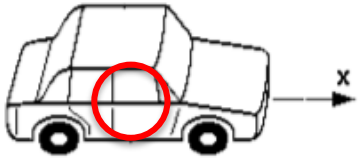
The parameters $p=(p_x,p_y,p_z)$ of the translation $T(p_x,p_y,p_z)$ defines the position that the origin of the object (in its local space) will occupy after being moved in the world space.

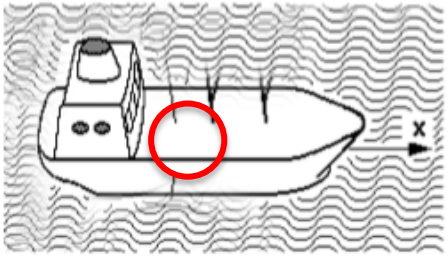


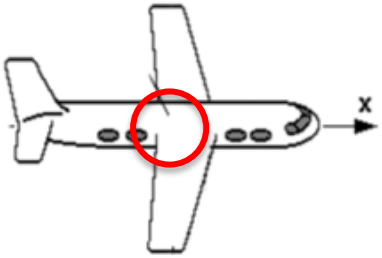
Euler angles

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





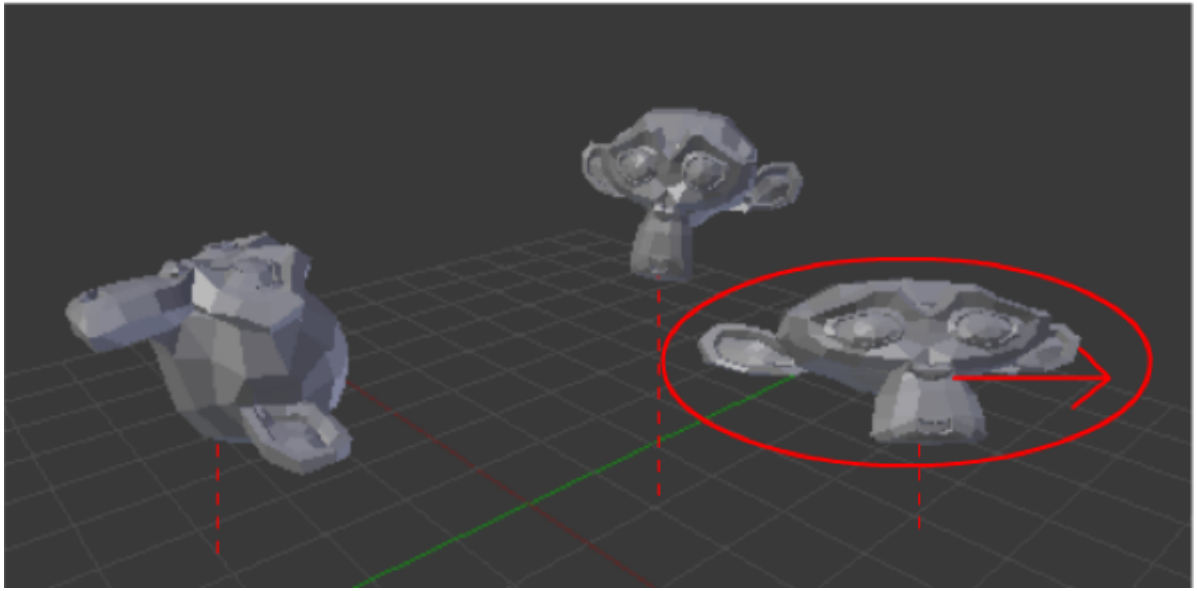




Creating a World Matrix: scaling

If the object should be scaled not proportionally (or mirrored) of s_x, s_y, s_z, rotations must be applied after, otherwise scaling direction would be oriented differently.

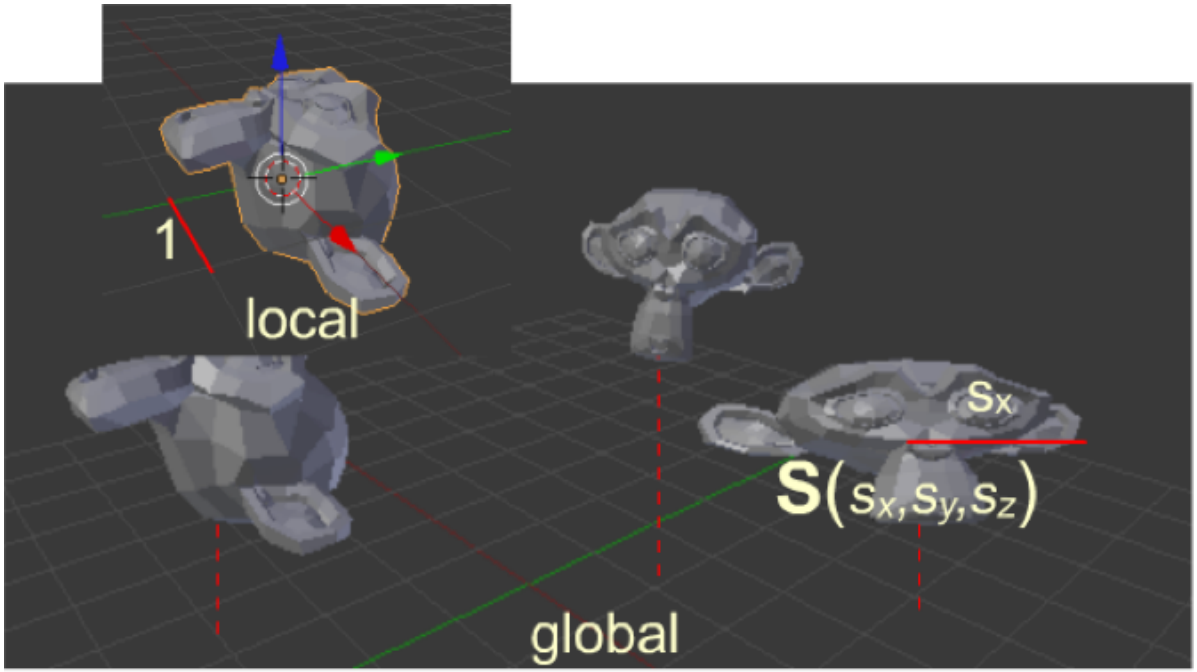
Scaling of s_x, s_y, s_z must then be performed by a transformation $S(s_x, s_y, s_z)$ that is applied **first**.



Creating a World Matrix: scaling

The parameters of the scaling convert the local units to the global units.

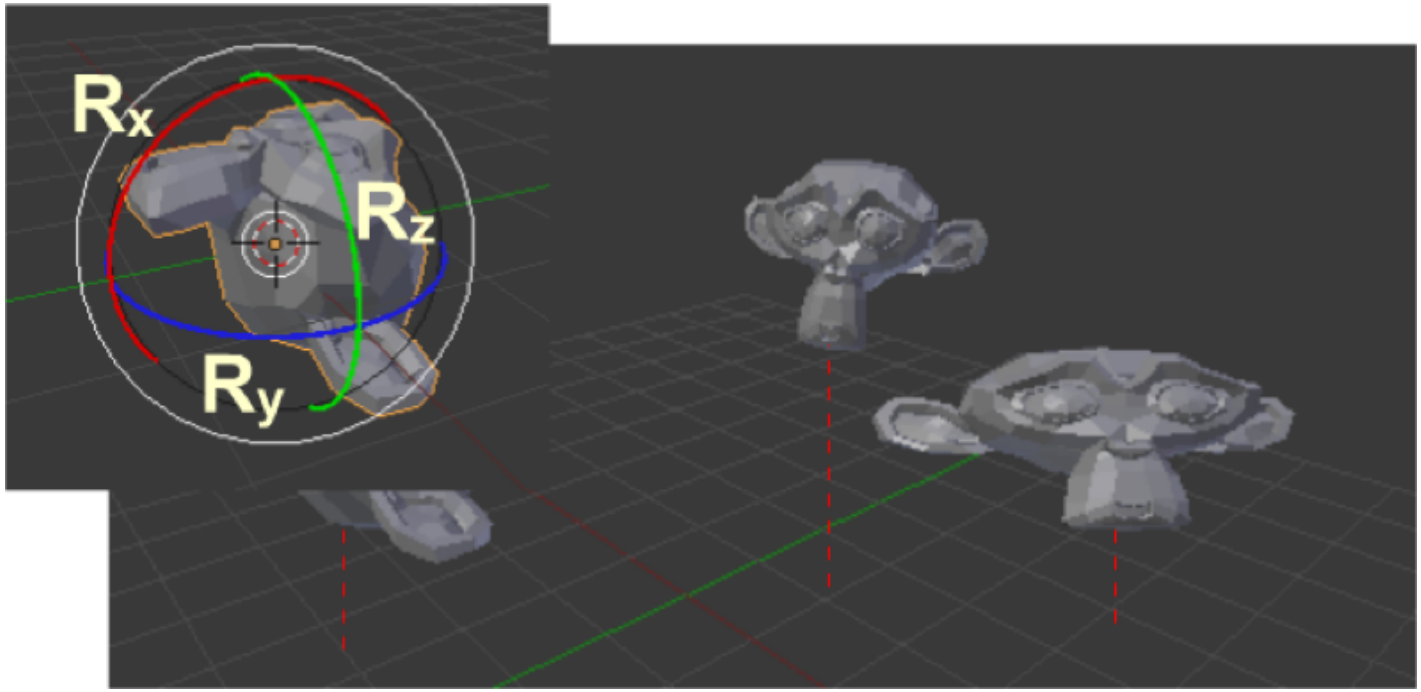
In particular a scaling factor s_x means that a unitary segment in local coordinates (along the *x-axis*), becomes s_x units in global coordinates. The same applies also to factors s_y and s_z.



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 by the user required to define the direction of an object.

