POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DEIB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

- Quaternions and Projections Wrap Up

Gimbal Lock

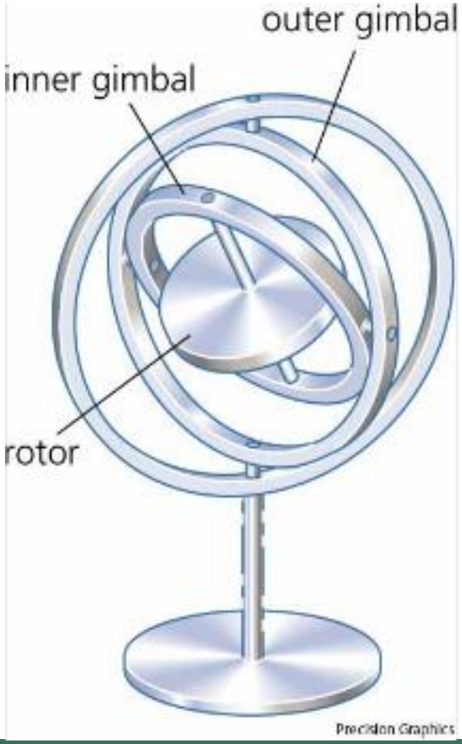
A rotation defined by the Euler Angles, is perfect for "planar" applications, for example like a driving simulation or a FPS.

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

Gimbal Lock

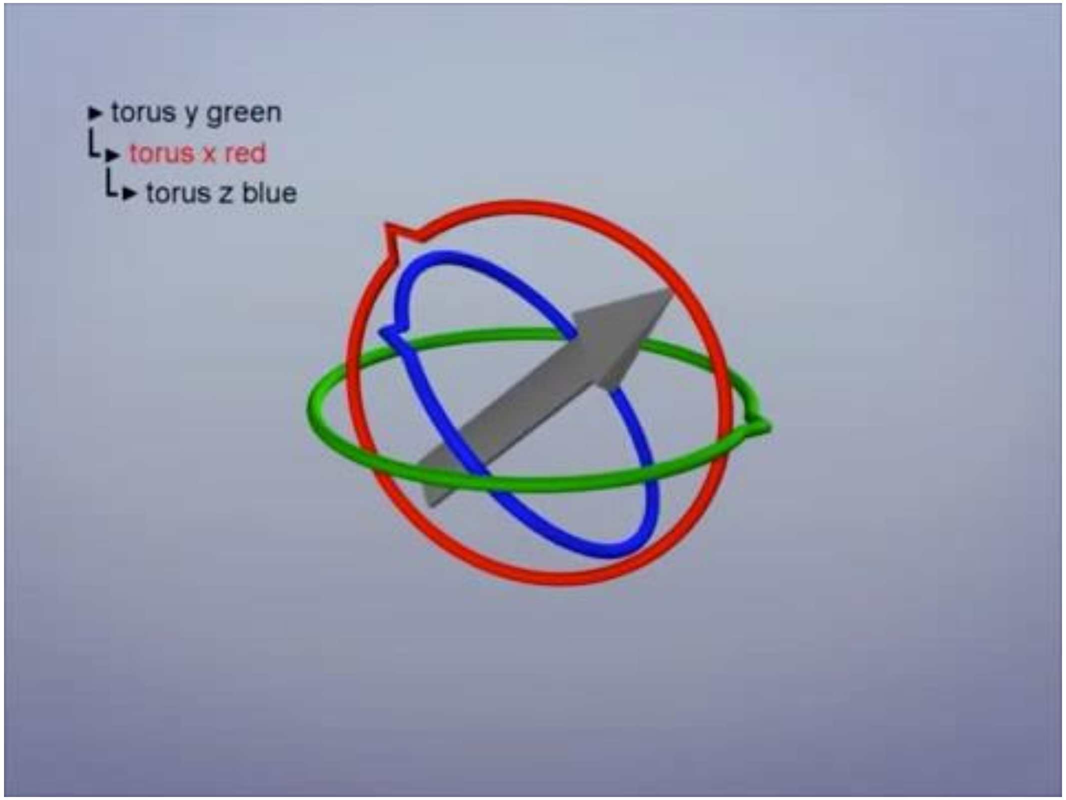
A *gimbal* is a ring that can spin around its diameter.

