POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

D E
I B

2023

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2023

Computer Graphics

- Advanced Transformations

Inversion of transformations

Transformations can be inverted to return a transformed object to its original position, size or orientation.

A nice feature about using matrices is that the inverse transformation can be obtained by inverting the corresponding matrix:

- If point *p'* is obtained by applying the transformation coded into matrix *M* to point *p*, then point *p* can be obtained back from point *p'* by multiplying it with M^{-1}, the inverse of the matrix *M*.

$$p = (x, y, z, 1)$$

$$p' = M \cdot p$$

$$p' = (x', y', z', 1)$$

$$p' = (x', y', z', 1)$$

$$p = M^{-1} \cdot p'$$

$$p = (x, y, z, 1)$$

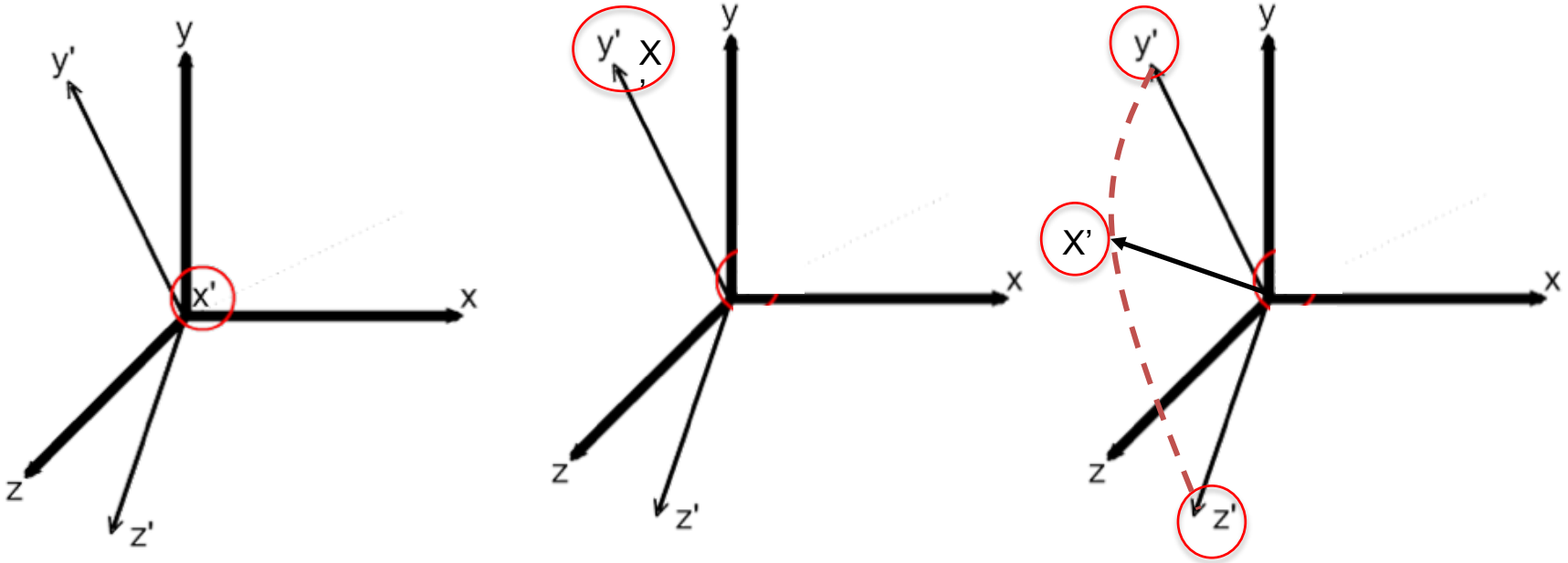
Inversion of transformations

Since the last row is |0 0 0 1|, it can be proven that a transform matrix M is invertible if its sub-matrix composed by the first 3 rows and 3 columns is invertible.

$$M = \left|\begin{array}{ccc|c} n_{xx} & n_{yx} & n_{zx} & d_x \\ n_{xy} & n_{yy} & n_{zy} & d_y \\ n_{xz} & n_{yz} & n_{zz} & d_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right|$$

Inversion of transformations

This is generally the case, except when for some reasons one or more of the projected axis degenerates to be either of zero length, or when two axis perfectly overlaps, or when one axis is aligned with the plane defined by the other two.



Inversion of transformations

The inverse of a general transformation matrix can be obtained by applying numerical inversion techniques
Closed formulas also exists.

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

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

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

$$\mathrm{adj}(\mathbf{A}) = \begin{pmatrix} +\begin{vmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{vmatrix} & -\begin{vmatrix} A_{12} & A_{13} \\ A_{32} & A_{33} \end{vmatrix} & +\begin{vmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \end{vmatrix} \\ -\begin{vmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{vmatrix} & +\begin{vmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{vmatrix} & -\begin{vmatrix} A_{11} & A_{13} \\ A_{21} & A_{23} \end{vmatrix} \\ +\begin{vmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{vmatrix} & -\begin{vmatrix} A_{11} & A_{12} \\ A_{31} & A_{32} \end{vmatrix} & +\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \end{pmatrix}$$

$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$\det(A) = ad - bc$

det(A)= a11(a22*a33 - a32*a23) - a21(a12*a33 - a32*a13) + a31(a12*a23 - a22*a13)

$$A^{-1} = \frac{adj\,(A)}{det\,(A)}$$

However, if we want to the invert some of the transformations previously presented, we have simple matrix patterns that we can use to compute them immediately.

Inversion of transformations

For translation:

$$T(d_x, d_y, d_z) = \begin{vmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$\left[T(d_x, d_y, d_z)\right]^{-1} = T(-d_x, -d_y, -d_z) = \begin{vmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Inversion of transformations

For scaling:

$$S(s_x, s_y, s_z) = \begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$\left[S(s_x, s_y, s_z)\right]^{-1} = S(1/s_x, 1/s_y, 1/s_z) = \begin{vmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Note that *1/s* can be computed only if *s != 0*.

This explains why a transformation cannot be inverted if the length of one axis is reduced to 0.

Inversion of transformations

For rotation:

$$R_x(\alpha) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad \left[R_x(\alpha)\right]^{-1} = R_x(-\alpha) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$R_y(\alpha) = \begin{vmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad \left[R_y(\alpha)\right]^{-1} = R_y(-\alpha) = \begin{vmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$R_z(\alpha) = \begin{vmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad \left[R_z(\alpha)\right]^{-1} = R_z(-\alpha) = \begin{vmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Inversion of transformations

Example:

Consider a scaling of 2.5 on the y-axis and 1/3 on z-axis. The transform matrix and its inverse are the following:

$$S(1,2.5,1/3) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 2.5 & 0 & 0 \\ 0 & 0 & 0.33333 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$\left[S(1,2.5,1/3)\right]^{-1} = S(1,1/2.5,1/(1/3)) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Composition of transformations

During the creation of a scene, an object is subject to several transformations.