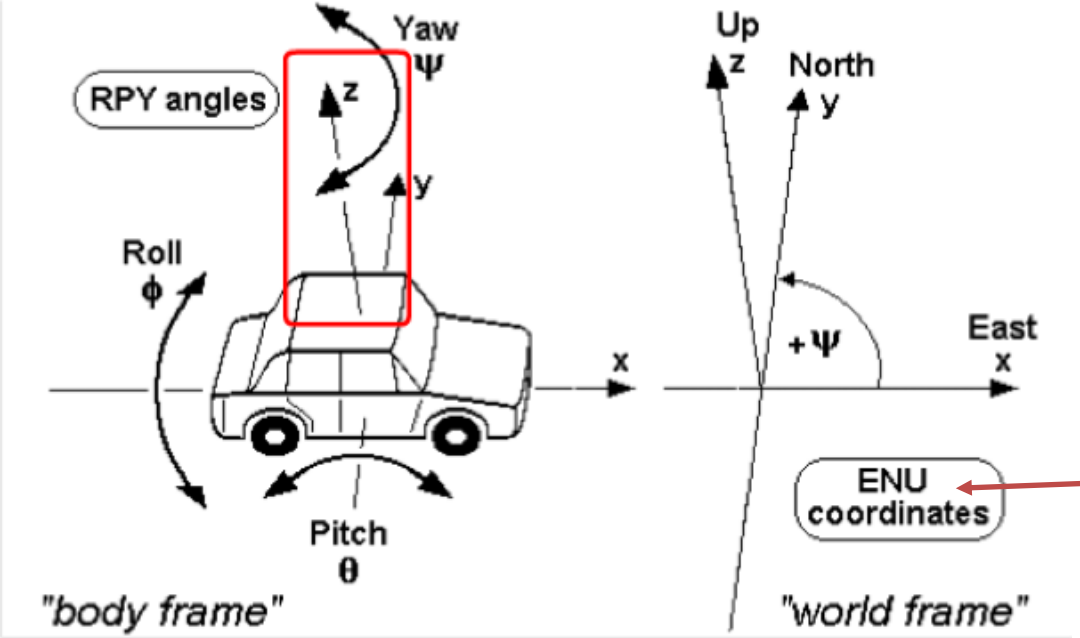
The *Euler Angles* are three parameters that define the orientation of an object in space, and they are usually addressed as:

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

Euler angles

Euler angles were introduced for *z-up* coordinates systems, and are much easier to remember in that case.

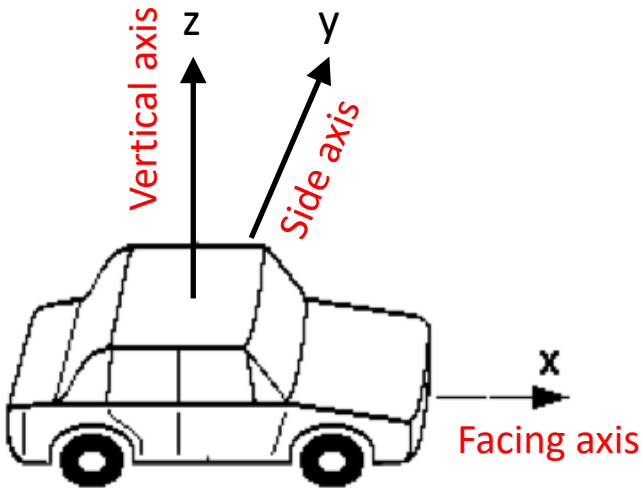
We will initially explain them in a *z-up* coordinate system: in this case the World's North will be aligned with the positive *y-axis*.



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

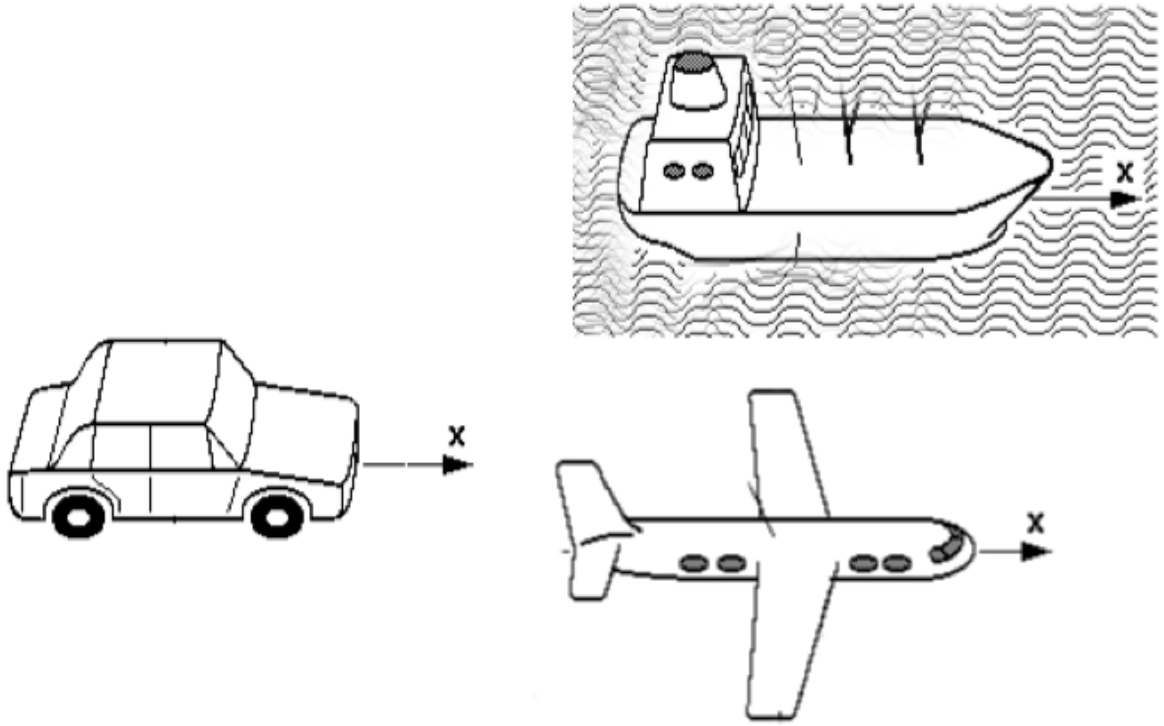
Euler angles

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



Euler angles

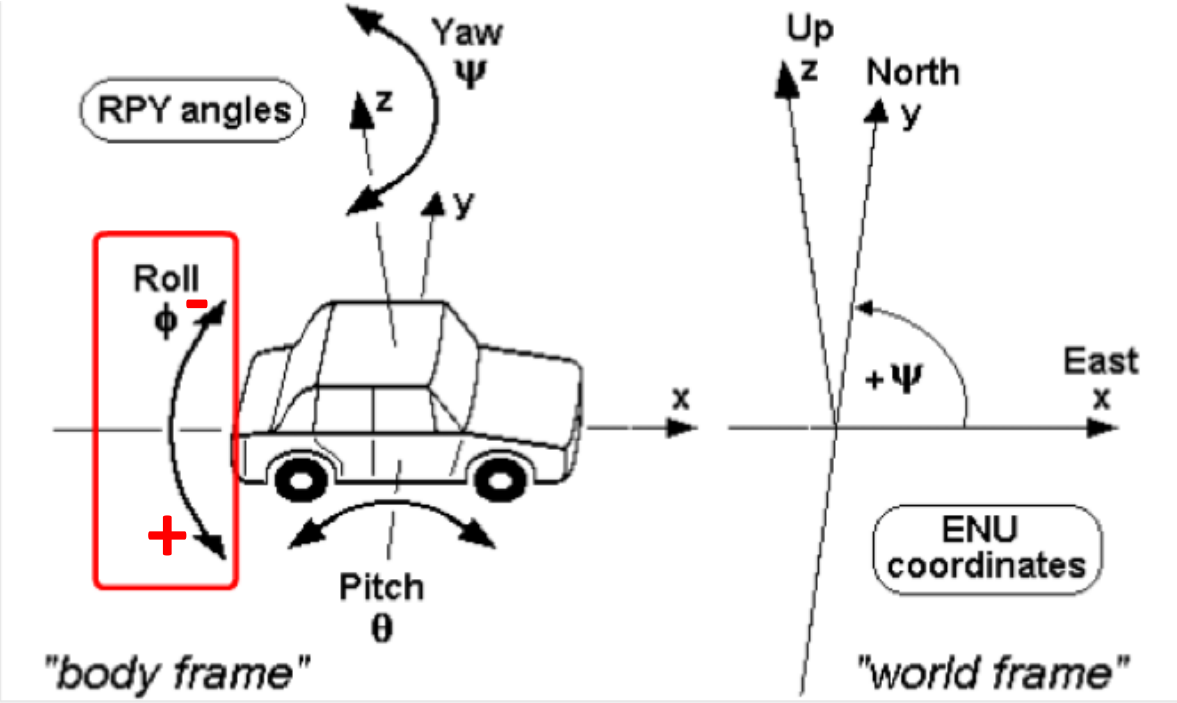
The facing axis can be for example where the car has its front view, or where an airplane or a ship is directed.



Euler angles

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

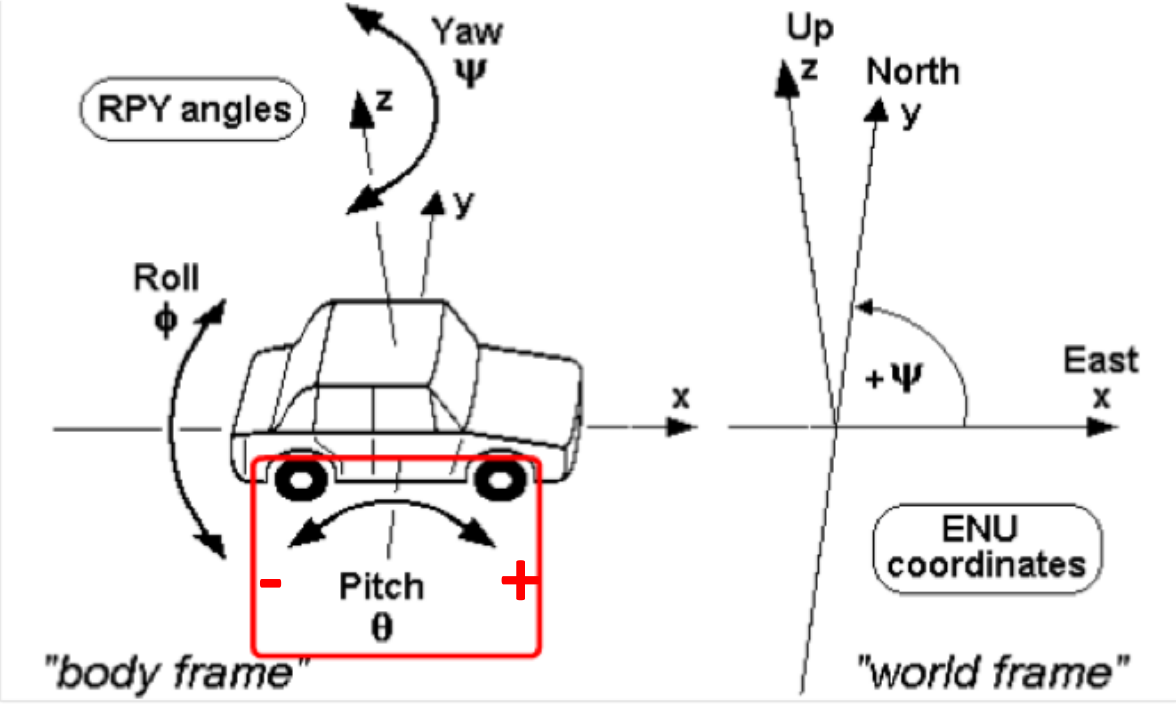
A positive roll, turns the object *clockwise* in the direction it is facing.