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



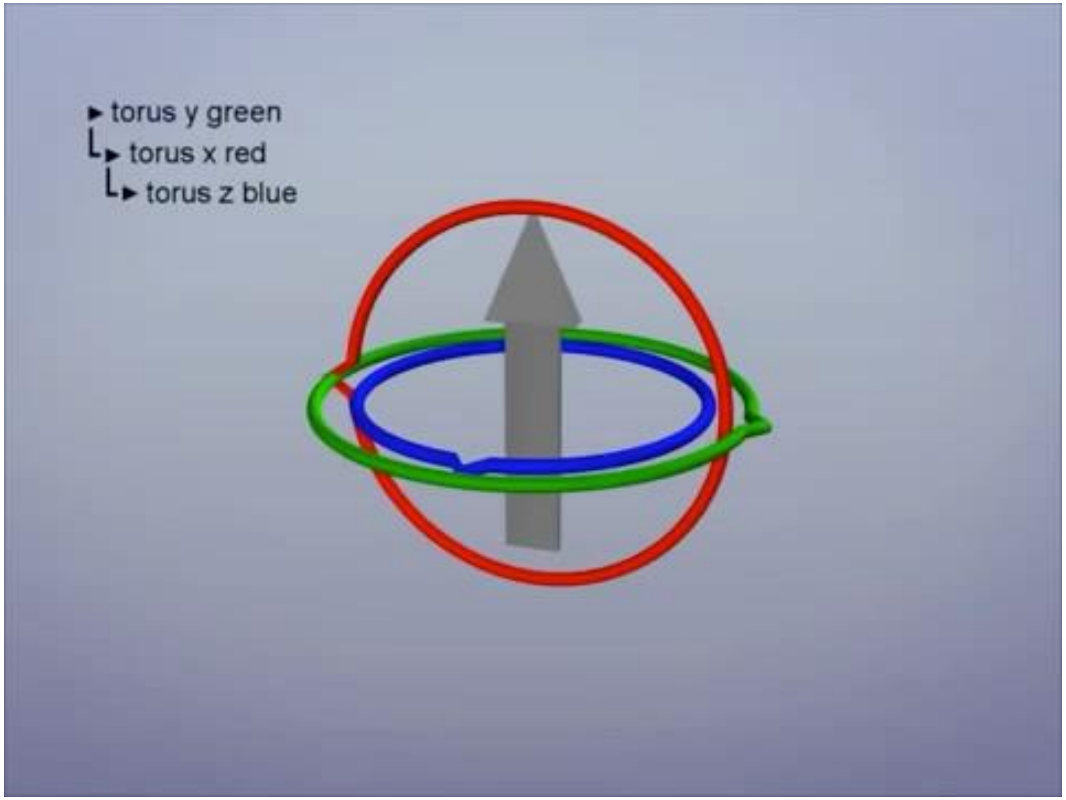
Gimbal Lock

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



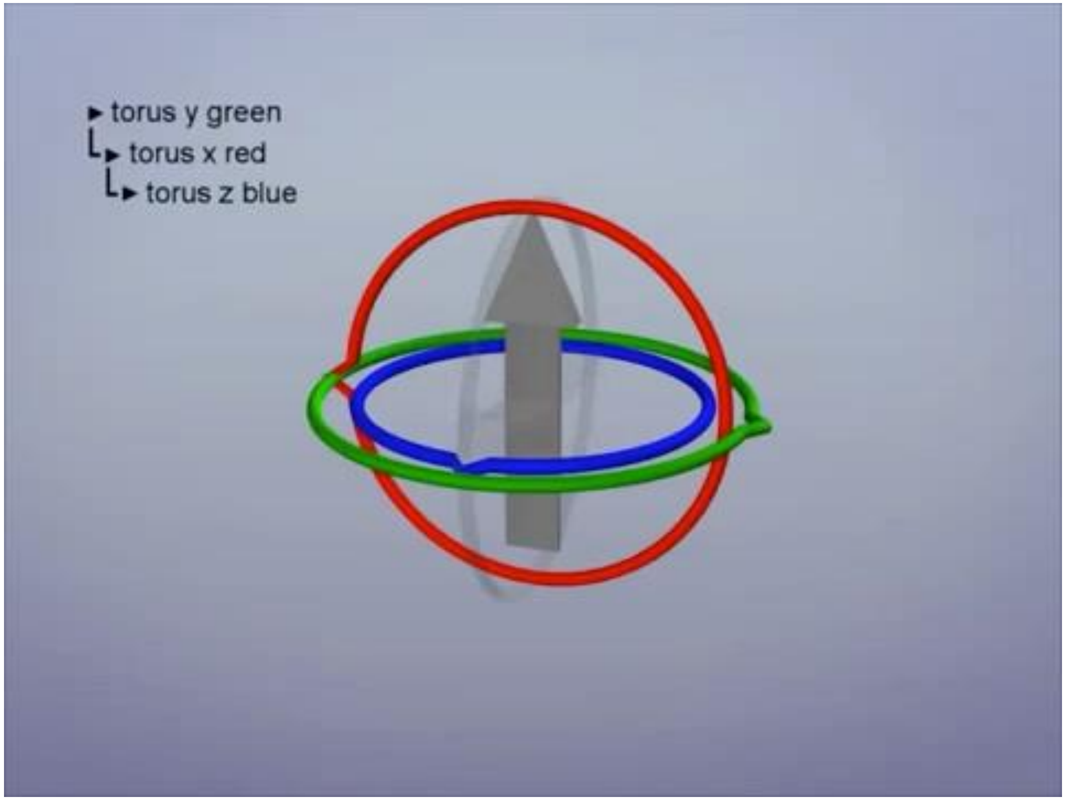
Gimbal Lock

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



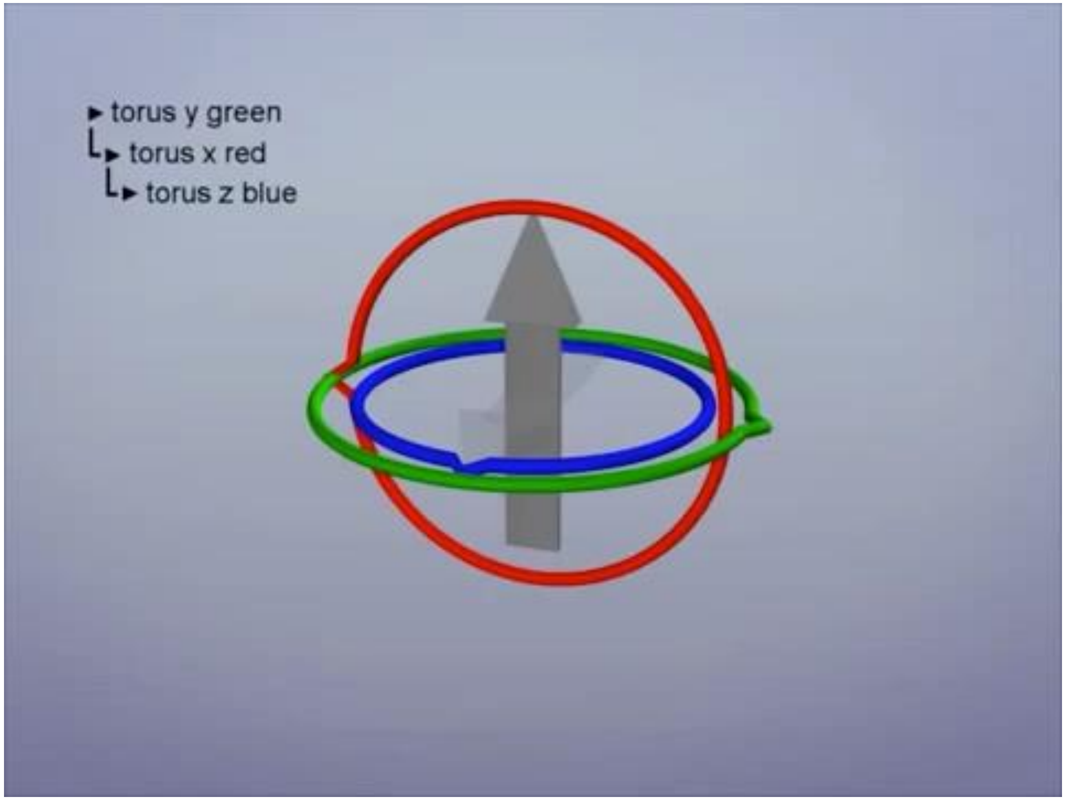
Gimbal Lock

A degree of freedom is thus lost.



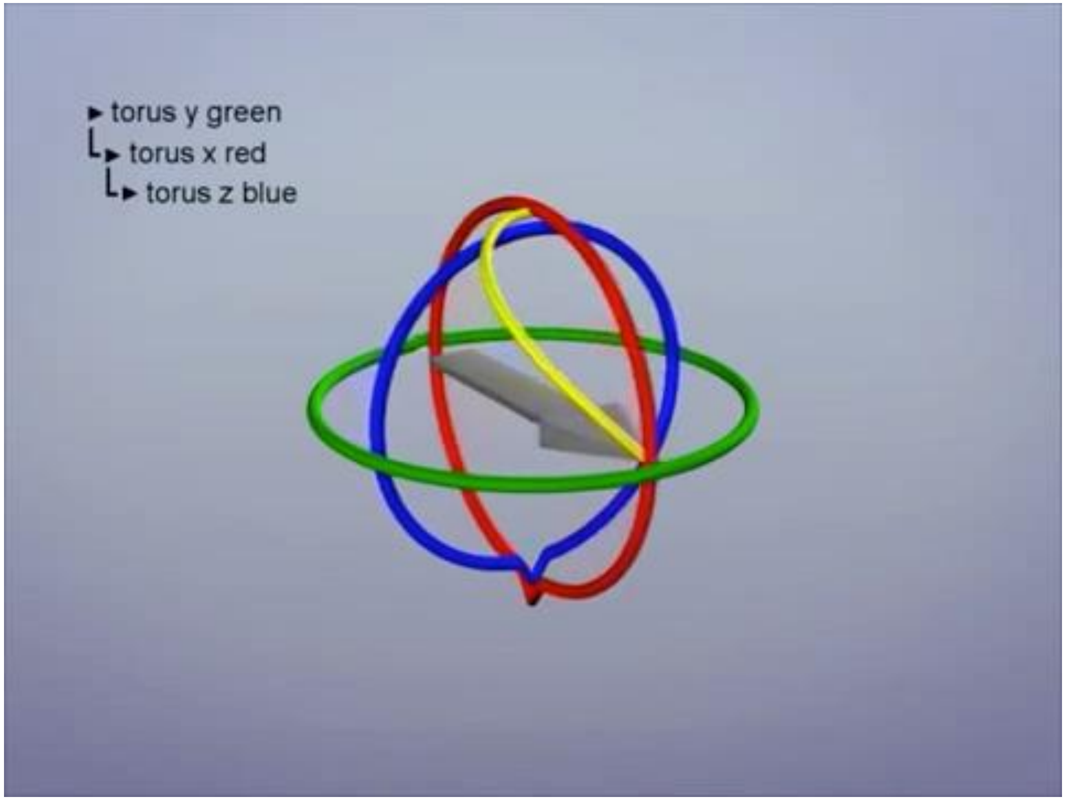
Gimbal Lock

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

By playing with these basic rules, all the combinations of the products between the imaginary parts can be computed:

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

$$i \cdot j = -i \cdot j \cdot (-1) = -i \cdot j \cdot k^2 = -(i \cdot j \cdot k) \cdot k = -(-1) \cdot k = k$$

$$j \cdot k = -(-1) \cdot j \cdot k = -i^2 \cdot j \cdot l = -i \cdot (i \cdot j \cdot k) = -i \cdot (-1) = i$$

$$\begin{aligned} k \cdot i &= -k \cdot i \cdot (-1) = -k \cdot i \cdot j^2 = k \cdot i \cdot j \cdot (-1) \cdot j = k \cdot i \cdot j \cdot (k^2) \cdot j \\ &= k \cdot (i \cdot j \cdot k) \cdot k \cdot j = -k \cdot k \cdot j = -(-1) \cdot j = j \end{aligned}$$

$$j \cdot i = j \cdot (j \cdot k) = (j \cdot j) \cdot k = -k$$

...

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)

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

Phase

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

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:

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

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

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

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:

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

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

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 $(R_{31} \neq \pm 1)$
- $\theta_1 = -\mathtt{asin}(R_{31})$
- $\theta_2 = \pi - \theta_1$
- $\psi_1 = \mathtt{atan2}\left(\frac{R_{32}}{\cos\theta_1}, \frac{R_{33}}{\cos\theta_1}\right)$
- $\psi_2 = \mathtt{atan2}\left(\frac{R_{32}}{\cos\theta_2}, \frac{R_{33}}{\cos\theta_2}\right)$
- $\phi_1 = \mathtt{atan2}\left(\frac{R_{21}}{\cos\theta_1}, \frac{R_{11}}{\cos\theta_1}\right)$
- $\phi_2 = \mathtt{atan2}\left(\frac{R_{21}}{\cos\theta_2}, \frac{R_{11}}{\cos\theta_2}\right)$

else
- $\phi =$ anything; can set to 0
- if $(R_{31} = -1)$
  - $\theta = \pi/2$
  - $\psi = \phi + \mathtt{atan2}(R_{12}, R_{13})$
- else
  - $\theta = -\pi/2$
  - $\psi = -\phi + \mathtt{atan2}(-R_{12}, -R_{13})$
- end if

end if

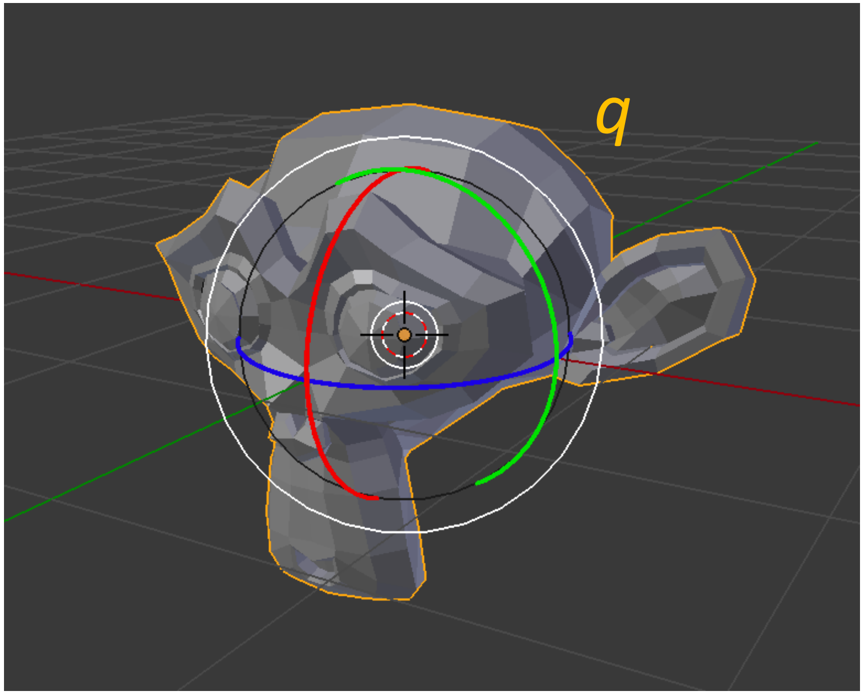
To extract Euler angles from a quaternion, first its rotation matrix is computed, and then angles are extracted from the matrix.

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.