The application of a sequence of transformation is called **composition**.

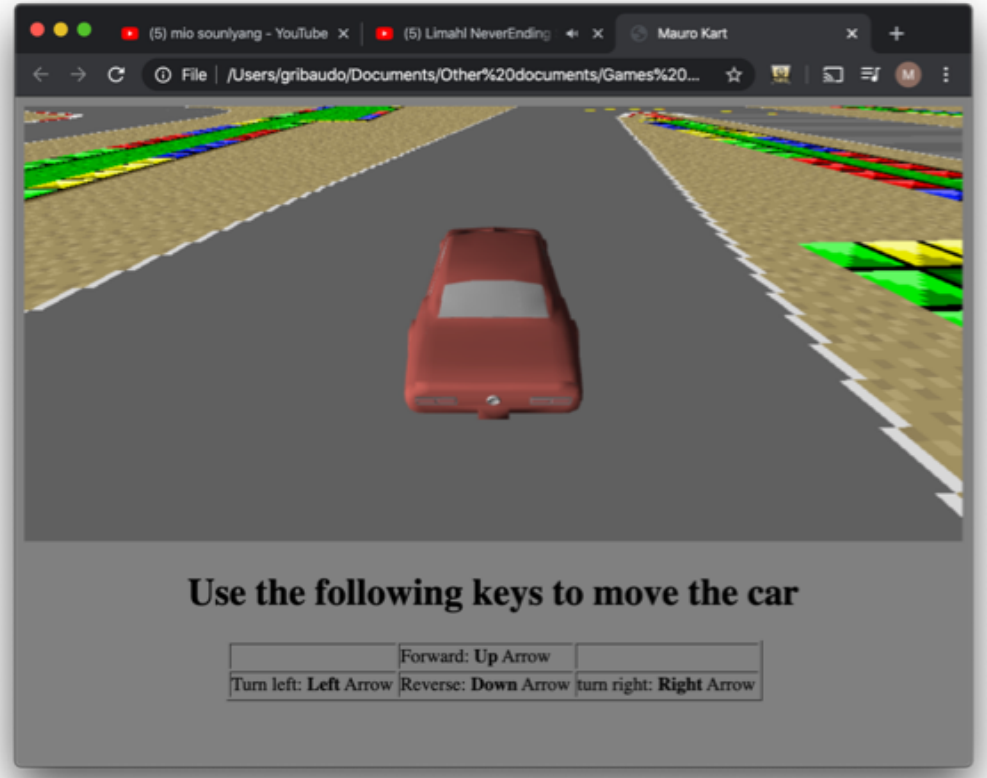
Usually, an object should be translated and rotated in all directions to be positioned in the scene.

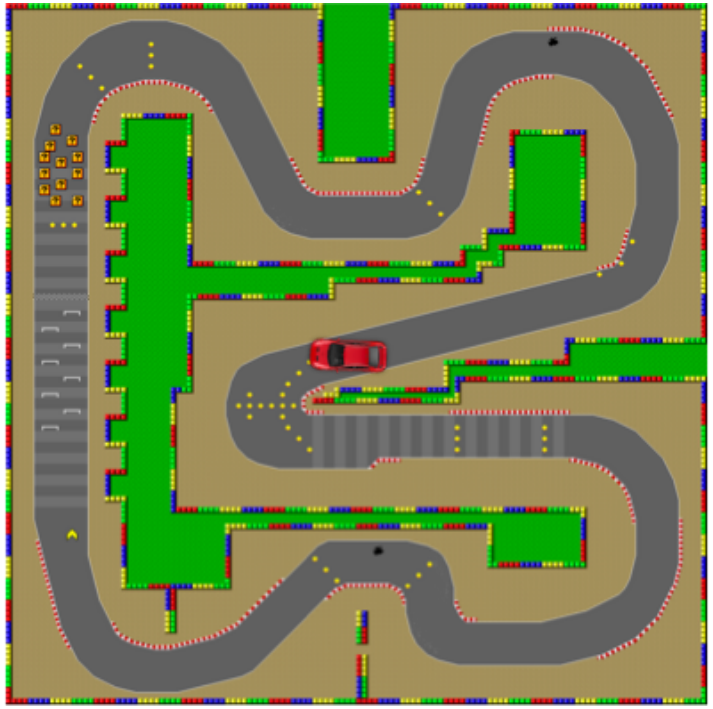
Moreover, rotations among arbitrary axis and scaling with different centers can be performed by composing different transformations.

Thanks to the properties of matrix product, composition of transformations can be be done in a very efficient way.

Composition of transformations

Let us consider the problem of placing a car on a track.