Euler angles

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

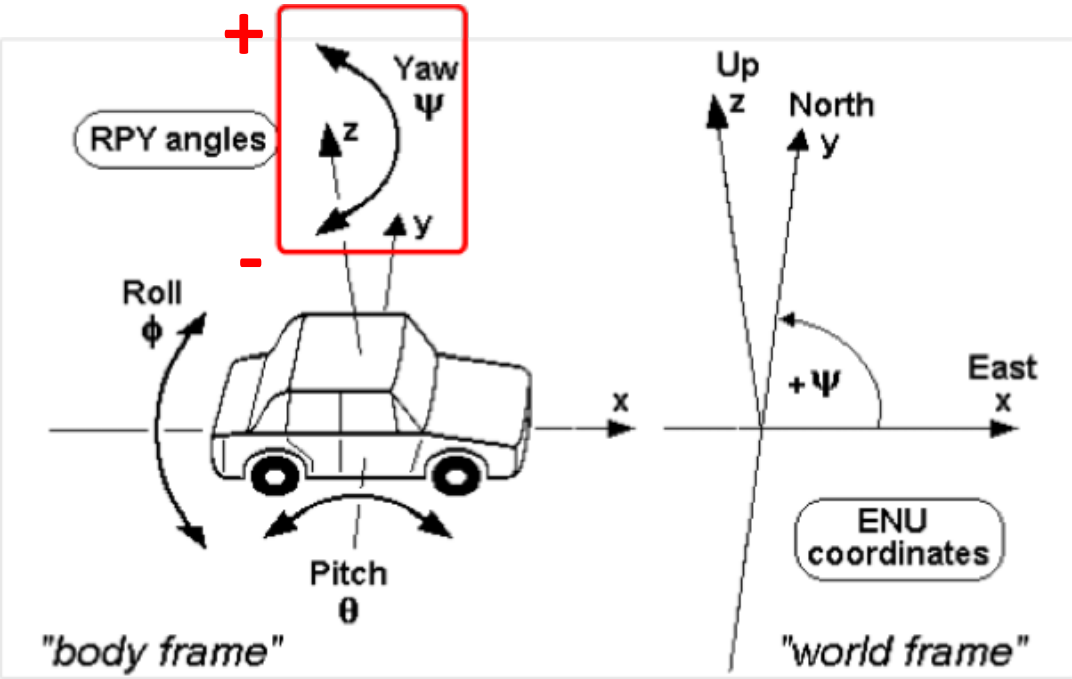
A positive pitch turns the head of the object *facing down*.



Euler angles

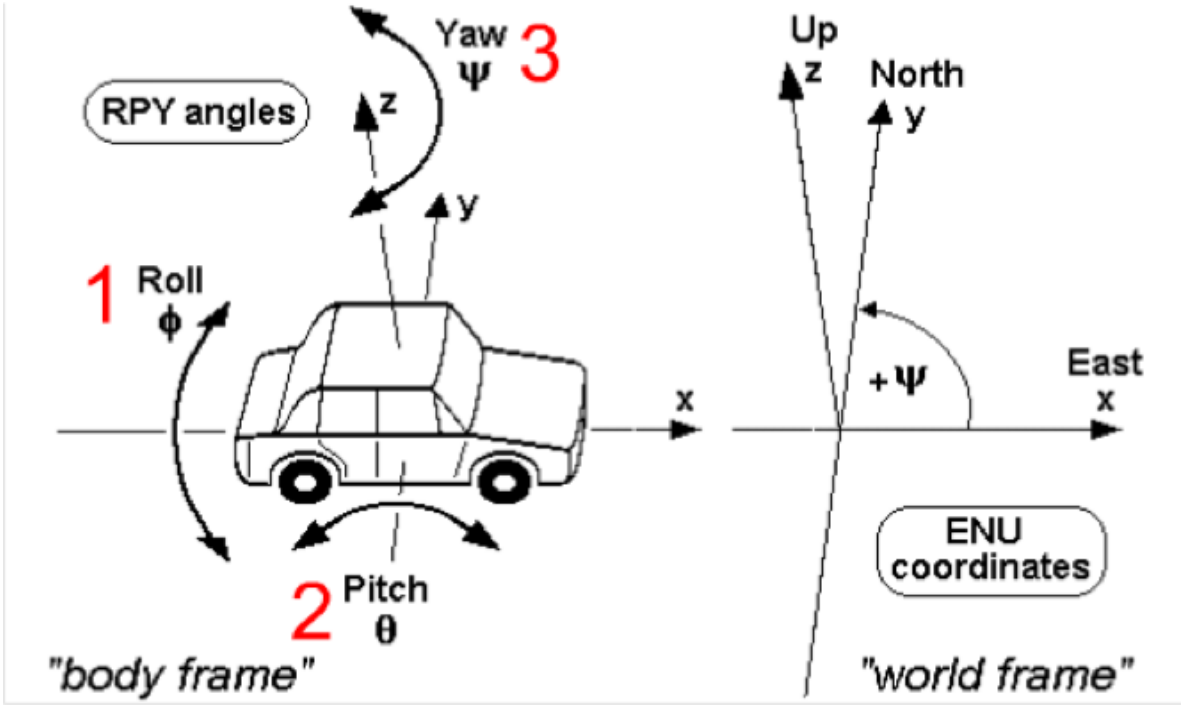
The *Yaw* ψ defines the direction of the object, and corresponds to a rotation along the *z-axis* (the *vertical axis*).

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*.

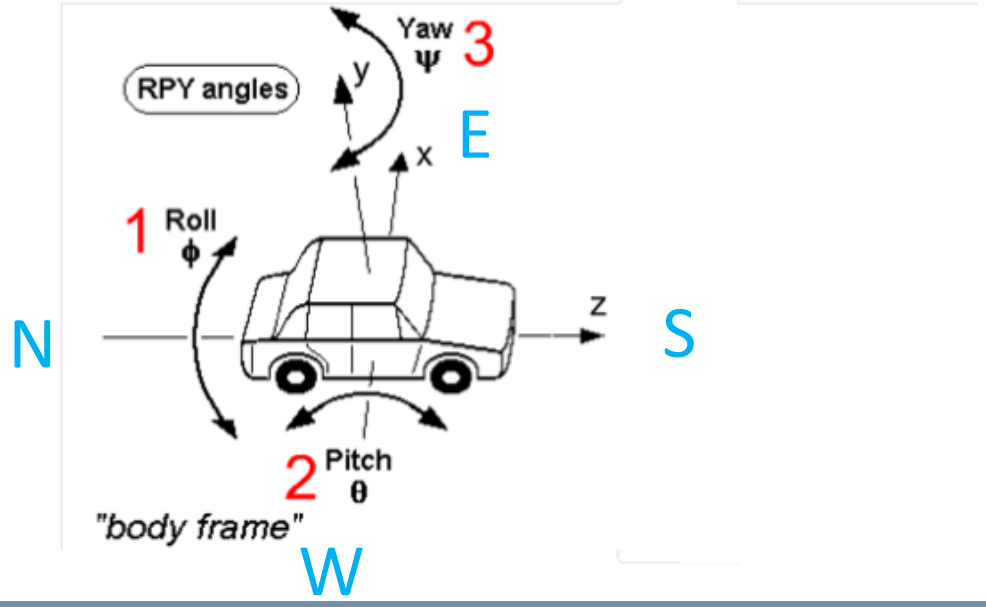


Euler angles

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

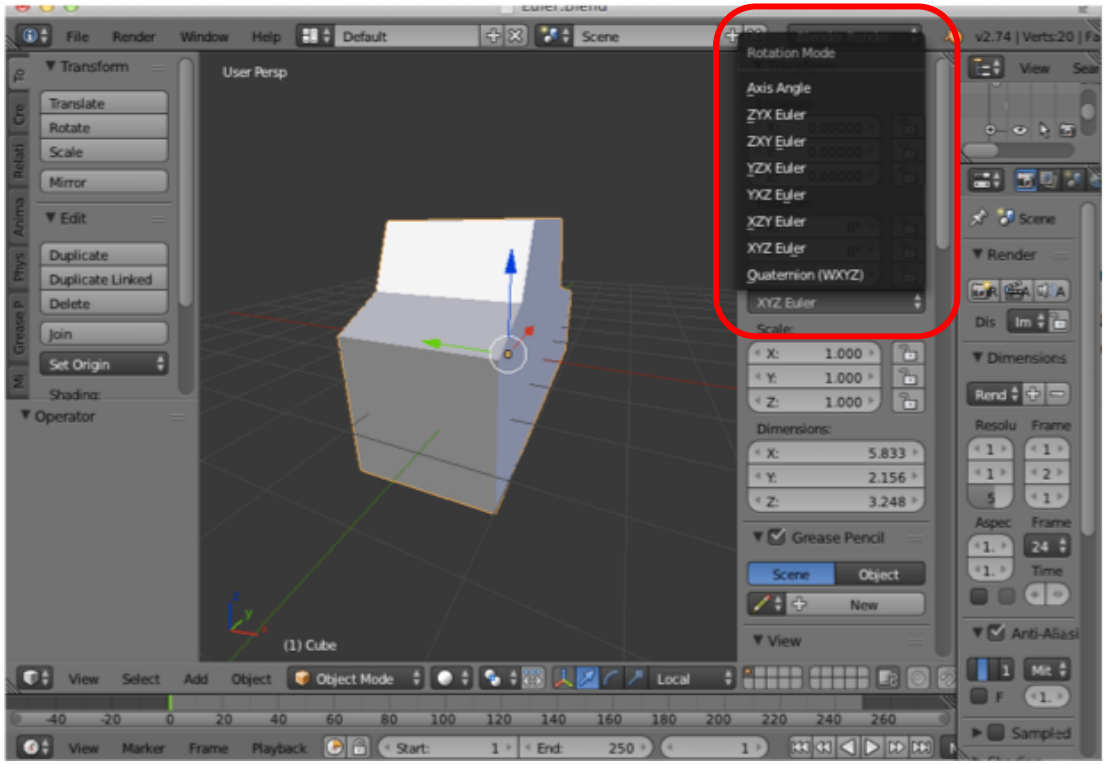
If the coordinates system is *y-up* (as we have considered until now), and the object faces the positive *z-axis*, and rotations should be performed in the *z*, *x* and *y* order.