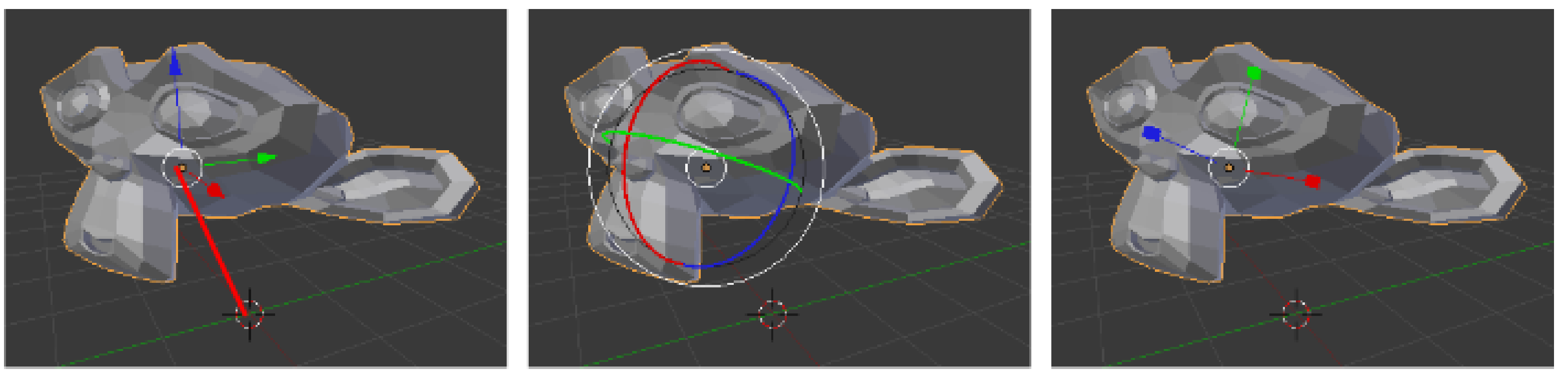
Such matrix is then multiplied with both the translation and the scaling components.



Quaternion usage

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

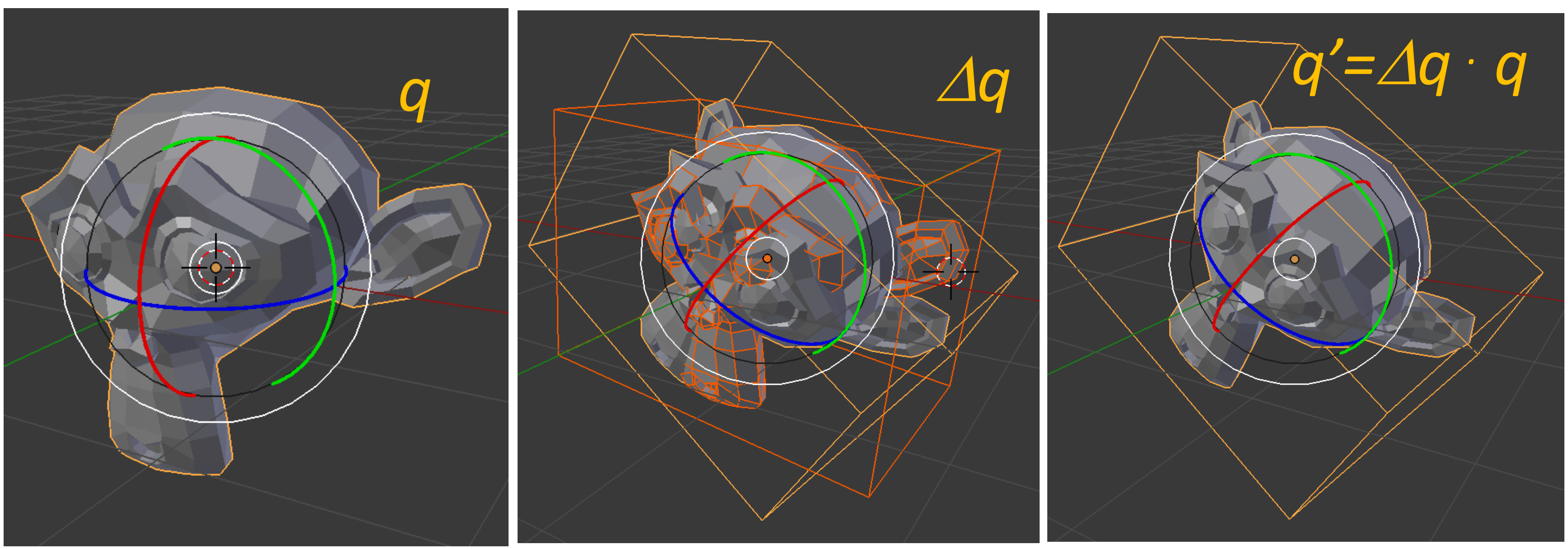
$$M_W = T(p_x, p_x, p_x) \cdot R(q) \cdot S(s_x, s_x, s_x)$$



p_x, p_y, p_z $\quad$ q $\quad$ 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 aomunt 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°*):

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

2) The quaternion *q* representing the current direction of the object is multiplied by $\Delta\, q$:

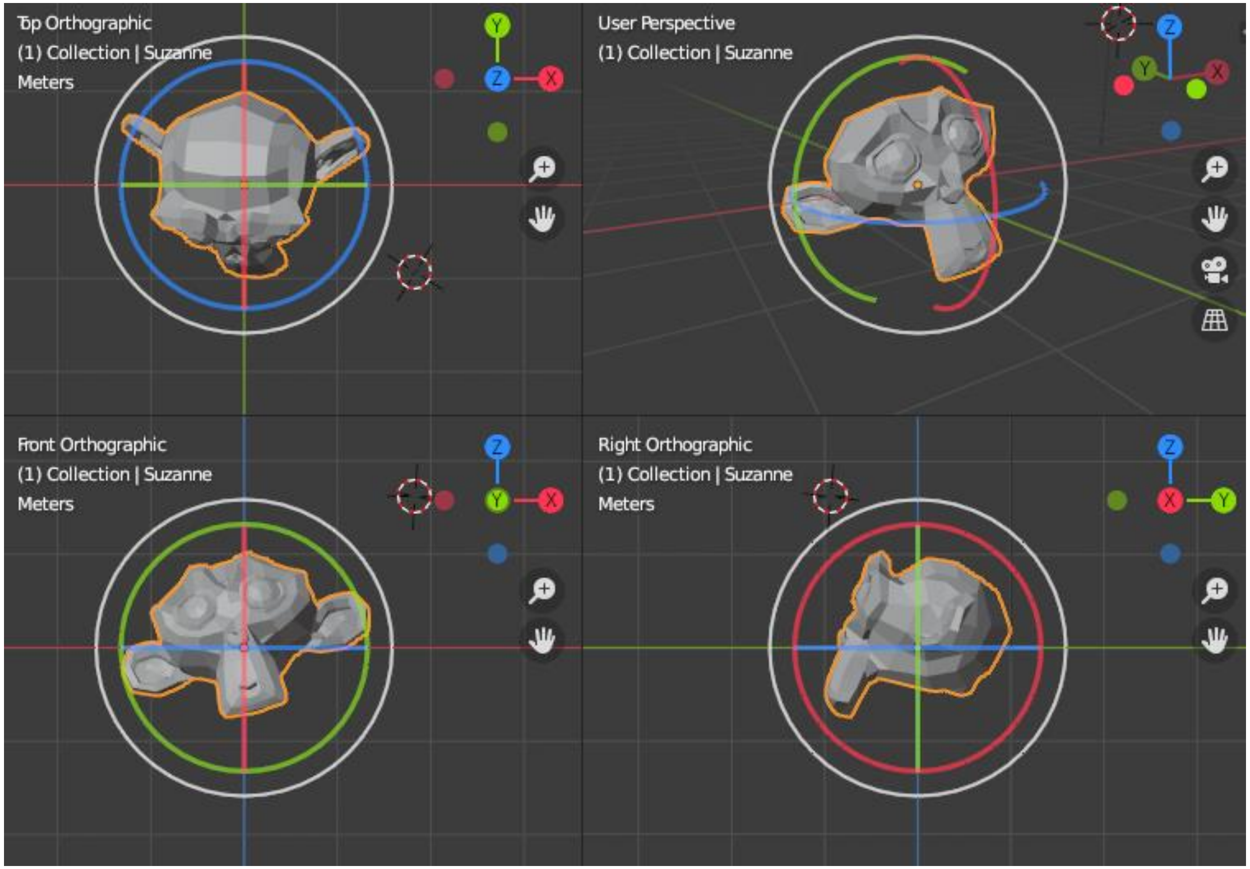
$$q = \Delta q \cdot q \quad \text{or} \quad q = q \cdot \Delta q$$

Quaternion usage

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

World space rotation

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

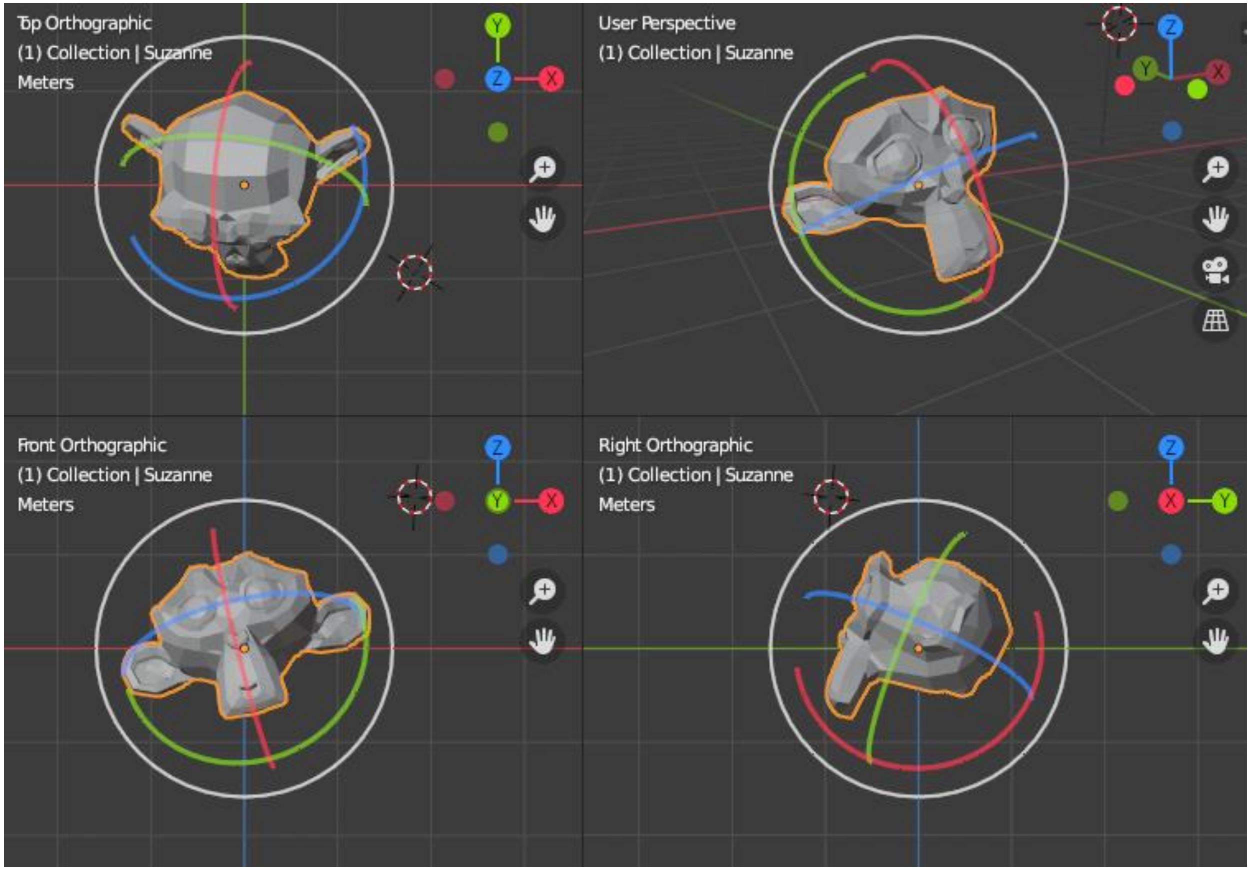


Quaternion usage

When the rotation quaternion appears *last, local space* is used (i.e. the rotation occurs around the axis in which the object was modelled).