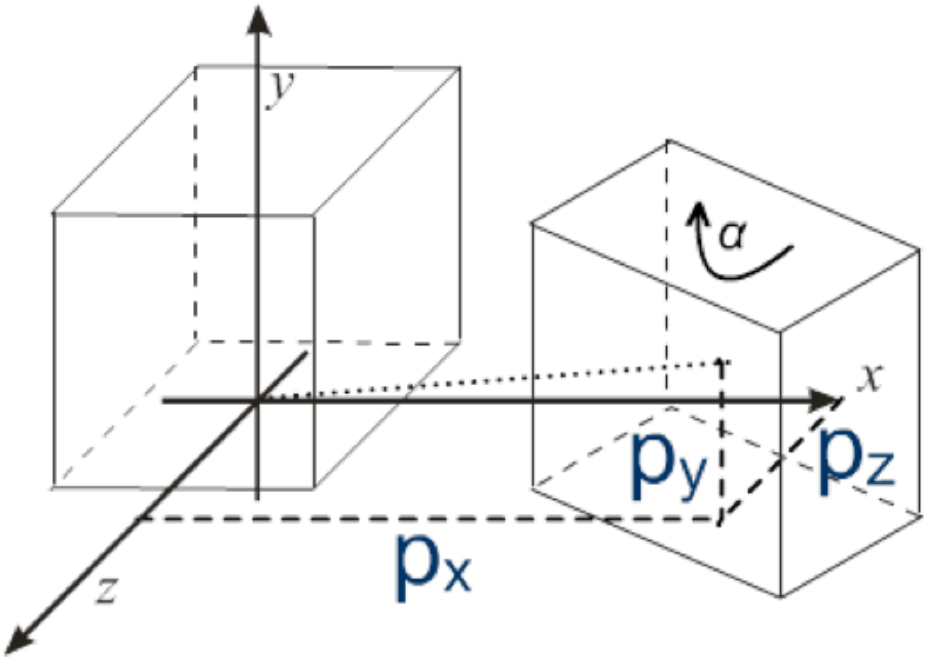




Composition of transformations

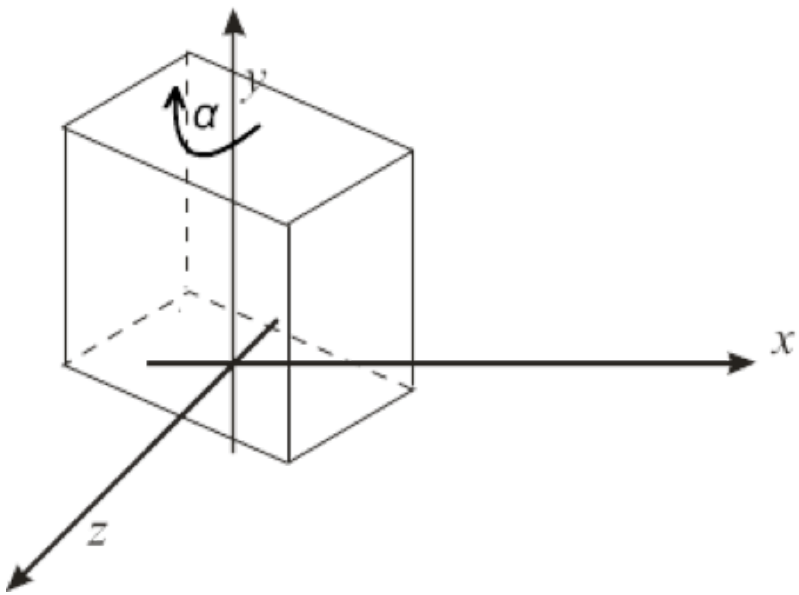
In general, we have to place its center in position (p_x, p_y, p_z), and to orient its side at an angle α around the *y* axis.

To simplify the presentation, we will display the car as a cube, with its sides parallel to the *x*, *y*, and *z* axes, and with its center in the origin.



Composition of transformations

The goal can be reached by first rotating all the vertices of the car, of an angle α around the *y*-axis (which can be obtained by a rotation matrix $R_y(\alpha)$).

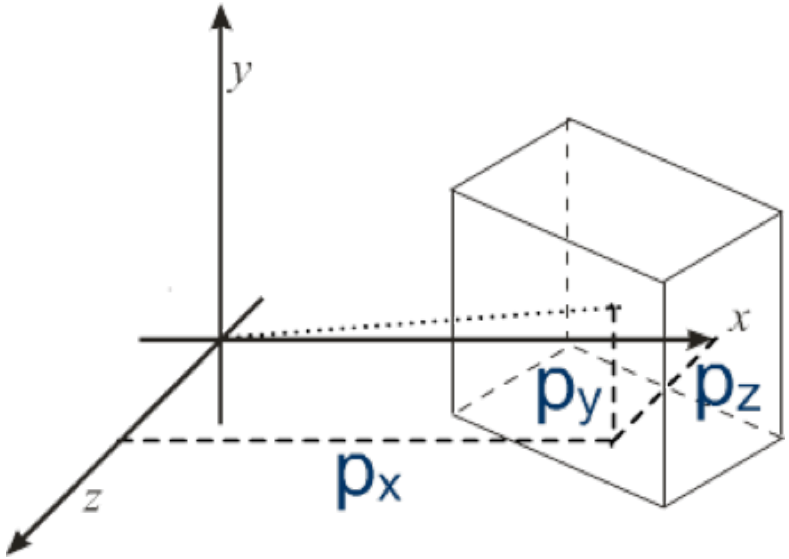


$$R_y(\alpha) = \begin{vmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$p' = R_y(\alpha) \cdot p$$

Composition of transformations

Then the rotated car can be translated in position (p_x, p_y, p_z) using a translation matrix $T(p_x, p_y, p_z)$.



$$T(p_x, p_y, p_z) = \begin{vmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

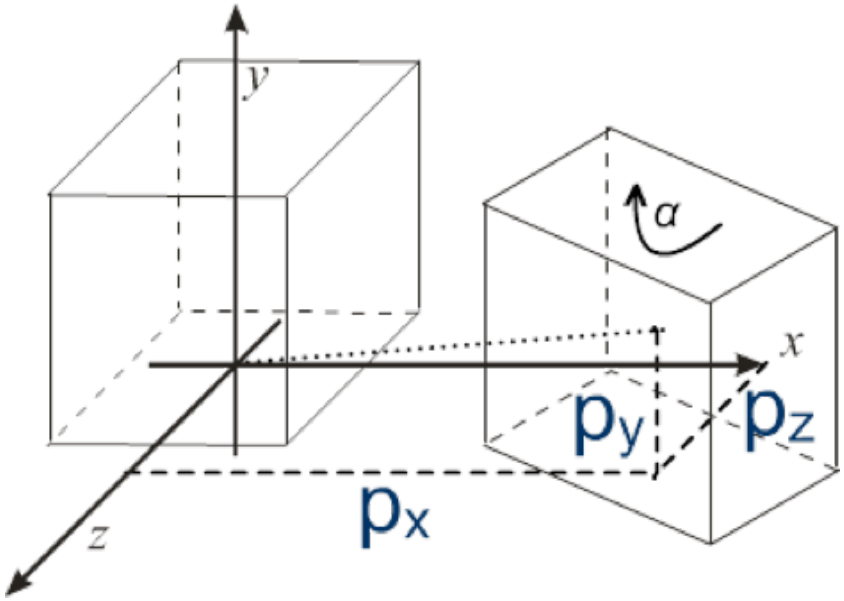
$$p' = R_y(\alpha) \cdot p$$

$$p'' = T(p_x, p_y, p_z) \cdot p' = T(p_x, p_y, p_z) \cdot R_y(\alpha) \cdot p$$

Please note that the performing rotation first and translation next, it would not work! We will return on this in the following.