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.

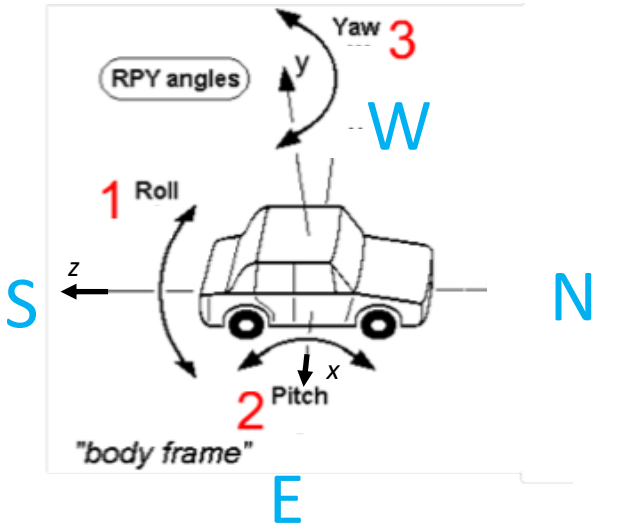


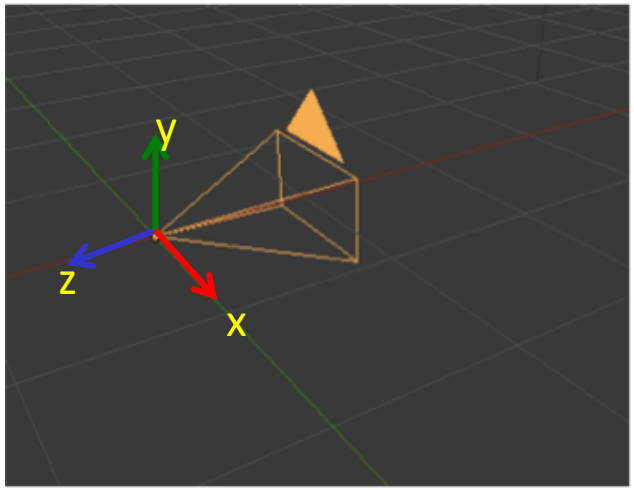
Euler angles

For example, the *"Look-in-direction"* camera model introduced earlier, uses also 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* ϕ and *Pitch* θ works in the opposite way with respect to ρ and β, 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.

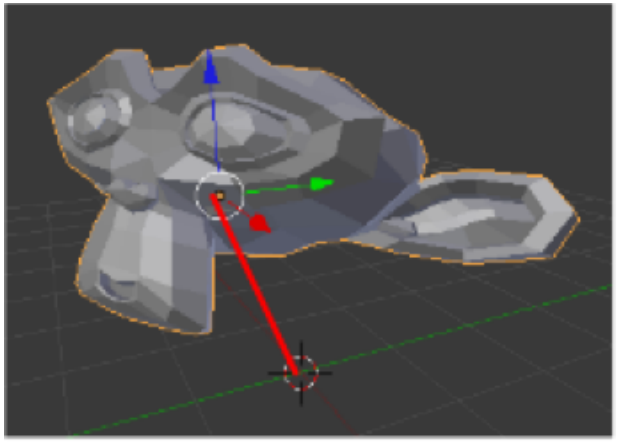




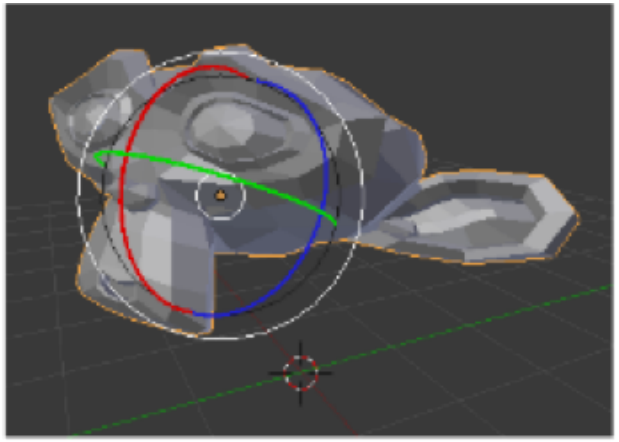
Creating a World Matrix: final result

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

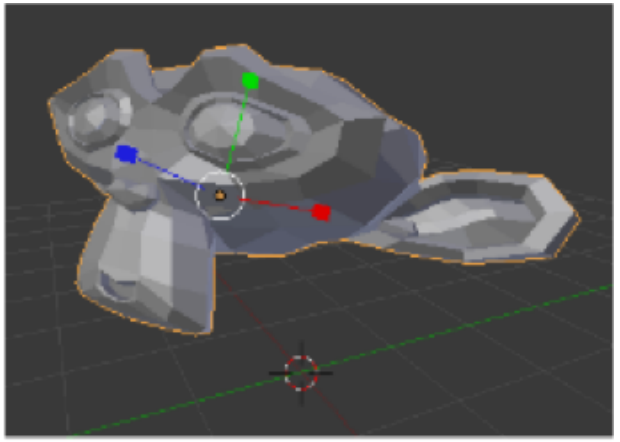
All the parameters have an intuitive "physical" meaning.



p_x, p_y, p_z



φ, ψ, θ

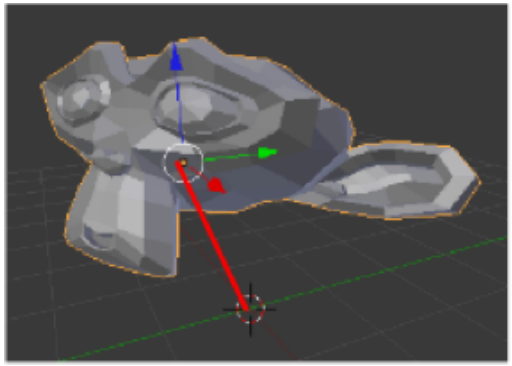


s_x, s_y, s_z

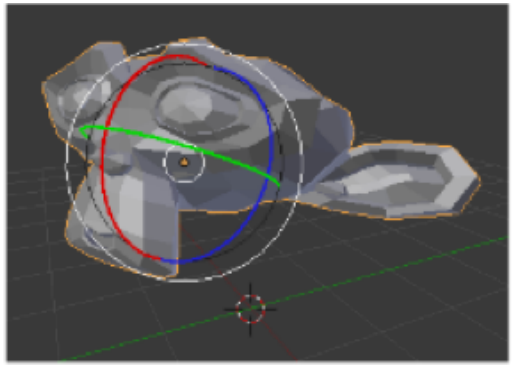
Creating a World Matrix: final result

The *World Matrix* M_w can then be computed by factorizing the five transformations in the correct order:

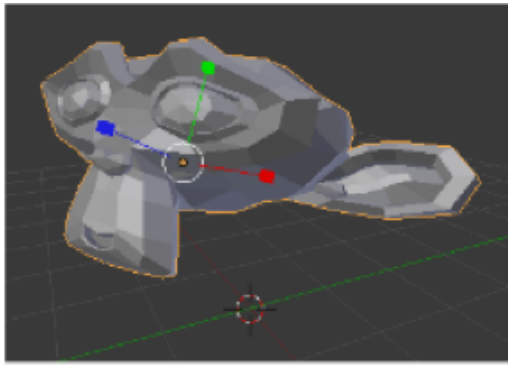
$$M_W = T(p_x, p_y, p_z) \cdot R_y(\psi) \cdot R_x(\theta) \cdot R_z(\varphi) \cdot S(s_x, s_y, s_z)$$



p_x, p_y, p_z



φ, ψ, θ



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:

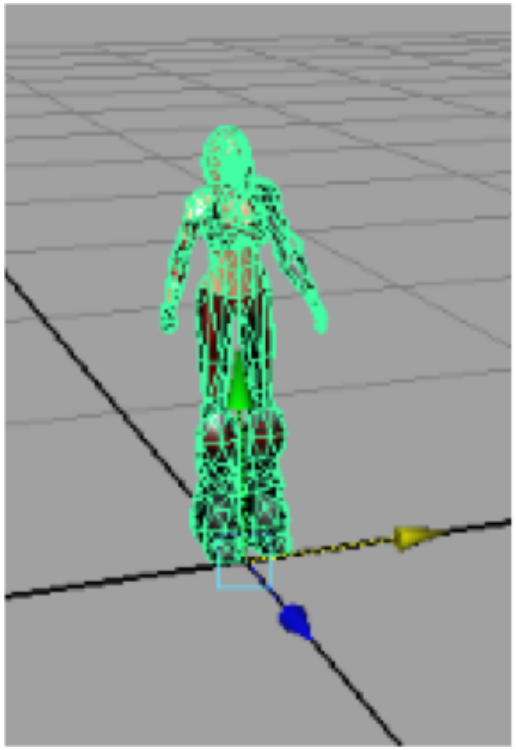
```
glm::mat4 Mw =
    glm::translate(glm::mat4(1.0), glm::vec3(px, py, pz)) *
    glm::rotate(glm::mat4(1.0),    yaw, glm::vec3(0,1,0)) *
    glm::rotate(glm::mat4(1.0),  pitch, glm::vec3(1,0,0)) *
    glm::rotate(glm::mat4(1.0),   roll, glm::vec3(0,0,1)) *
    glm::scale(glm::mat4(1.0), glm::vec3(sx, 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 positive x-axis. Rotation uses Euler angles in xzy order, character face east with a zero yaw.

Gimbal Lock

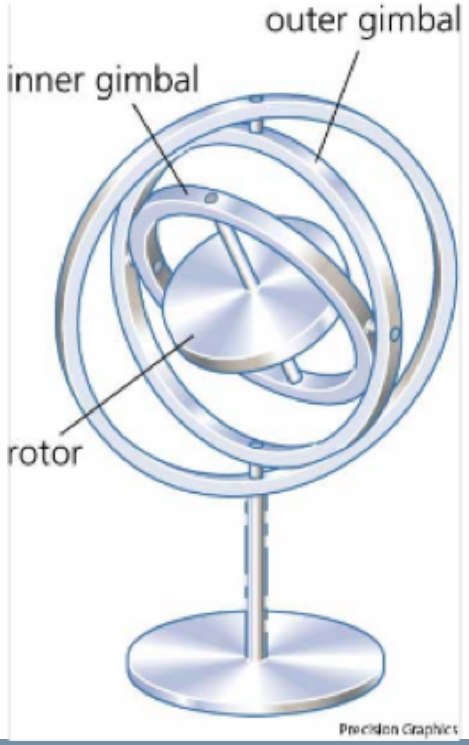
A rotation defined by the Euler Angles, is perfect for "planar" movements, like the one available in a driving simulation or in a FPS.

However they are not the proper solution for applications such as flight or space simulators since they can suffer a problem known as *Gimbal Lock*.

Gimbal Lock

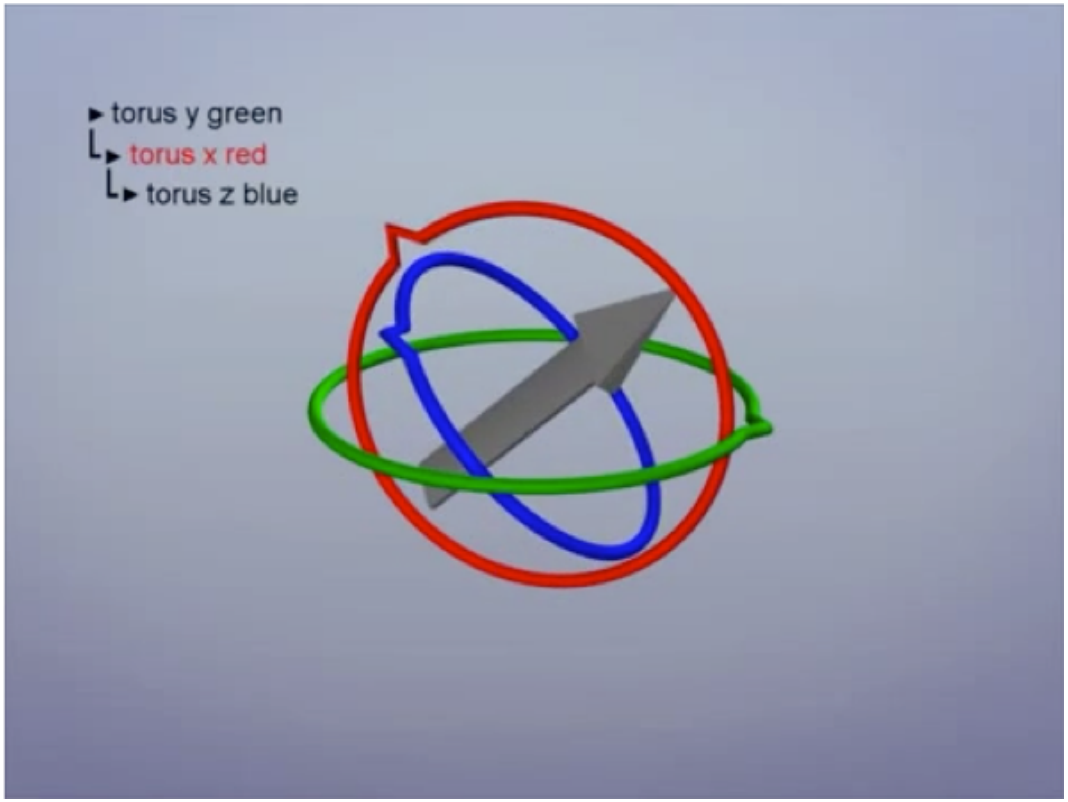
A *gimbal* is a ring that can spin around its diameter.

A physical system that allows freely orienting an object in the space has at least three gimbals connected to each other.



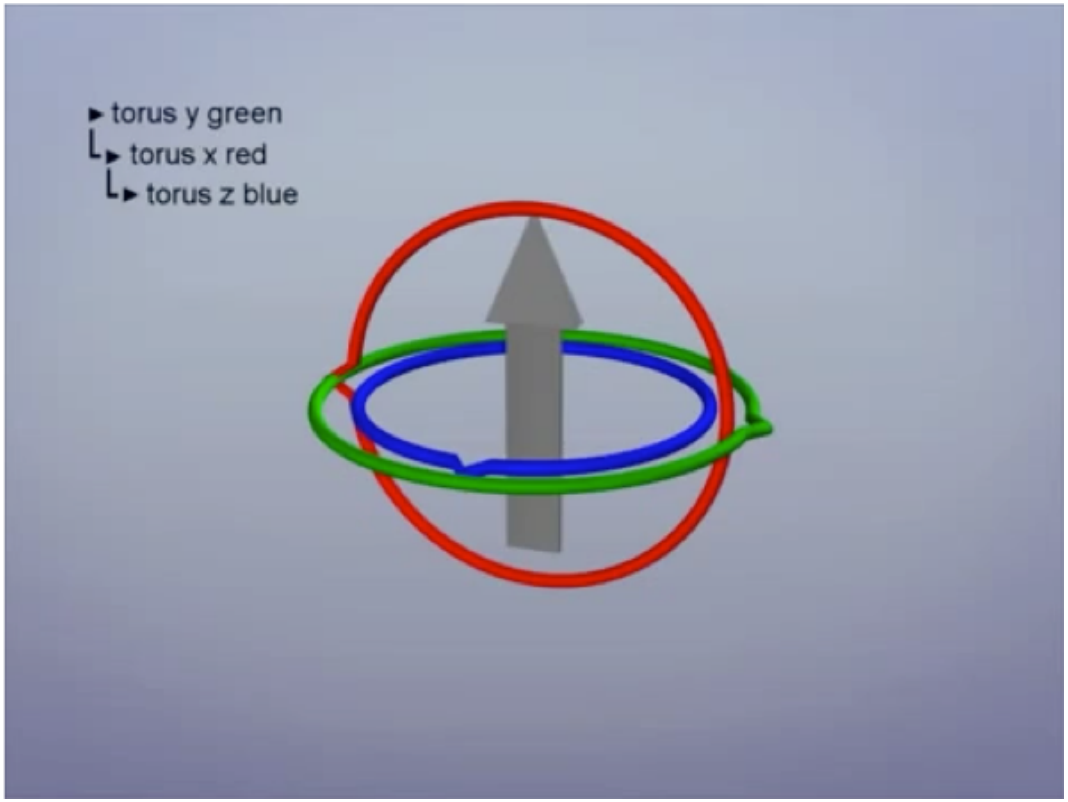
Gimbal Lock

During rotations, the *pitch* also moves the *roll* axis, and the *yaw* moves both the *pitch* and the roll *axes*.



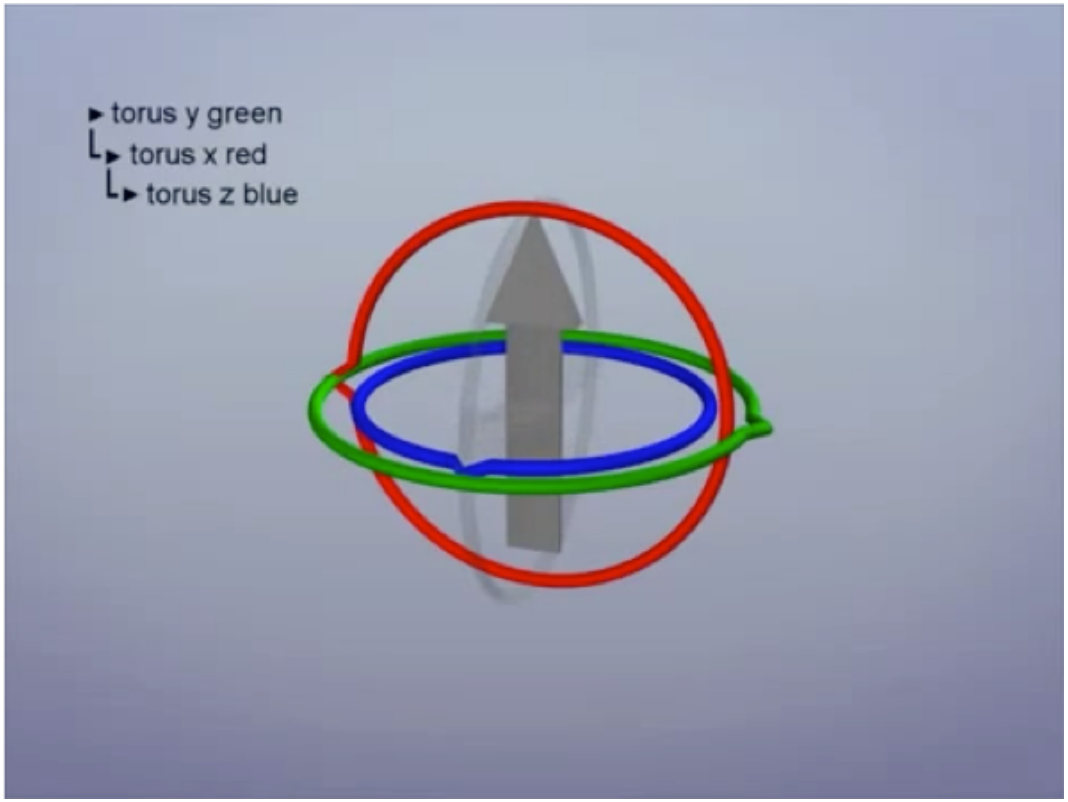
Gimbal Lock

If the pitch rotates exactly *90º* degrees, the roll and the yaw become superposed.



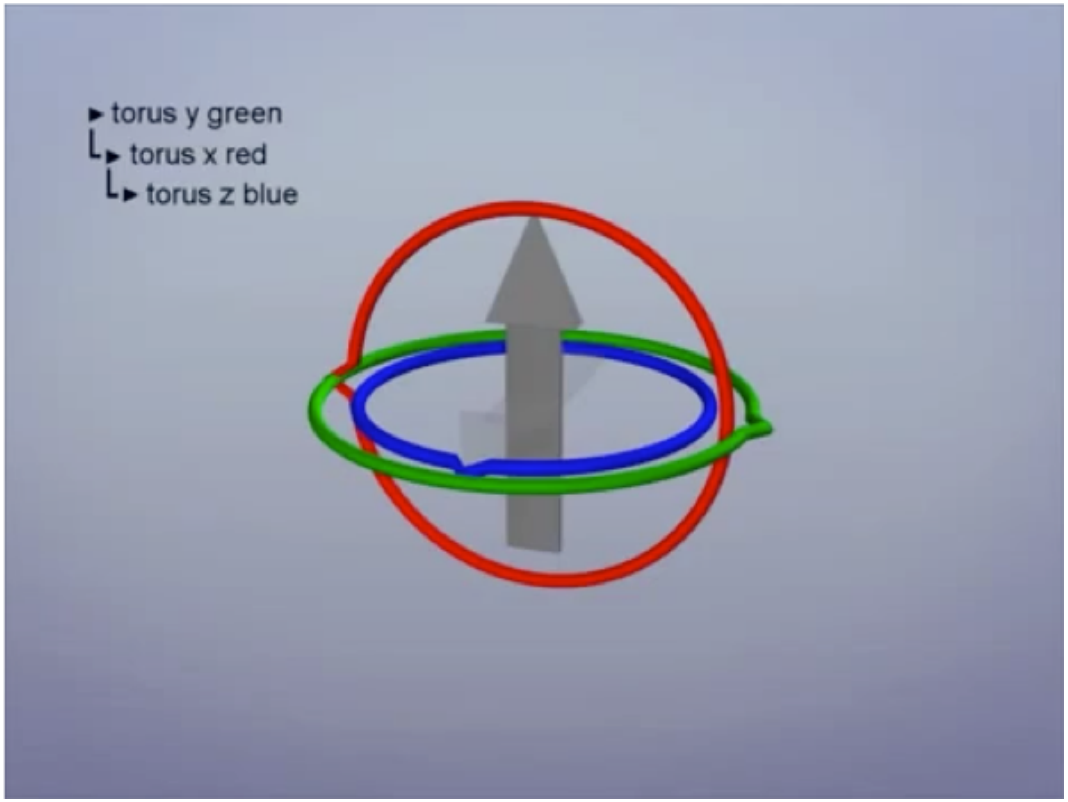
Gimbal Lock

A degree of freedom is thus lost.



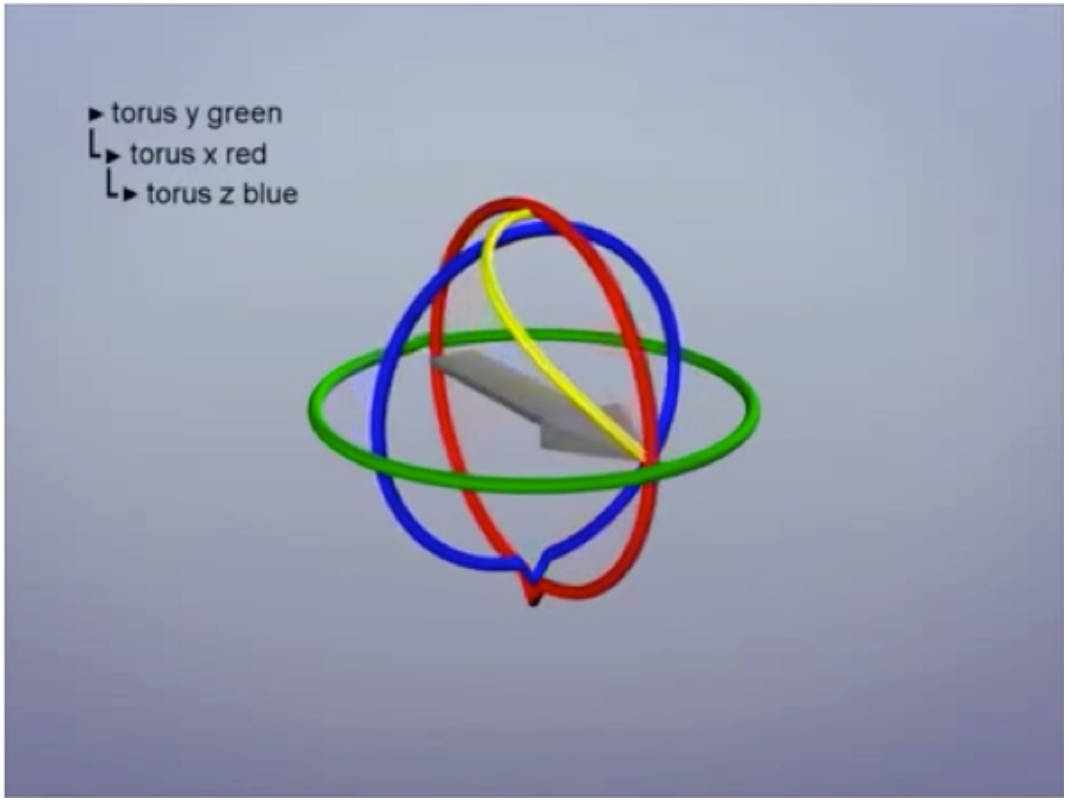
Gimbal Lock

When a *Gimbal Lock* occurs, some movements are no longer possible in a natural way.



Gimbal Lock

Such movements must be accomplished by complex combinations of the three basic rotations.



Gimbal Lock: Quatenrions

A common solution is to express the rotation of an object with a mathematical device called a *quaternion*.

Euler Angles are however used very commonly, since Gimbal Lock does not occur in most common VR applications, and quaternions increase the mathematical complexity of the procedure.

Quaternions

Quaternions are an extension of complex numbers that have three imaginary components:

Complex number: $a + ib$

Quaternion:

$$a + ib + jc + kd$$

The three imaginary components, that are called the *vector part*, are subject to the following relations:

$$i^2 = j^2 = k^2 = ijk = -1$$

Quaternions

From the previous specification, a complete algebra can be defined, where some of the operations are as follows:

$$(a_1 + ib_1 + jc_1 + kd_1) + (a_2 + ib_2 + jc_2 + kd_2) = (a_1 + a_2) + i(b_1 + b_2) + j(c_1 + c_2) + k(d_1 + d_2)$$

Sum

$$\alpha(a + ib + jc + kd) = \alpha a + i \cdot \alpha b + j \cdot \alpha c + k \cdot \alpha d$$

Product (with scalar)

$$\begin{aligned}(a_1 + ib_1 + jc_1 + kd_1)(a_2 + ib_2 + jc_2 + kd_2) = \quad & (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + \\ & i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) + \\ & j(a_1c_2 + c_1a_2 + d_1b_2 - b_1d_2) + \\ & k(a_1d_2 + d_1a_2 + b_1c_2 - c_1b_2)\end{aligned}$$

Product (two quaternions)

$$\|a + ib + jc + kd\| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

Norm (length)

$$\theta = \arccos \frac{a}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

$$(a + ib + jc + kd)^\alpha = \|a + ib + jc + kd\|^\alpha \left(\cos(\alpha\theta) + \frac{ib + jc + kd}{\sqrt{b^2 + c^2 + d^2}} \sin(\alpha\theta) \right)$$

Power

Quaternion rotation

A unitary quaternion *q* has it norm ||*q*||= *1.*

Unitary quaternions can be used to encode 3D rotations.

Let us consider a rotation of an angle θ around an axis oriented along a unitary vector ***v*** = *(x,y,z)*. This rotation can be represented by the following quaternion:

$$q = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(ix + jy + kz)$$

Since ***v*** is unitary, also *q* is unitary.

Quaternion rotation

Example:

Consider a rotation of 90^o about an arbitrary axis that lies on the *xy-plane* and it is angled 30^o with respect to the x-axis. Write the quaternion that encodes such rotation.

The vector that defines the direction of the axis has the following form:

$$v = (\cos 30^\circ, \sin 30^\circ, 0) = (0.866, 0.5, 0)$$

Rotation is thus encoded by the following *Quaternion:*

$$0.707 + 0.612i + 0.354j$$

Quaternion rotation

Example:

An arbitrary axis v lies on the diagonal of a box, from point A (3, 3, 0) to point B (0, 3, 0). A quaternion that encodes a rotation of 30° around it can be computed in the following way:

1. We start computing vector **v** as the normalized difference of the two points:

$$v = \frac{(0,3,0) - (3,3,0)}{|(0,3,0) - (3,3,0)|} = \frac{(-3,0,0)}{|(3,0,0)|} = (-1,0,0)$$

2. We then apply the formula previously given: $q = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(ix + jy + kz)$

$$q = \cos 15^o + \sin 15^o\,(-1,0,0) = 0.966 - 0.259\boldsymbol{i}$$

Quaternion rotation

Quaternions can be directly converted to rotation transform matrices:

$$q = a + ib + jc + kd$$

$$R(q) = \begin{vmatrix} 1-2c^2-2d^2 & 2bc+2ad & 2bd-2ac & 0 \\ 2bc-2ad & 1-2b^2-2d^2 & 2cd+2ab & 0 \\ 2bd+2ac & 2cd-2ab & 1-2b^2-2c^2 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

PS: for the matrix on the right notation, the transform matrix would be transposed.

Quaternion rotation

If q_1 and q_2 are two unitary quaternions that encode two different rotations, their products encodes the composed transform:

$$M_1 \Leftrightarrow q_1, \quad M_2 \Leftrightarrow q_2 \quad \Rightarrow \quad M_1 \cdot M_2 \Leftrightarrow q_1 \cdot q_2$$

With these definitions, we can transform a set of Euler angles to a quaternion:

$$R = R_y(\psi) \cdot R_x(\theta) \cdot R_z(\varphi), \quad q = (\cos\frac{\psi}{2} + j\sin\frac{\psi}{2})(\cos\frac{\theta}{2} + i\sin\frac{\theta}{2})(\cos\frac{\varphi}{2} + k\sin\frac{\varphi}{2})$$

$$q = \left(\cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\varphi}{2} - \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\varphi}{2}\right) + i\left(\cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\varphi}{2} - \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\varphi}{2}\right) +$$
$$j\left(\sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\varphi}{2} - \cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\varphi}{2}\right) + k\left(\cos\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\varphi}{2} - \sin\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\varphi}{2}\right)$$

Quaternion rotation

Note that also the quaternions product is not commutative.

Since the rotation order matters, the order in which quaternions are multiplied should be identical to the one of the corresponding matrices.

$$M_1 \Leftrightarrow q_1, \quad M_2 \Leftrightarrow q_2 \quad \Rightarrow \quad \underrightarrow{M_1 \cdot M_2} \Leftrightarrow \underrightarrow{q_1 \cdot q_2}$$

Quaternion rotation

It is possible to extract the Euler angles from a rotation matrix:

$$
\begin{aligned}
R &= R_z(\phi)R_y(\theta)R_x(\psi) \\
&= \begin{bmatrix} \cos\theta\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \cos\theta\sin\phi & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi \\ -\sin\theta & \sin\psi\cos\theta & \cos\psi\cos\theta \end{bmatrix}
\end{aligned}
$$

$$
R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix}
$$