Local space rotation

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



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 in the considered *zxy* order, the building blocks must 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 also be created:

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 created:

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

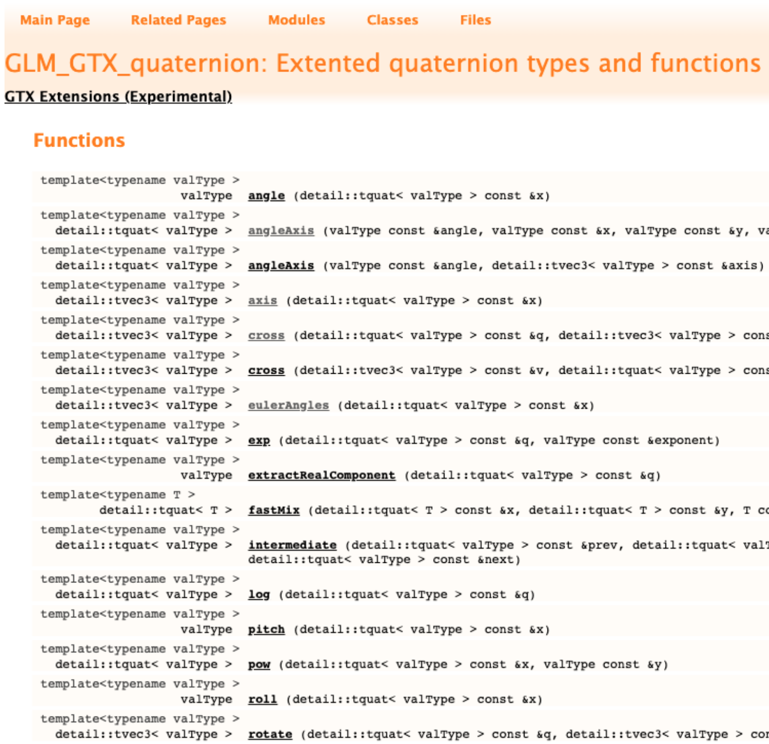
```
// create quaterion from Euler angles
glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));

// 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, faster construction and conversion procedures.

Will not consider them more in depth right now.



Main Page Related Pages Modules Classes Files

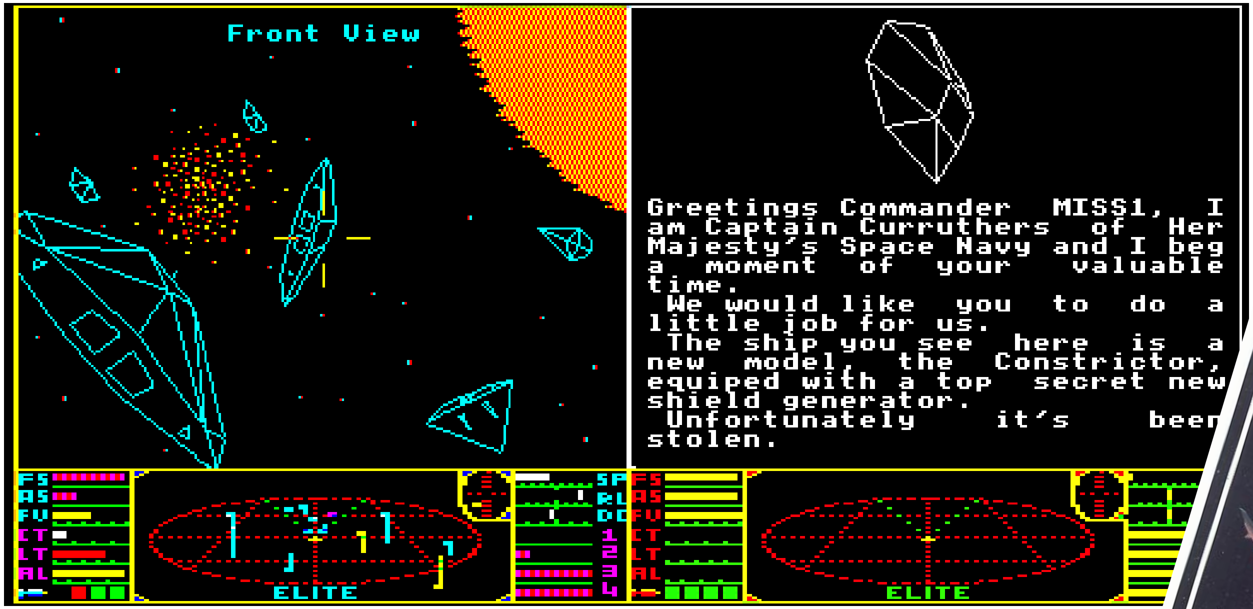
GLM_GTX_quaternion: Extented quaternion types and functions

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
```

World-View-Projection Wrap-up



Year: 1984

World-View-Projection Matrices

To obtain the position of the pixels on screen from the local coordinates that define a 3D model (as exported for example from a tool like Blender), five steps should be performed in a fixed order: *World Transform*, *View Transform*, *Projection*, *Normalization* and *Screen Transform*.

Each step transforms the coordinates from one space to another.

The first three steps (and possibly the last) can be implemented with a matrix-vector product.

Normalization instead requires a different procedure that cannot be integrated with the others.

World-View-Projection Matrices

As we have seen, an object is modeled in *local coordinates* p_M.

Usually local coordinates are 3D Cartesian, and they are first transformed into homogeneous coordinates p_L by adding a fourth component equal to one (*Step I.a*).

The *World Transform* converts the coordinates from *Local Space* to *Global Space*, by multiplying them with the *World Matrix* M_W (*Step I.b*).

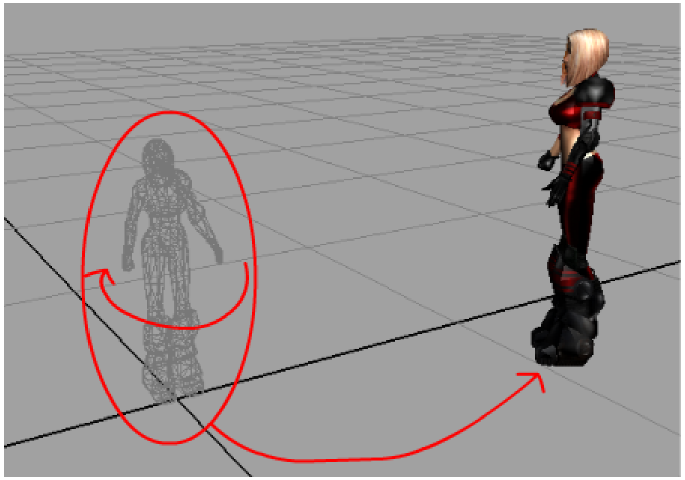
$$p_M = \left| \begin{matrix} p_{Mx} & p_{My} & p_{Mz} \end{matrix} \right|$$

Step I.a ↓

$$p_L = \left| \begin{matrix} p_{Mx} & p_{My} & p_{Mz} & 1 \end{matrix} \right|$$

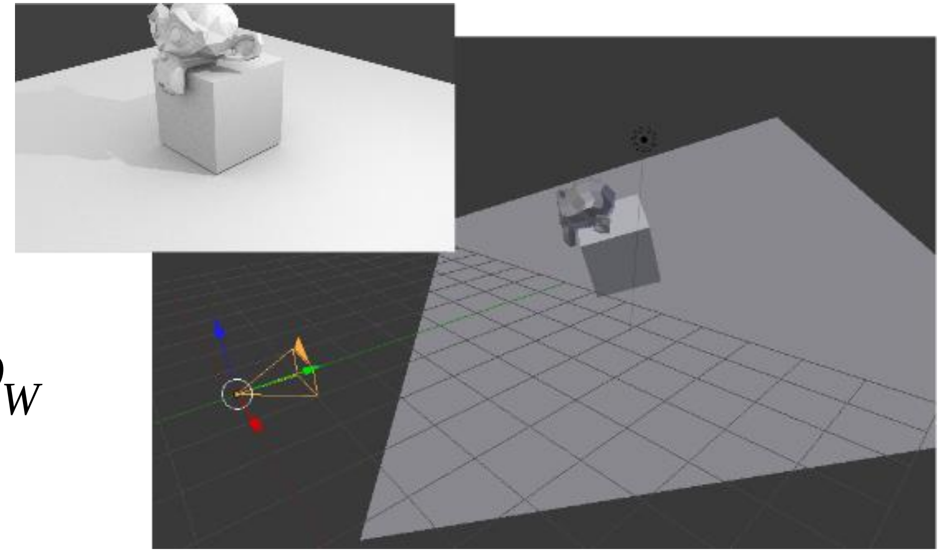
Step I.b ↓

$$p_W = M_W \times p_L$$



The *View transform* allows to see the 3D world from a given point in space.