Composition of transformations

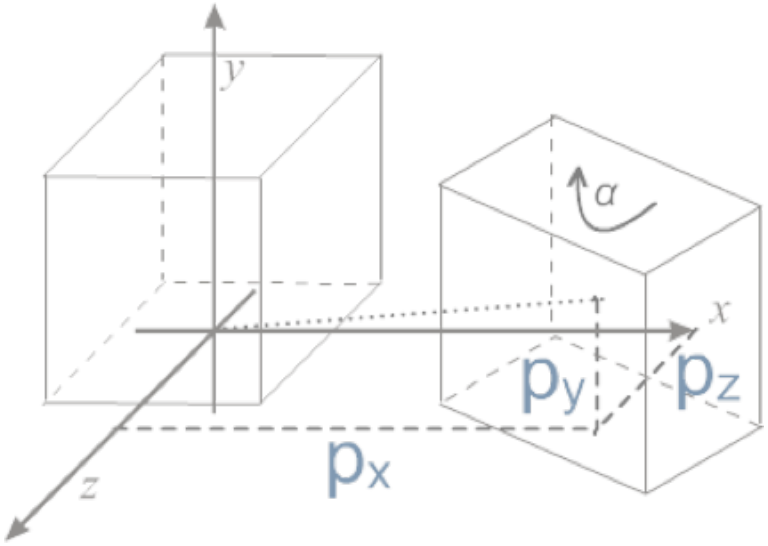
Note that, as in functional programming, matrices appear inside the expression in reverse order with respect to the transformations they represent.



$$p'' = T(p_x, p_y, p_z) \cdot R_y(\alpha) \cdot p$$

Composition of transformations

(In case of the *matrix-on-the-right* convention the order of transformation would be from left to right)

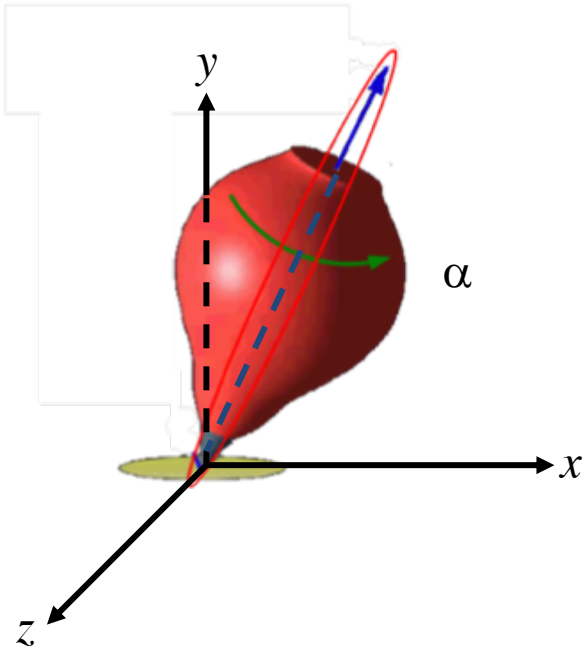


In this course, we will never use the matrix-on-the-right convention.

$$p'' = p \cdot R_y(\alpha)^T \cdot T(p_x, p_y, p_z)^T$$

Transformations around an arbitrary axis or center

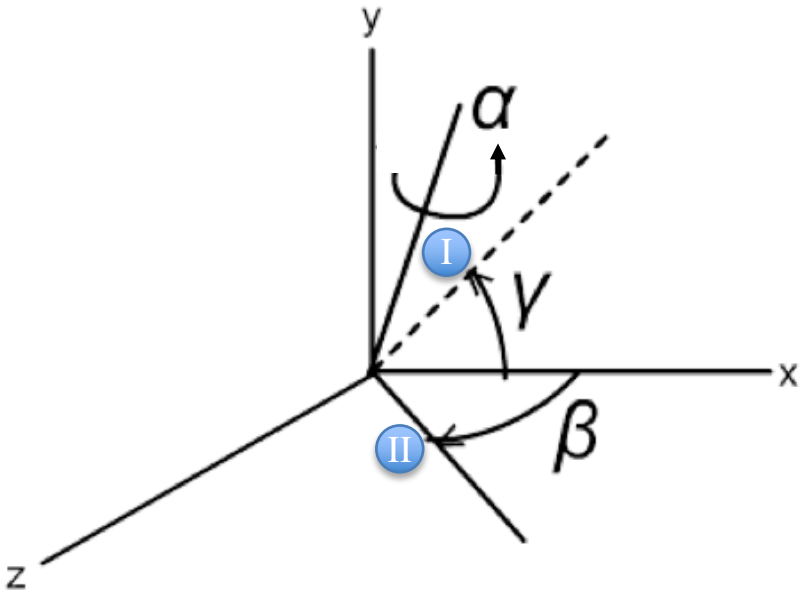
Now consider the rotation of an object of an angle α about an arbitrary axis that passes through the origin.



Transformations around an arbitrary axis or center

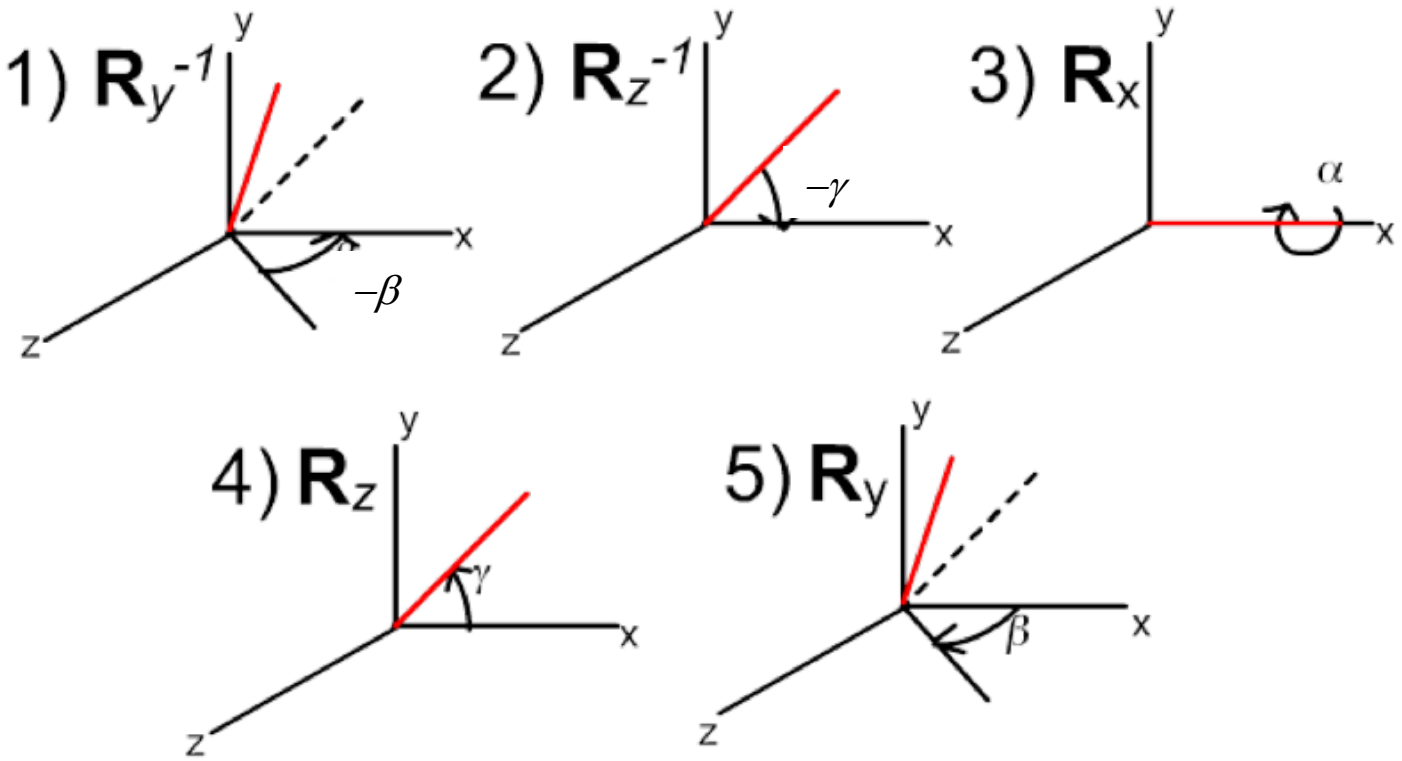
Let us focus on a case where the direction of the arbitrary axis can be expressed with a couple of angles.