```
if (R31 ≠ ±1)
    θ1 = −asin(R31)
    θ2 = π − θ1
    ψ1 = atan2(R32/cos θ1, R33/cos θ1)
    ψ2 = atan2(R32/cos θ2, R33/cos θ2)
    φ1 = atan2(R21/cos θ1, R11/cos θ1)
    φ2 = atan2(R21/cos θ2, R11/cos θ2)
else
    φ = anything; can set to 0
    if (R31 = −1)
        θ = π/2
        ψ = φ + atan2(R12, R13)
    else
        θ = −π/2
        ψ = −φ + atan2(−R12, −R13)
    end if
end if
```

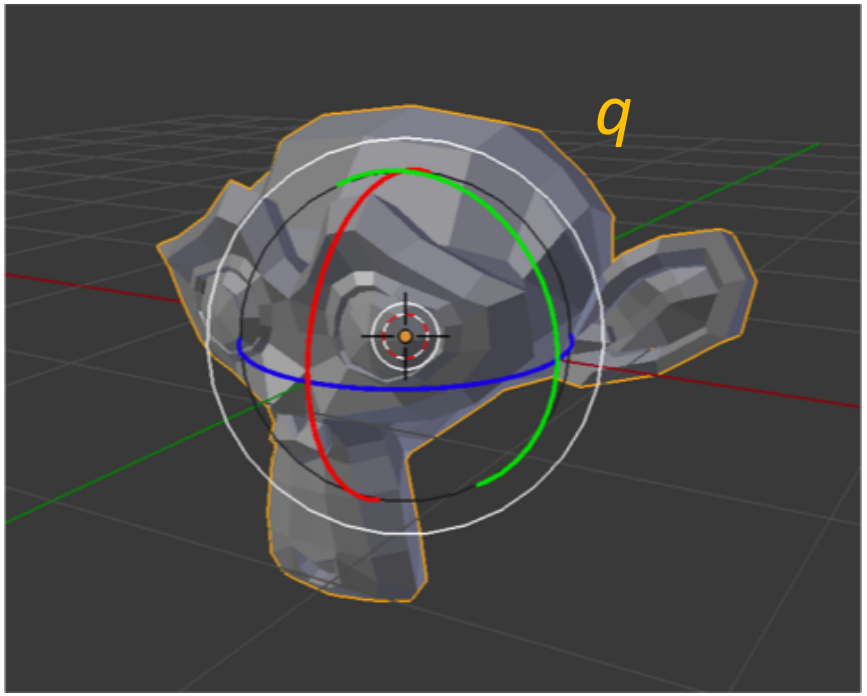
To extract Euler angles from a quaternion, first its rotation matrix is computed, and then angles are extracted from the matrix.

Quaternion usage

In applications characterized by complex rotations, the orientation of an object is stored in memory with a quaternion *q*.

When the world-matrix has to be computed, this quaternion is converted into the corresponding rotation matrix.

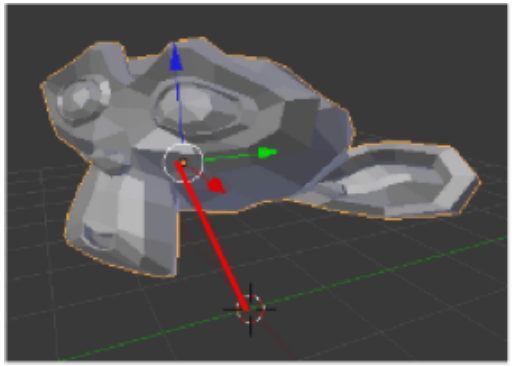
Such matrix is then multiplied with both the translation and the scaling components.



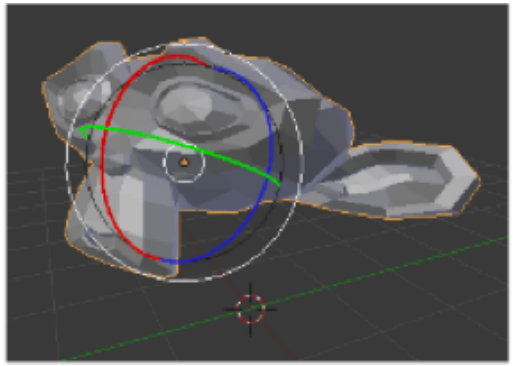
Quaternion usage

In other words, the *World Matrix* M_w can be computed as follows:

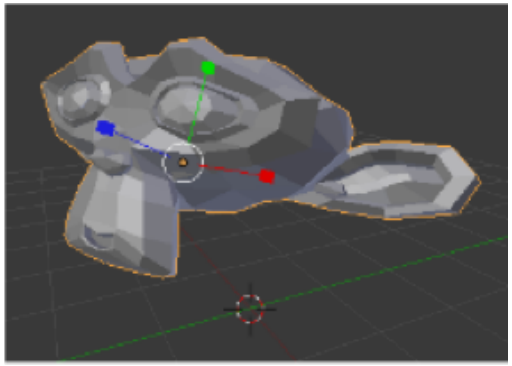
$$M_W = T(p_x, p_x, p_x) \cdot R(q) \cdot S(s_x, s_x, s_x)$$



p_x, p_y, p_z



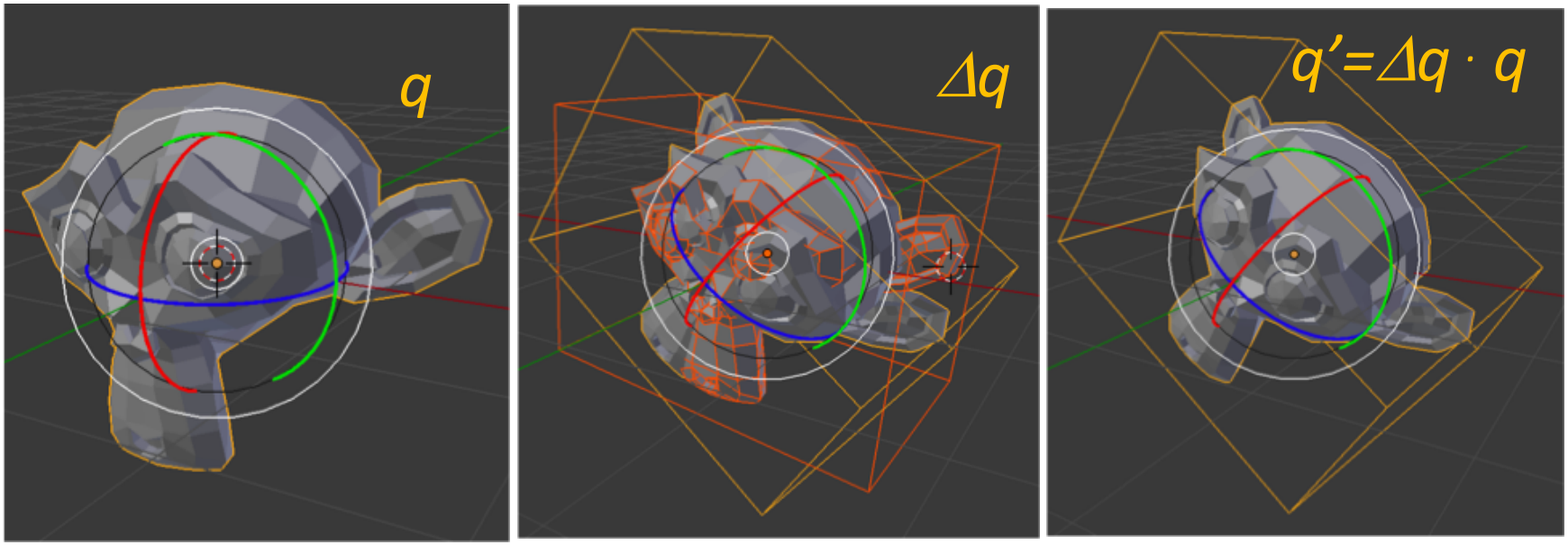
q



s_x, s_y, s_z

Quaternion usage

The application always performs the rotations using quaternion operations: all relative changes in the direction of an object are encoded with a quaternion Δq that expresses the direction and the entity of the rotation.



Quaternion usage

For example, to rotate an object whose current orientation is quaternion q, 6 degrees around the *y axis*, the following steps are used:

1) Rotation is encoded into a quaternion $\Delta q =$ that considers the direction (vector *(0,1,0)* since it is the *y-axis*) and the amount (6^o):

$$\Delta q = \cos 3^o + \sin 3^o \, (0,1,0) = 0.9986 + 0.0523\boldsymbol{j}$$

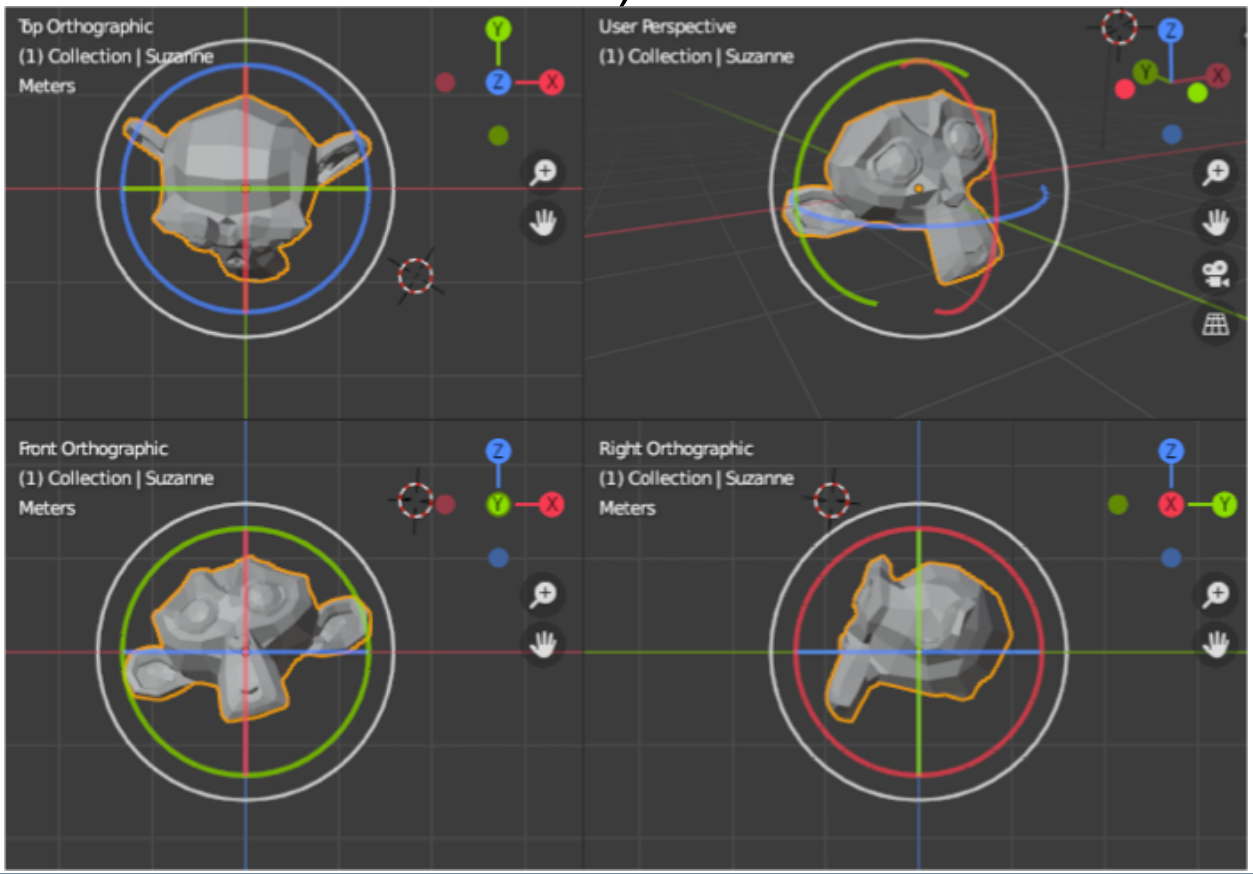
2) The quaternion *q* representing the current direction of the object is multiplied by Δq:

$$q = \Delta q \cdot q \quad \text{or} \quad q = q \cdot \Delta q$$

Quaternion usage

When the rotation is first, it is performed in *world* space (i.e. using the axis of the current world coordinates)

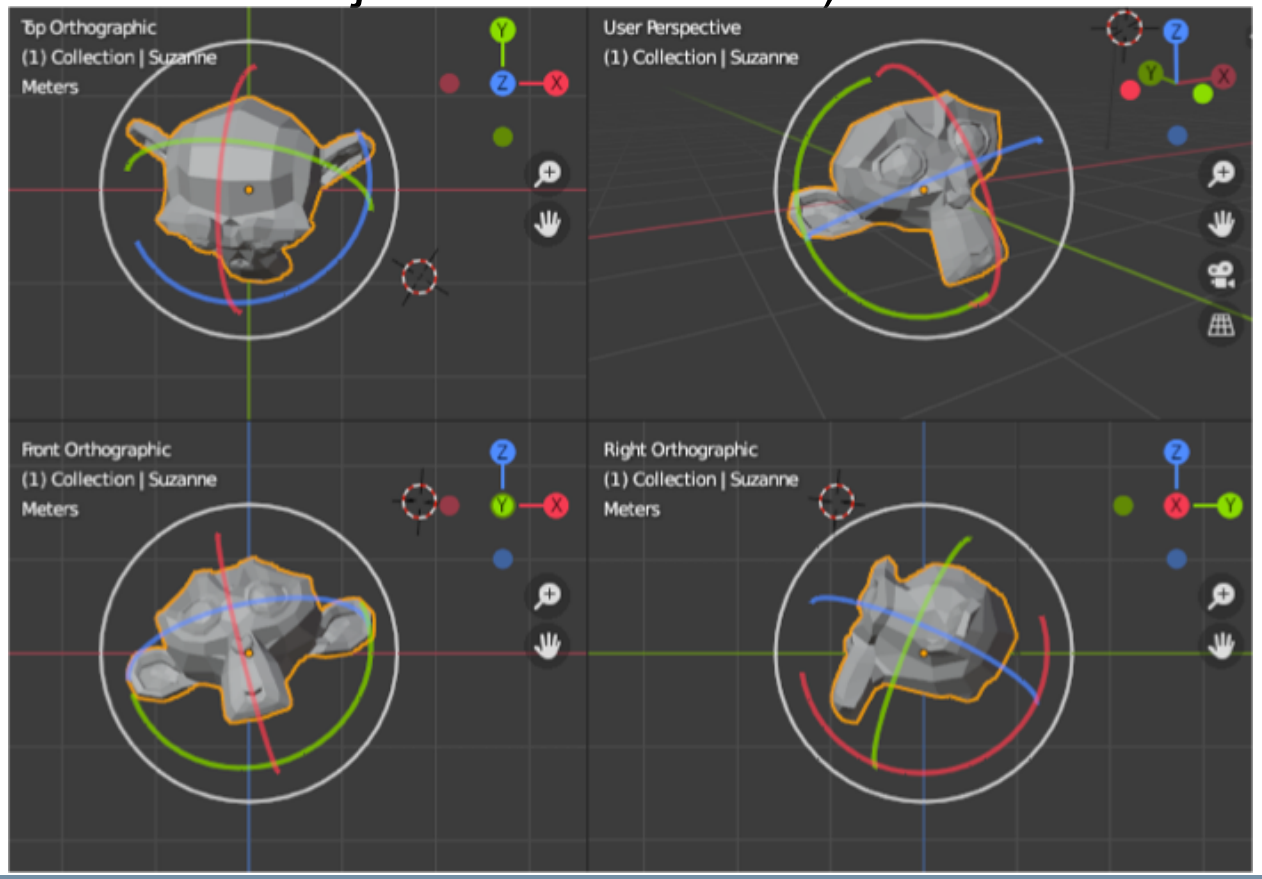
$$q = \Delta q \cdot q$$



Quaternion usage

When the rotation is last, it is performed in *local* space (i.e. using the axis of the in which the object was modelled).

$$q = q \cdot \Delta q$$



Quaternions in GLM

Quaternion support in GLM is split between base and extended functions:

We will only focus on base functions

Main Page Related Pages Modules Classes Files

GLM_GTC_quaternion: Quaternion types and functions

GTC Extensions (Stable)

Classes

struct tquat< T >
Template for quaternion. More...

Typedefs

typedef detail::tquat< double > dquat
typedef detail::tquat< float > fquat
typedef detail::tquat < highp_float > highp_quat
typedef detail::tquat < detail::half > hquat
typedef detail::tquat< lowp_float > lowp_quat
typedef detail::tquat < mediump_float > mediump_quat
typedef detail::tquat< float > quat

Functions

template<typename T > detail::tquat< T > conjugate (detail::tquat< T > const &q)
template<typename T > T dot (detail::tquat< T > const &q1, detail::tquat< T > const &q
template<typename T > detail::tquat< T > inverse (detail::tquat< T > const &q)
template<typename T > T length (detail::tquat< T > const &q)
template<typename T > detail::tmat3x3< T > mat3_cast (detail::tquat< T > const &x)
template<typename T > detail::tmat4x4< T > mat4_cast (detail::tquat< T > const &x)
template<typename T > detail::tquat< T > mix (detail::tquat< T > const &x, detail::tquat< T > const &y,

Main Page Related Pages Modules Classes Files

GLM_GTX_quaternion: Extented quaternion types and functions

GTX Extensions (Experimental)

Functions

template<typename valType > valType angle (detail::tquat< valType > const &x)
template<typename valType > detail::tquat< valType > angleAxis (valType const &angle, valType const &x, valType const &y, va
template<typename valType > detail::tquat< valType > angleAxis (valType const &angle, detail::tvec3< valType > const &axis)
template<typename valType > detail::tvec3< valType > axis (detail::tquat< valType > const &x)
template<typename valType > detail::tvec3< valType > cross (detail::tquat< valType > const &q, detail::tvec3< valType > cons
template<typename valType > detail::tvec3< valType > cross (detail::tvec3< valType > const &v, detail::tquat< valType > cons
template<typename valType > detail::tvec3< valType > eulerAngles (detail::tquat< valType > const &x)
template<typename valType > detail::tquat< valType > exp (detail::tquat< valType > const &q, valType const &exponent)
template<typename valType > valType extractRealComponent (detail::tquat< valType > const &q)
template<typename T > detail::tquat< T > fastMix (detail::tquat< T > const &x, detail::tquat< T > const &y, T co
template<typename valType > detail::tquat< valType > intermediate (detail::tquat< valType > const &prev, detail::tquat< valT detail::tquat< valType > const &next)
template<typename valType > detail::tquat< valType > log (detail::tquat< valType > const &q)
template<typename valType > valType pitch (detail::tquat< valType > const &x)
template<typename valType > detail::tquat< valType > pow (detail::tquat< valType > const &x, valType const &y)
template<typename valType > valType roll (detail::tquat< valType > const &x)
template<typename valType > detail::tvec3< valType > rotate (detail::tquat< valType > const &q, detail::tvec3< valType > con

Quaternions in GLM

To use quaternions functions, the specific part of the GLM library must be included:

```
#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/gtc/quaternion.hpp>
```

Quaternions in GLM

Quaternions can be created in three ways:

1 - from the Euler angles, specified in the pitch, yaw, roll order:

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

Please note that the pitch, yaw, roll order is very rarely used, therefore this constructor can be used only to build the base quaternions corresponding to the three main rotations.

To have a quaternion representing a proper rotation matrix with given Euler angles, the building blocks then should be manually composed:

```
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(yaw), 0)) *
               glm::quat(glm::vec3(glm::radians(pitch), 0, 0)) *
               glm::quat(glm::vec3(0, 0, glm::radians(roll)));

glm::mat4 MQ = glm::mat4(qe);
```

Quaternions in GLM

Quaternions can be:

2 – specifying the *scalar part* first, then the ***i***, ***j***, ***k*** vector components:

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

Quaternions can be:

3 - from the angle rotation, and the axis direction:

(this can be achieved rotating a unitary quaternion using the *`rotate()`* function)

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

Scalar component can be accessed with the `.w` field. The ***i***, ***j***, ***k*** vector components respectively with the `.x` , `.y` and `.z` fields.

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

Product and other algebraic operations among quaternions can be performed with the usual symbols. For example, for product:

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

The equivalent 4x4 rotation matrix can be computed passing it as a constructor parameter to the `glm::mat4` matrix type.

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

Since rotations are encoded with unitary quaternions, the `glm::normalize` function can ensure this property starting from an arbitrary element.

```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// create quaterion from scalar, i, j and k components
glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0, sin(glm::radians(22.5f)), 0);

// create quaterion from axis and angle
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f), glm::vec3(0,1,0));

// access quaternion elements
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;

// create rotation matrix from quaternion
glm::mat4 R = glm::mat4(qe);

// multiply quaternios
glm::quat qb = glm::quat(glm::vec3(glm::radians(30.0f), 0, 0));
glm::quat qp = qb * qa;

// normalize quaternions
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

Colors:
Red -> parameters
Green -> function names or operators.

Quaternions in GLM

The extended functions include many other interesting features such as Euler angle extraction, interpolation and faster construction and conversion procedures.

Will not consider them more in depth right now.