It transform the coordinates from *Global Space* to *Camera Space* using the *View Matrix* M_V, usually created with either the *look-in-direction* or *look-at* techniques (*Step II*).



Step II

$$p_V = M_V \times p_W$$

World-View-Projection Matrices

The *Projection Transform* prepares the coordinates to be shown on screen by performing either a parallel or a perspective projection (*Step III*).

For parallel projections, it uses a *parallel projection matrix* $M_{P\text{-}Ort}$, and it converts *Camera Space Coordinates* to *Normalized Screen Coordinates*.

For perspective projections, a *perspective projection matrix* $M_{P\text{-}Persp}$ is used: in this case the results are not yet *Normalized Screen Coordinates*, but an intermediate space called *Clipping Coordinates*, for reasons that will be explained later.

Step III

$$p_C = M_P \times p_V$$

The World-View-Projection Matrix

In most of the cases the World, View and Projection matrices are factorized in a single matrix.

$$p_C = M_P \times M_V \times M_W \times p_L = M_{WVP} \times p_L$$

This combined matrix M_{WVP} is usually known as *the World-View-Projection Matrix*.

Step I-II-III

$$M_{WVP} = M_P \times M_V \times M_W$$

For perspective projections, the *Normalization* step produces the *Normalized Screen Coordinates* from the *Clipping Coordinates* (*Step IV*).

As opposed to the other transformations, this step is accomplished by converting the homogenous coordinates that describe the points in the clipping space, into the corresponding Cartesian ones.

In particular, every coordinate is divided by the fourth component of the homogenous coordinate vector. The last component (which will always be equal to one) is then discarded.

Step IV

$$\left| \begin{matrix} x_C & y_C & z_C & w_C \end{matrix} \right| \rightarrow \left| \begin{matrix} \frac{x_C}{w_C} & \frac{y_C}{w_C} & \frac{z_C}{w_C} & 1 \end{matrix} \right| = (x_n, y_n, z_n)$$

This step is not necessary in parallel projections, since in this case matrix $M_{P\text{-}Ort}$ already provides normalized screen coordinates: it sufficient to just drop the last component, which should already be equal to one.

World-View-Projection Matrices

A 3D application usually performs the World-View-Projection and sends to the underlay framework the *clipping coordinates* that define the primitives it wants to display.

$$p_C = M_{WVP} \times p_L$$

The driver of the video card converts the *clipping coordinates* first to *normalized screen coordinates* (if necessary), and then to *pixel coordinates* to visualize the objects (*Step V*). This is done in a way that is transparent to final user.

$$(x_n, y_n, z_n) = \left(\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c} \right)$$

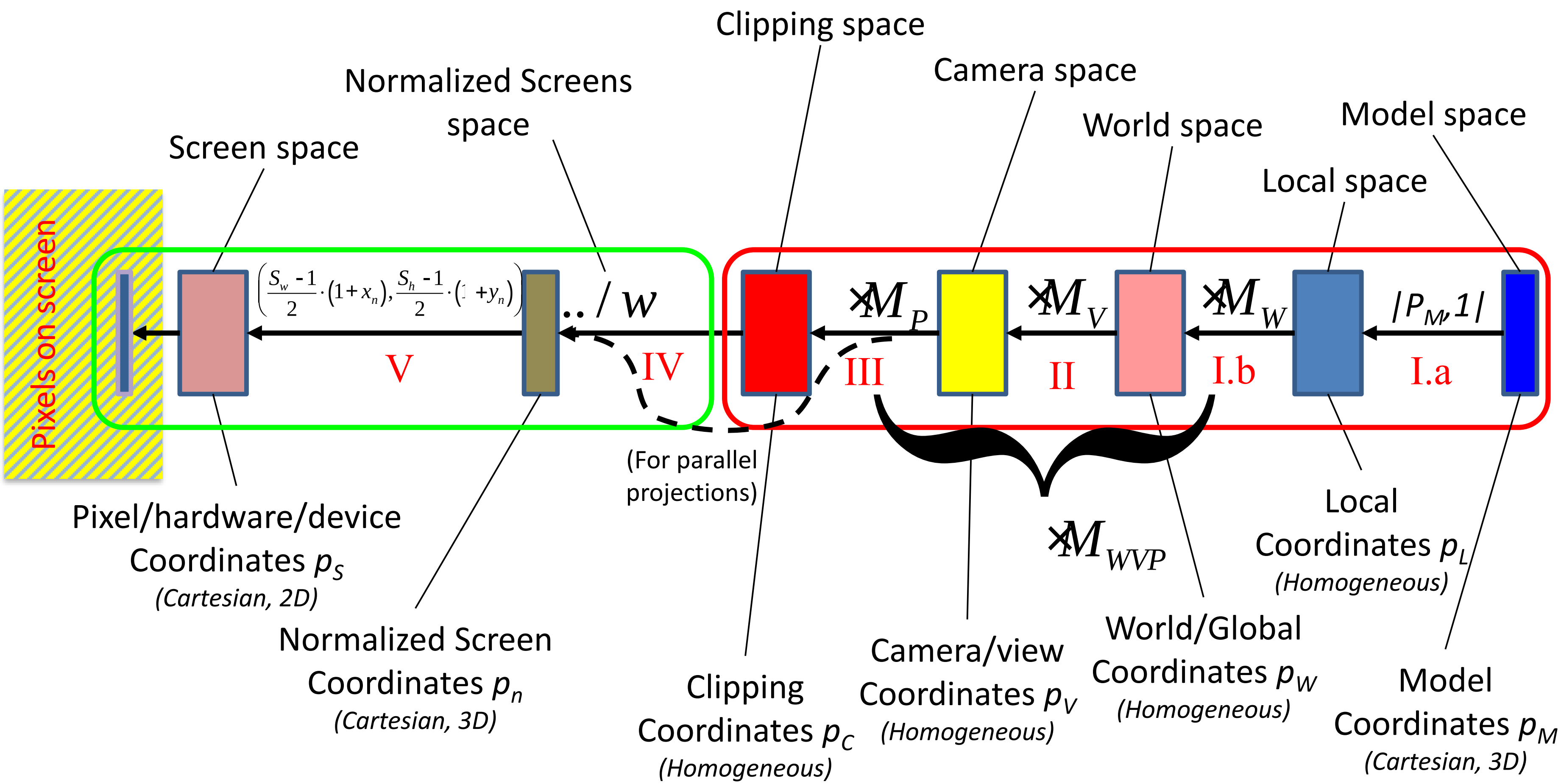
Step V ↓

$$(x_S, y_S) = \left(\frac{S_w - 1}{2} \cdot (1 + x_n), \frac{S_h - 1}{2} \cdot (1 - y_n) \right)$$

World-View-Projection Matrices: summary

Done by drivers

Done by the user



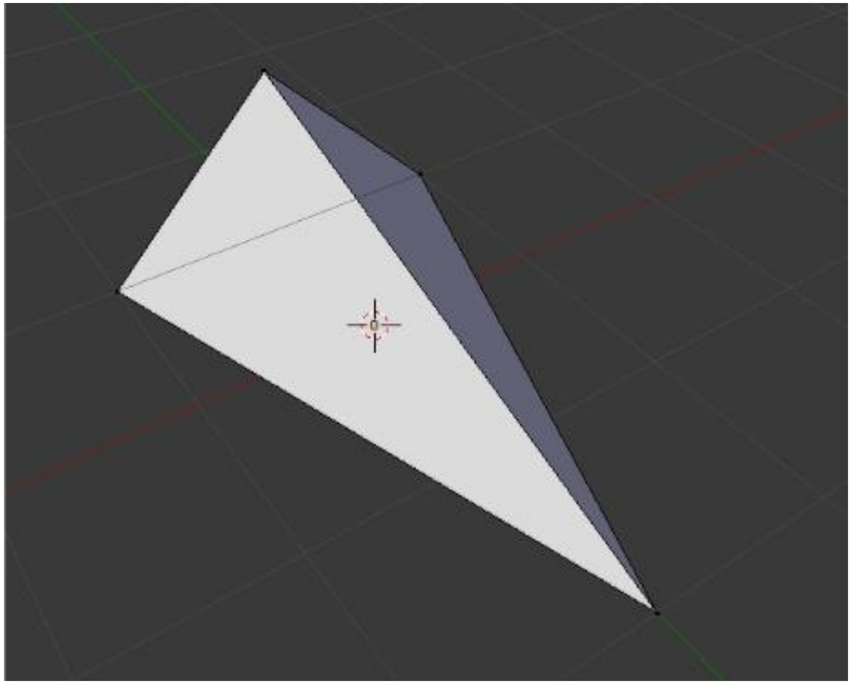
A complete projection example

Implementing the sequence of steps just presented, we can produce a 3D view of a scene, on any tool allowing to draw points, line or triangles on a surface, addressed by a set of bidimensional cartesian coordinates.

For example, this can be used to create a line based representation of a 3D object on a scatter-plot chart of Microsoft Excel.