In particular, the x-axis can be aligned to the arbitrary axis by first rotating an angle γ around the z-axis, and then an angle β around the y-axis.



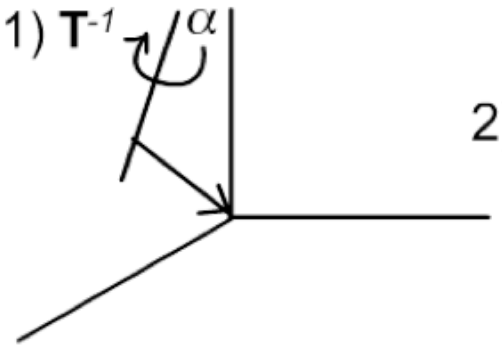
Transformations around an arbitrary axis or center

The arbitrary axis rotation can be obtained by combining 5 elementary transformations:

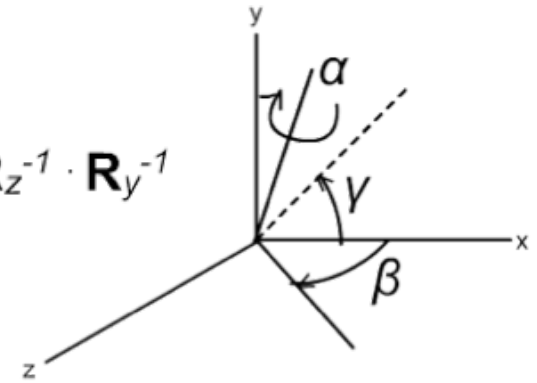


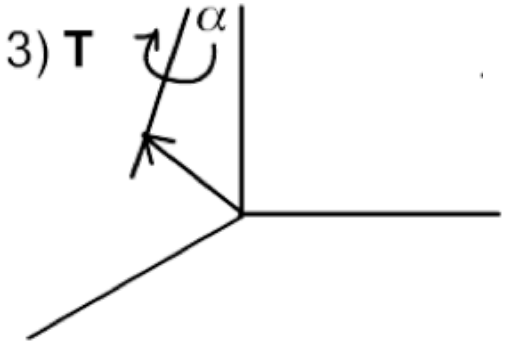
Transformations around an arbitrary axis or center

If the axis does not pass through the origin, a translation T^{-1} must be applied to make it pass through (0,0,0), and then at the end use T to restore the initial position.



2) $\mathbf{R}_y \cdot \mathbf{R}_z \cdot \mathbf{R}_x \cdot \mathbf{R}_z^{-1} \cdot \mathbf{R}_y^{-1}$





Transformations around an arbitrary axis or center

To summarize, a rotation of α about an arbitrary axis passing through point (p_x, p_y, p_z) and such that the x-axis can be aligned to it by first rotating an angle γ around the z-axis, and then an angle β around the y-axis, can be computed as:

$$p' = T(p_x, p_y, p_z) \cdot R_y(\beta) \cdot R_z(\gamma) \cdot R_x(\alpha) \cdot R_z(\gamma)^{-1} \cdot R_y(\beta)^{-1} \cdot T(p_x, p_y, p_z)^{-1} \cdot p$$

Similar procedures can be followed if we know different rotation sequences that would align another of the three main axis to the arbitrary one.

Transformations around an arbitrary axis or center

The same considerations can be applied for scaling an object along arbitrary directions, and with an arbitrary center.

They can also be applied to generalize shear, and to perform symmetries about arbitrary planes, axes or centers.

Since there is an extremely large variety of possibilities, determining the right sequence of transformations to obtain a specific result is left to the programmer.

Transformations around an arbitrary axis or center

Example:

A sequence of transformations that performs a uniform scaling of 2, centered at (3,2,1) is the following:

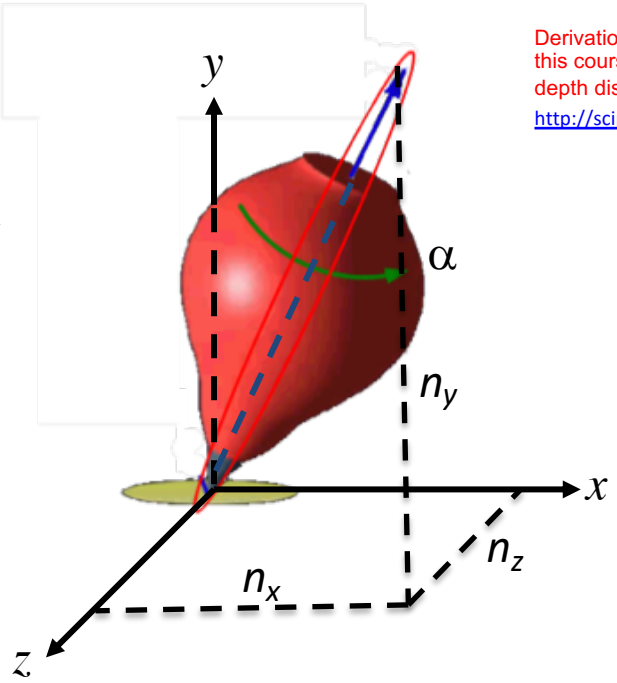
$$p' = T(p_x, p_y, p_z) \cdot S(s_x, s_y, s_z) \cdot T(p_x, p_y, p_z)^{-1} \cdot p$$

$$p' = T(3,2,1) \cdot S(2,2,2) \cdot T(3,2,1)^{-1} \cdot p$$

$$p' = \begin{vmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{vmatrix} \cdot p$$

Axis-angle rotation matrix

In many circumstances, the rotation axis can be expressed with a unitary vector $\boldsymbol{n} = (n_x, n_y, n_z)$ where $n_x^2 + n_y^2 + n_z^2 = 1$. In this case, it is possible to determine the rotation matrix in the following pattern:



Derivation of the result is outside the scope of this course. Interested students can find a in-depth discussion here:
http://scipp.ucsc.edu/~haber/ph216/rotation_12.pdf

$$\mathrm{R}(\mathbf{n},\alpha) = \begin{vmatrix} \cos\alpha + n_x^2(1-\cos\alpha) & n_x n_y(1-\cos\alpha) - n_z\sin\alpha & n_x n_z(1-\cos\alpha) + n_y\sin\alpha & 0 \\ n_x n_y(1-\cos\alpha) + n_z\sin\alpha & \cos\alpha + n_y^2(1-\cos\alpha) & n_y n_z(1-\cos\alpha) - n_x\sin\alpha & 0 \\ n_x n_z(1-\cos\alpha) - n_y\sin\alpha & n_y n_z(1-\cos\alpha) + n_x\sin\alpha & \cos\alpha + n_z^2(1-\cos\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Matrix transformations: is it worth it?

One of the question that usually comes up when seeing transformation matrices for the first time is: “is it worth it?”

For example, in the car example just given, if I know the original position of a vertex of the object, (x, y, z), wouldn’t be easier to compute it directly?

$$\begin{cases} x' = p_x + x \cdot \cos\alpha - z \cdot \sin\alpha \\ y' = p_y + y \\ z' = p_z + z \cdot \cos\alpha + x \cdot \sin\alpha \end{cases}$$

Properties of composition of transformations

The product of two matrices, and of a matrix and a vector is associative:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

A, B and C are 4x4 matrices.

$$A \cdot (B \cdot p) = (A \cdot B) \cdot p$$

p is a vector

We can exploit this feature to factorize all the transforms in a single matrix.

Properties of composition of transformations

Using the associative property, a single matrix corresponding to the product of all the transformations can be computed.

Then the composed matrix can be used to apply all the transformations in a single product.

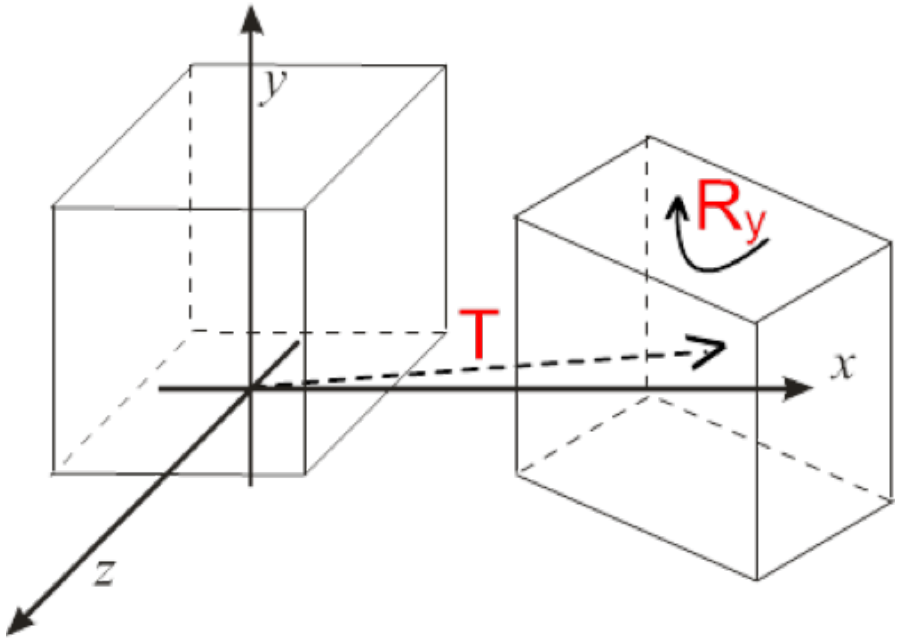
$$p' = T(p_x, p_y, p_z) \cdot R_y(\beta) \cdot R_z(\gamma) \cdot R_x(\alpha) \cdot R_z(\gamma)^{-1} \cdot R_y(\beta)^{-1} \cdot T(p_x, p_y, p_z)^{-1} \cdot p$$

$$R_{p_x, p_y, p_z, \beta, \gamma}(\alpha) = T(p_x, p_y, p_z) \cdot R_y(\beta) \cdot R_z(\gamma) \cdot R_x(\alpha) \cdot R_z(\gamma)^{-1} \cdot R_y(\beta)^{-1} \cdot T(p_x, p_y, p_z)^{-1}$$

$$p' = R_{p_x, p_y, p_z, \beta, \gamma}(\alpha) \cdot p$$

Properties of composition of transformations

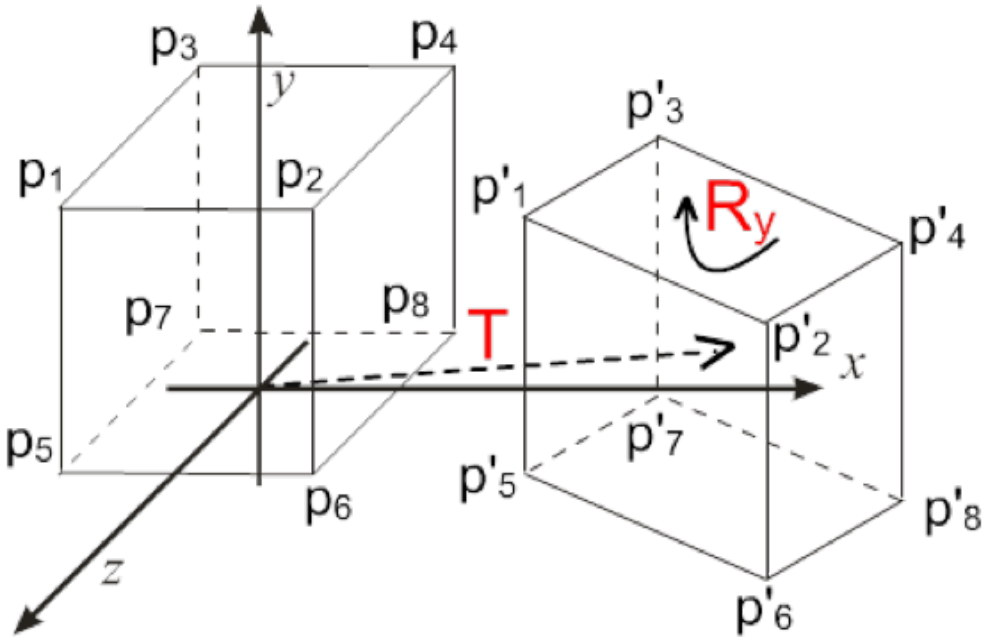
Computing the composed matrix greatly improves the performance of executing multiple transformations.



$$p' = T \cdot R_y \cdot p$$

Properties of composition of transformations

Usually, an object is composed by a large number of points (in current applications, between 10^4 and 10^6) that require the same transformation.



$$p'_1 = T \cdot R_y \cdot p_1$$
$$p'_2 = T \cdot R_y \cdot p_2$$
$$\vdots$$
$$p'_8 = T \cdot R_y \cdot p_8$$

Properties of composition of transformations

The transformation matrix is computed once per object.