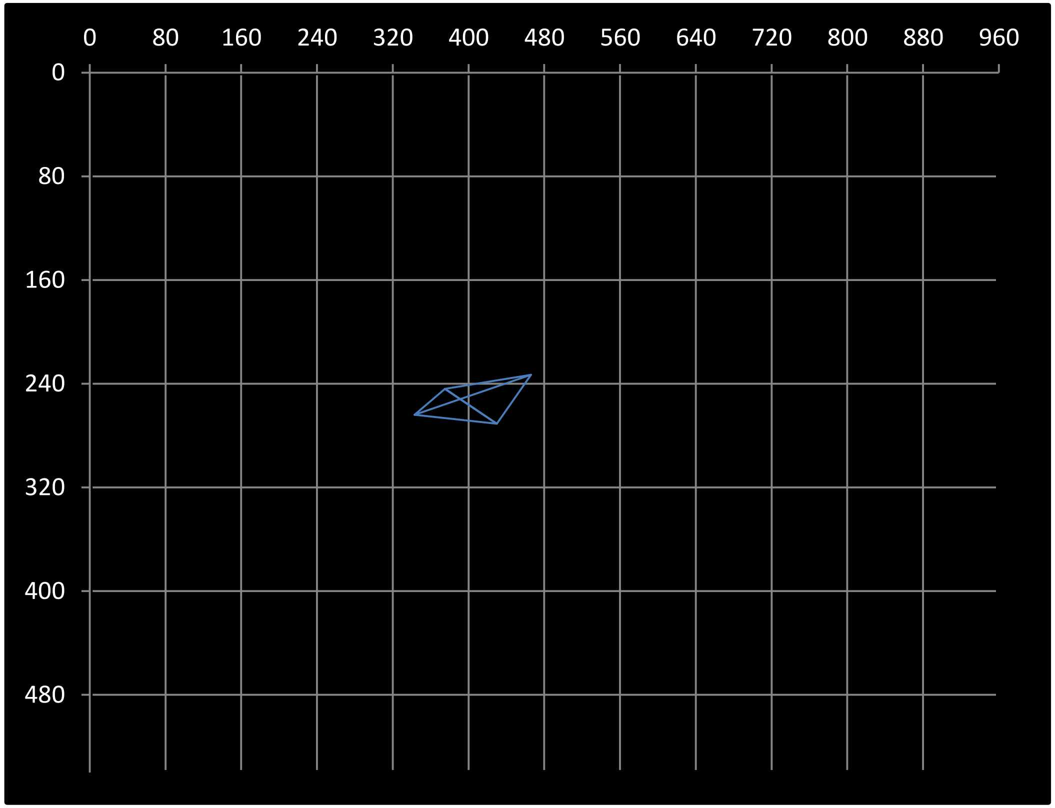
A complete projection example

Let’s try to show a simple starship modeled with a *tetrahedron*.



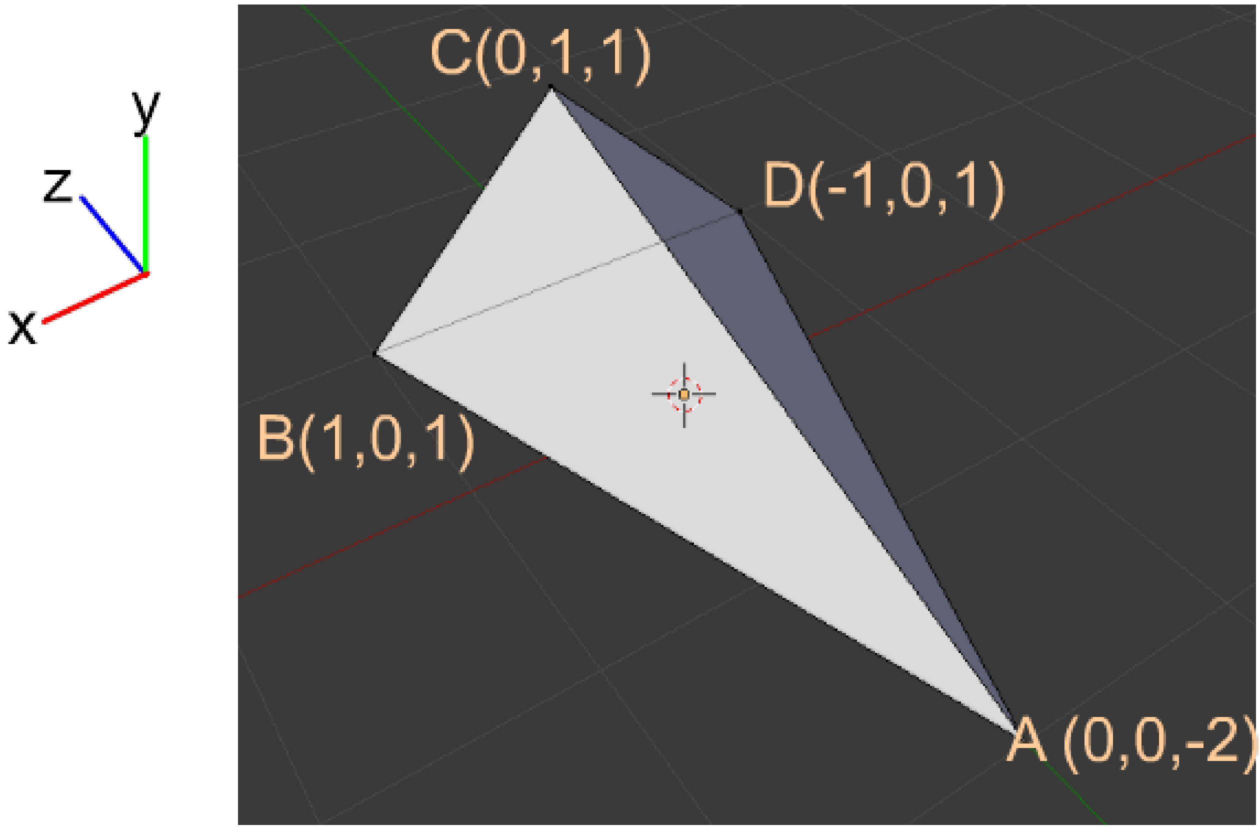
A complete projection example

To further simplify the visualization, the models is shown in wireframe mode (only its boundary).



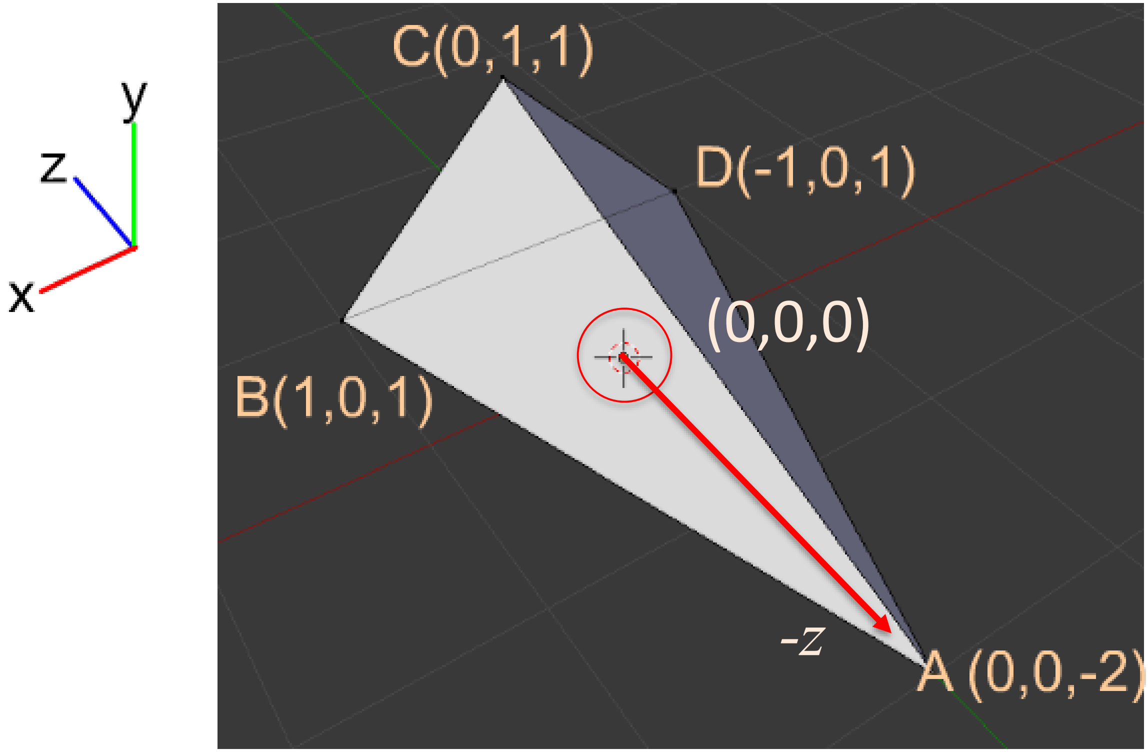
A complete projection example

The following local coordinates characterize the starship.



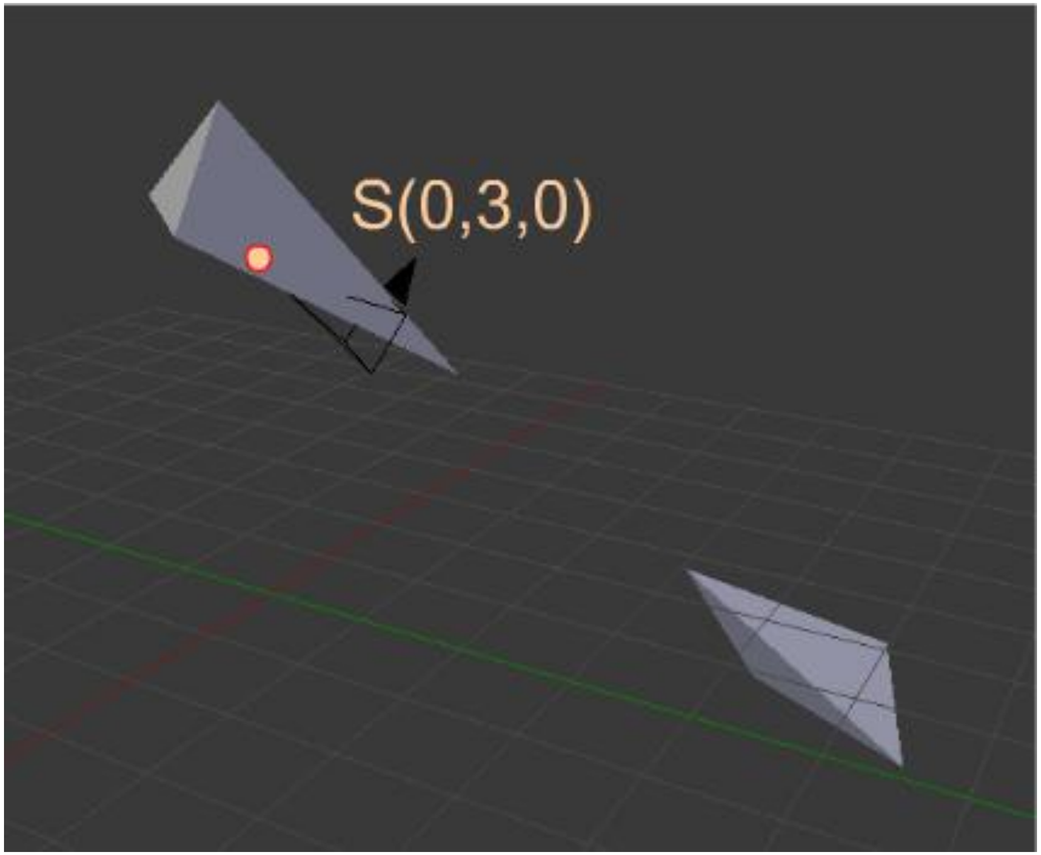
A complete projection example

Note that the model has been created oriented along the negative *z-axis*, with its center in the origin of the local coordinates system.