Then all the n_v points are transformed by multiplying all of them with the same transformation matrix.

$$p'_1 = T \cdot R_y \cdot p_1$$
$$p'_2 = T \cdot R_y \cdot p_2$$
$$\vdots$$
$$p'_8 = T \cdot R_y \cdot p_8$$

2 n_v MxV products

$$M = T \cdot R_y$$
$$p'_1 = M \cdot p_1$$
$$p'_2 = M \cdot p_2$$
$$\vdots$$
$$p'_8 = M \cdot p_8$$

n_v (+4) MxV products

[the +4 accounts for the computation required to define M]

Composition of transformation: inversion

Recall that the inverse of the product of two matrices, is the product of the inverse matrices in the opposite order:

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

This can be used to compute the inverse of a composed transform.

Example:

$$\left[R_{p_x,p_y,p_z,\beta,\gamma}(\alpha)\right]^{-1} = \left[T(p_x,p_y,p_z)\cdot R_y(\beta)\cdot R_z(\gamma)\cdot R_x(\alpha)\cdot R_z(\gamma)^{-1}\cdot R_y(\beta)^{-1}\cdot T(p_x,p_y,p_z)^{-1}\right]^{-1} =$$

$$= \left[T(p_x,p_y,p_z)^{-1}\right]^{-1}\cdot\left[R_y(\beta)^{-1}\right]^{-1}\cdot\left[R_z(\gamma)^{-1}\right]^{-1}\cdot R_x(\alpha)^{-1}\cdot R_z(\gamma)^{-1}\cdot R_y(\beta)^{-1}\cdot T(p_x,p_y,p_z)^{-1} =$$

$$= T(p_x,p_y,p_z)\cdot R_y(\beta)\cdot R_z(\gamma)\cdot R_x(\alpha)^{-1}\cdot R_z(\gamma)^{-1}\cdot R_y(\beta)^{-1}\cdot T(p_x,p_y,p_z)^{-1} = R_{p_x,p_y,p_z,\beta,\gamma}(-\alpha)$$

Composition of transformation: inversion

Example:

An object is translated of *(1, 2, 3)*, then rotated 30° around the *y* axis.

The inverse transform can be computed without inverting the matrix as follows:

$$M = R_y(30^\circ) \cdot T(1,2,3)$$

$$M^{-1} = T(1,2,3)^{-1} \cdot R_y(30^\circ)^{-1}$$

$$M^{-1} = \begin{vmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} 0.866 & 0 & -0.5 & 0 \\ 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 0.866 & 0 & -0.5 & -1 \\ 0 & 1 & 0 & -2 \\ 0.5 & 0 & 0.866 & -3 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Composition of transformation: not commutative

Note that matrix product is not commutative.

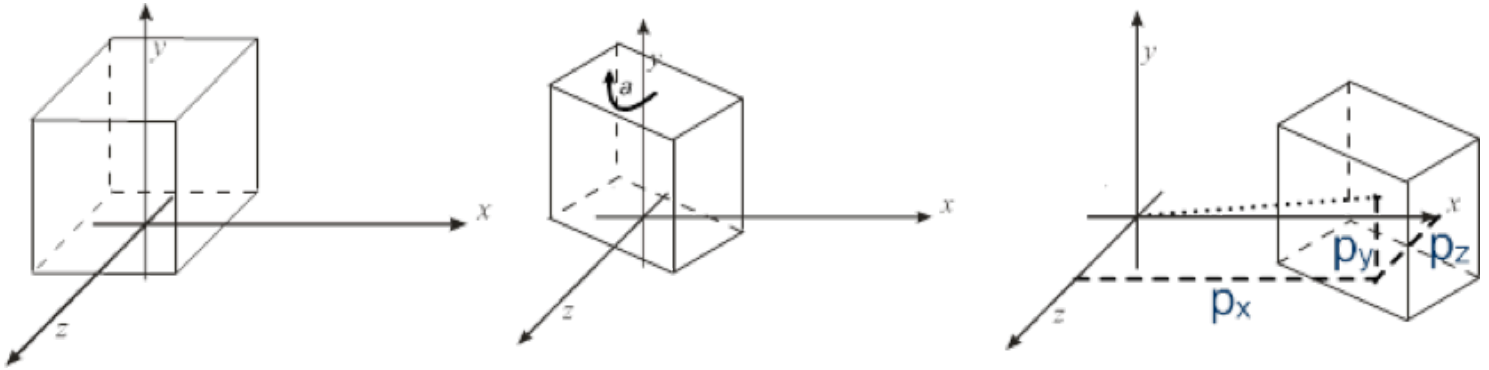
$$A \cdot B \neq B \cdot A$$

This means that the order of transformation is important.

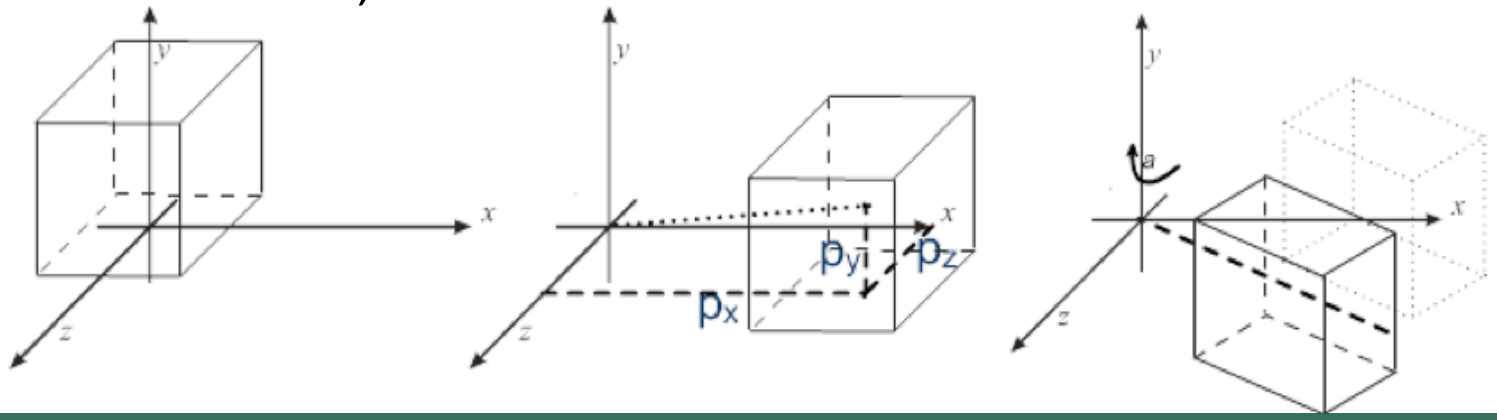
It also means that two transformations cannot be swapped without obtaining a different result.