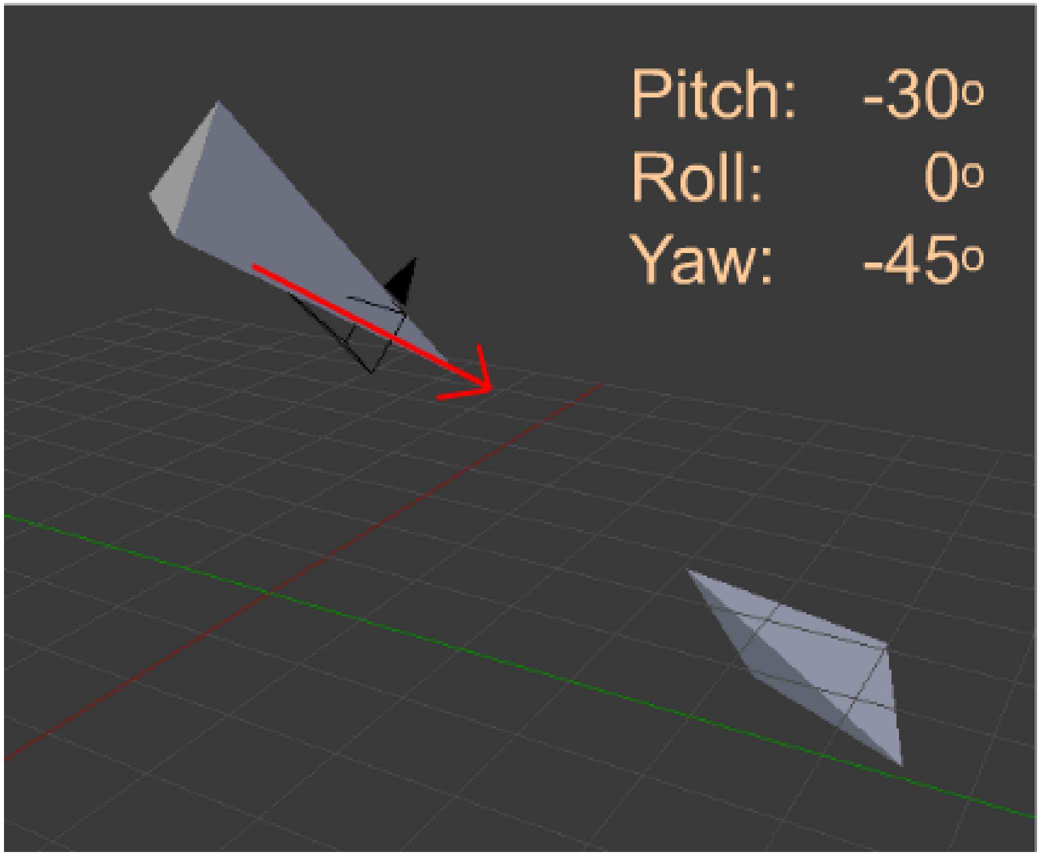
A complete projection example

The camera is located at:



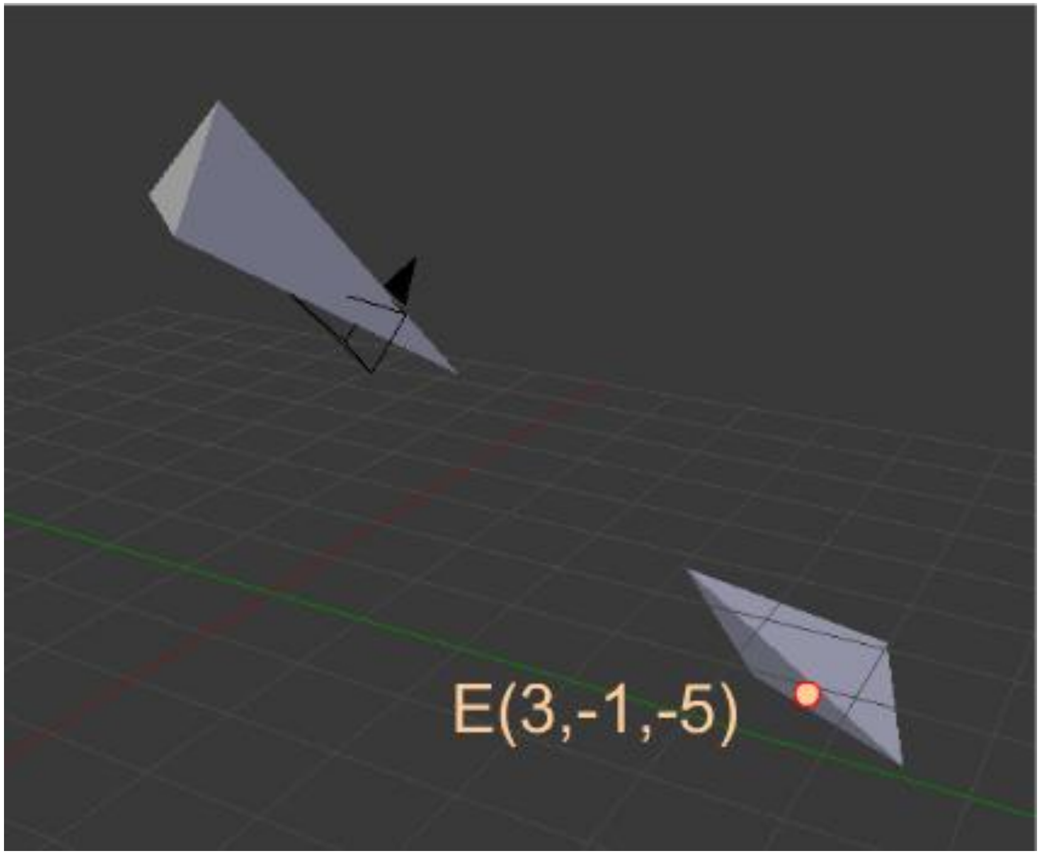
A complete projection example

And it is aiming in the following direction:



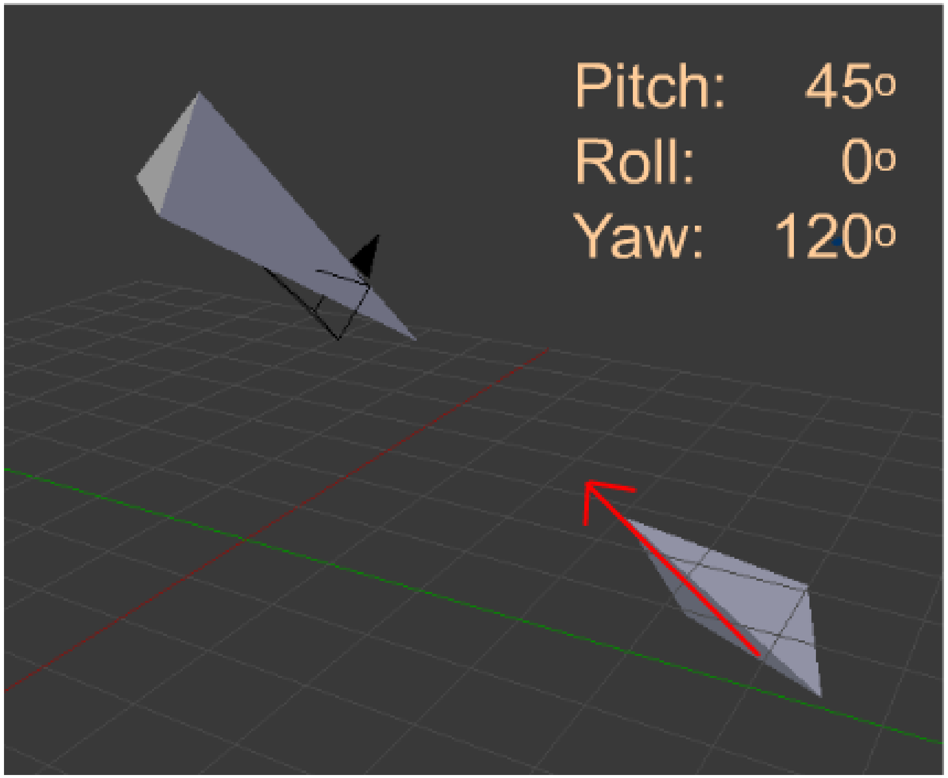
A complete projection example

The 3D ship is located at:



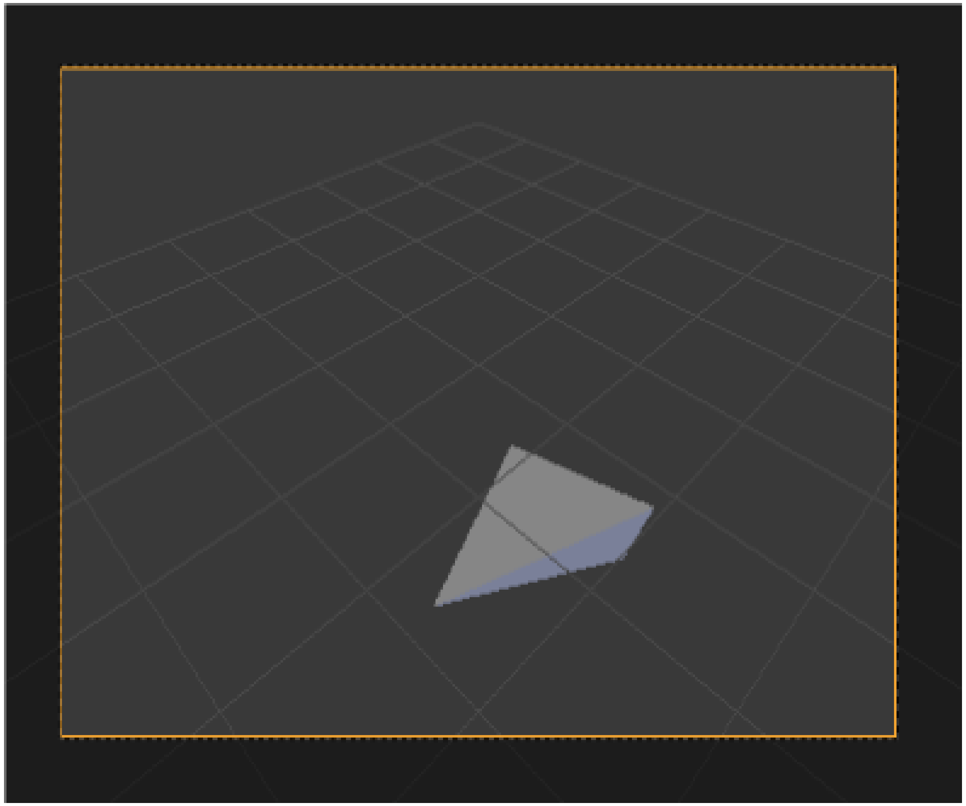
A complete projection example

And it is heading in the following direction



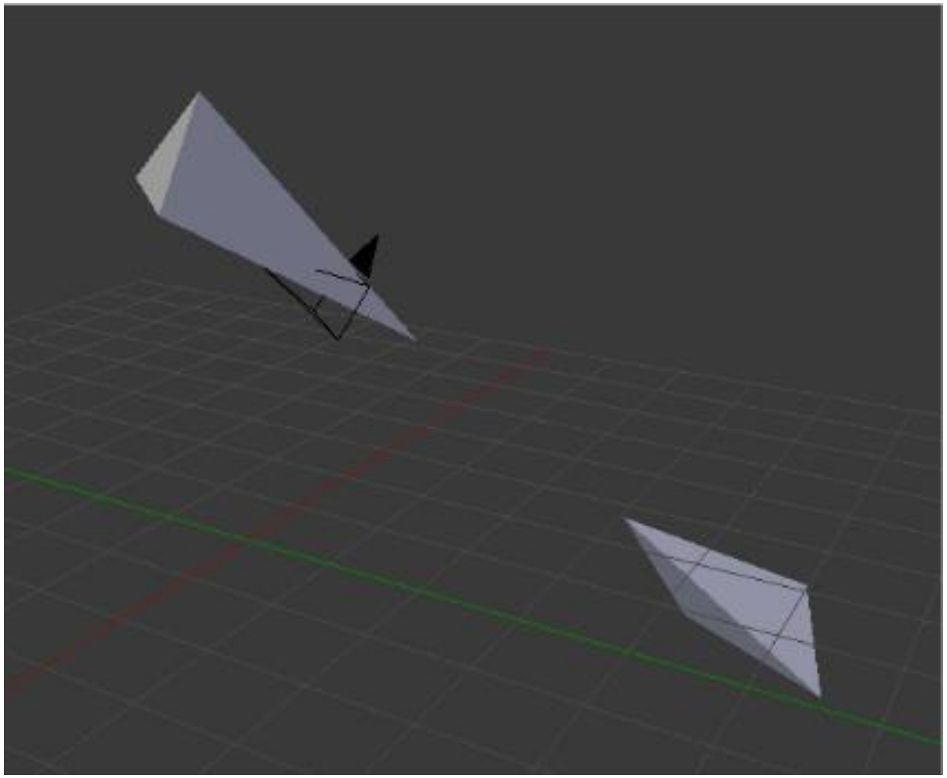
A complete projection example

The view of the player will be presented on scatter plot, simulating a 960x540 screen, with 5:4 aspect ratio and non-square pixels.



A complete projection example

The camera has a FOV of *90º*, and the *near* an *far* planes are respectively at: $n = 0.5$, $f = 9.5$.



A complete projection example

First we compute the *World Matrix* for the enemy ship using *Euler* angles:

World

Px	Py	Pz
3	-1	-5
Yaw	**Pitch**	**Roll**
120	45	0
Sx	**Sy**	**Sz**
1	1	1

T				
	1	0	0	3
	0	1	0	-1
	0	0	1	-5
	0	0	0	1

Ry				
	-0,5	0	0,87	0
	0	1	0	0
	-0,87	0	-0,5	0
	0	0	0	1

Rx				
	1	0	0	0
	0	0,71	-0,71	0
	0	0,71	0,71	0
	0	0	0	1

Rz				
	1	0	0	0
	0	1	0	0
	0	0	1	0
	0	0	0	1

S				
	1	0	0	0
	0	1	0	0
	0	0	1	0
	0	0	0	1

Step I

Mw				
	-0,5	0,61	0,61	3
	0	0,71	-0,71	-1
	-0,87	-0,35	-0,35	-5
	0	0	0	1

A complete projection example

Then we compute the View Matrix to account for the position of the player's ship using the *Look-In-Direction* technique:

View	Cx	Cy	Cz
	0	3	0
	Alfa	Beta	Rho
	-45	-30	0
	Yaw	Pitch	Roll

Rz				
	1	0	0	0
	0	1	0	0
	0	0	1	0
	0	0	0	1

Rx				
	1	0	0	0
	0	0,87	-0,5	0
	0	0,5	0,87	0
	0	0	0	1

Ry				
	0,71	0	0,71	0
	0	1	0	0
	-0,71	0	0,71	0
	0	0	0	1

T				
	1	0	0	0
	0	1	0	-3
	0	0	1	0
	0	0	0	1

Step II

Mv				
	0,71	0	0,71	0
	0,35	0,87	-0,35	-2,6
	-0,61	0,5	0,61	-1,5
	0	0	0	1

Please note that in this case we use the inverse transform matrix, in the opposite order, as required by the *Look-In-Direction* technique.

A complete projection example

We compute the projection matrix associated to the camera:

Perspective

FovY	a
90	1,25
n	**f**
0,5	9,5

Pp

0,8	0	0	0
0	1	0	0
0	0	-1,11	-1,06
0	0	-1	0

This Projection matrix follows the conventions required by OpenGL instead of Vulkan. Nevertheless, it is equally useful in understanding the complete process.

Step III

A complete projection example

We combine them together in the *World-View-Projection Matrix (WVP Matrix)*:

0,8	0	0	0
0	1	0	0
0	0	-1,11	-1,06
0	0	-1	0

*

0,71	0	0,71	0
0,35	0,87	-0,35	-2,6
-0,61	0,5	0,61	-1,5
0	0	0	1

*

-0,5	0,61	0,61	3
0	0,71	-0,71	-1
-0,87	-0,35	-0,35	-5
0	0	0	1

=

World-View-Projection

-0,7727	0,15	0,15	-1,13
0,1294	0,95	-0,27	-0,64
0,249	0,26	1,05	6,61
0,2241	0,24	0,95	6,9

Step I-II-III

Please note how the "-1" off diagonal in the projection matrix makes the WVP matrix with all 16 elements different from zero and one.

A complete projection example

We multiply the local coordinates of the vertices of the tetrahedron, obtained by adding a fourth component equal to one, with *world-view-projection matrix*, and we divide by *w*. Finally, we compute the screen coordinates and find the closest integers to the pixel corresponding to the vertices of enemy ship.

Points

A	B	C	D
0	1	0	-1
0	0	1	0
-2	1	1	1
1	1	1	1

*

-0,7727	0,15	0,15	-1,13
0,1294	0,95	-0,27	-0,64
0,249	0,26	1,05	6,61
0,2241	0,24	0,95	6,9

=

Clipping Coordinates

-1,4242	-1,7577	-0,84	-0,21
-0,0939	-0,7771	0,05	-1,04
4,5098	7,9091	7,92	7,41
5,0089	8,0682	8,08	7,62

Step IV

Normalized Screen Coordinates

-0,28	-0,22	-0,10	-0,03
-0,02	-0,10	0,01	-0,14
0,90	0,98	0,98	0,97

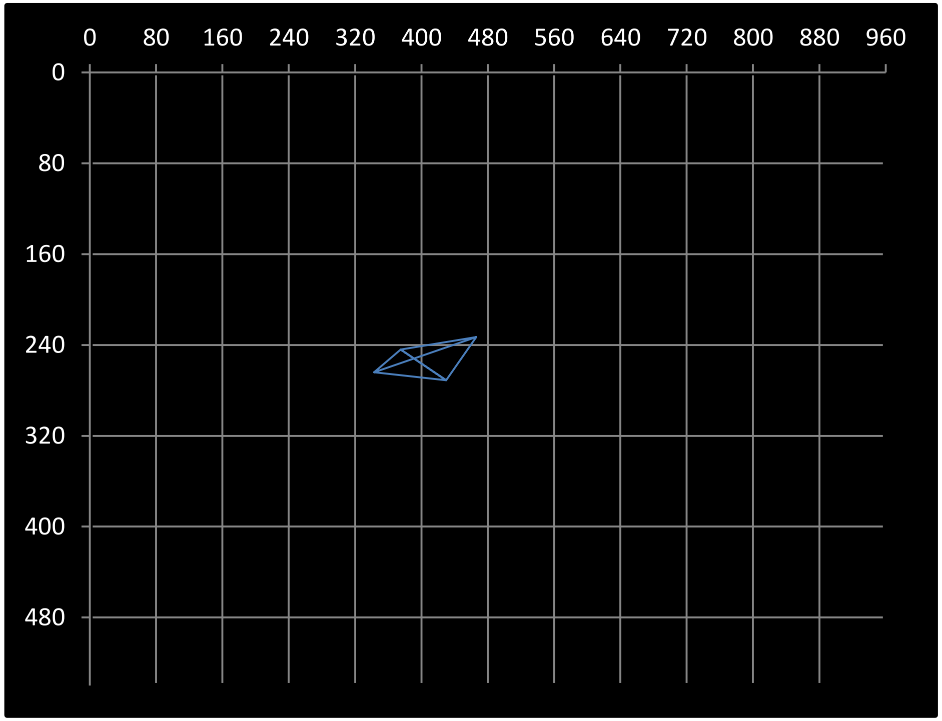
Pixel Coordinates

343	375	430	466
264	244	271	233

Step V

A complete projection example

We can then connect the four points with six lines (lines AB, AC, AD, BC, CD, DB), to produce a 2D representation of the considered 3D object.



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)