Composition of transformation: not commutative

For example if we first rotate a cube, and then perform a translation we obtain one result.



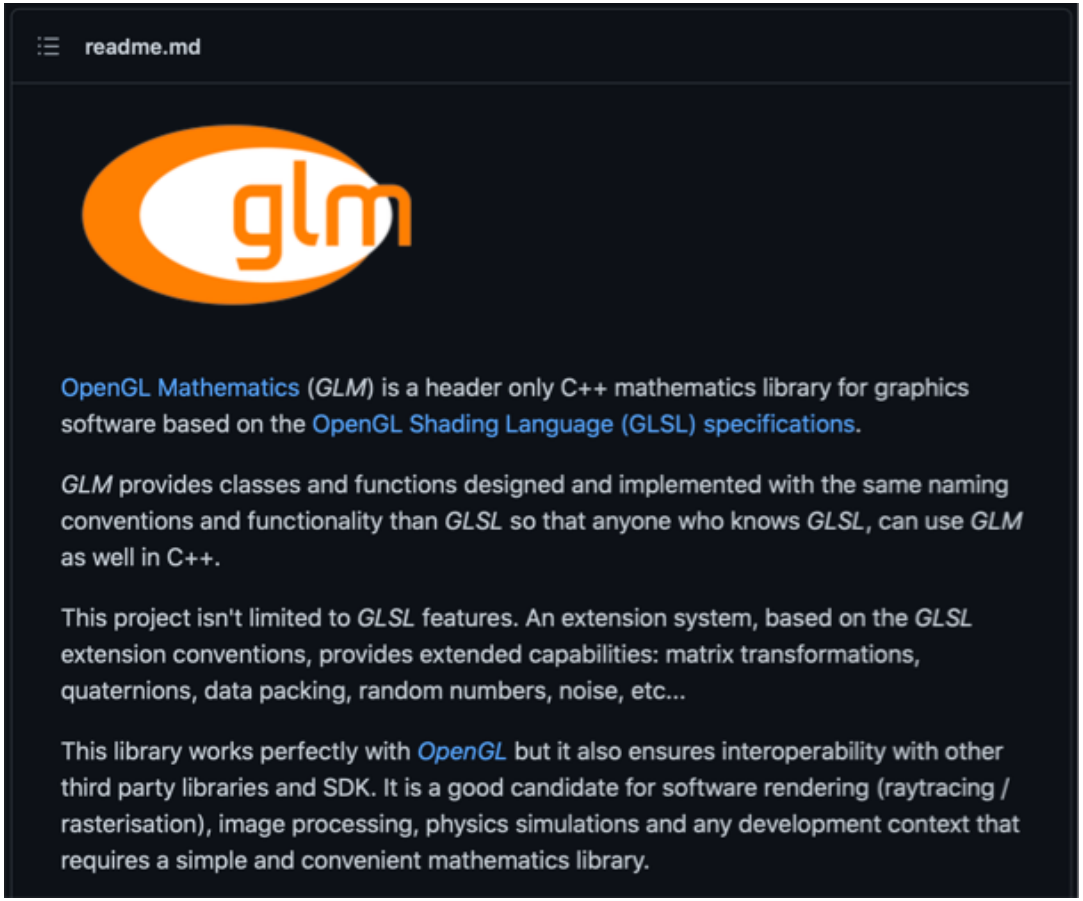
If we maintain the same parameters, but invert the order of the transformations, we obtain a different result.



GLM

GLM is a simple linear algebra library that can be used to simplify matrix operation in Computer Graphics.

It has been created for OpenGL, but it can be used in many other contexts, such as Vulkan.



GLM

It uses the same type names defined in the GLSL shading language.

It is a header-only library, which simplifies compilation and linking.

Exploiting the operator overloading features it can express complex math operations with a very simple syntax.

We will not describe it in-depth right now: we will start with features that can be helpful for this lesson's topics, and we will introduce additional features when needed.

Transform matrix creation with GLM

To use the library, it must be included at the beginning of the .cpp file.

- `glm/glm.hpp` is the main component of the library that must be included first.
- `glm/gtc/matrix_transform.hpp` is the extensions that contains the functions to create transform matrices.

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEFAULT_ALIGNED_GENTYPES
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/gtx/hash.hpp>
```

glm::mat4

Type `glm::mat4` is used to define 4x4 matrices, that can be used to encode transform matrices.

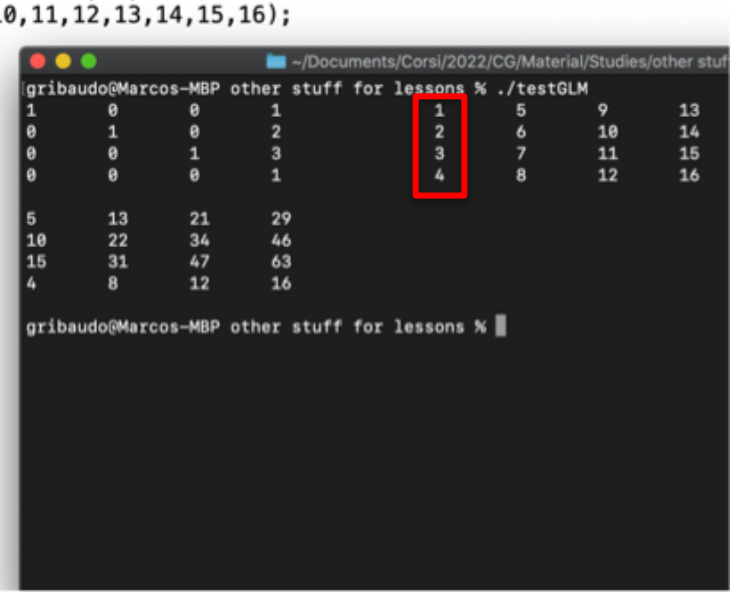
```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
          for(int j = 0; j < 4; j++) {
            std::cout << M2[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```

```
~/Documents/Corsi/2022/CG/Material/Studies/other stuf
gribaudo@Marcos-MBP other stuff for lessons % ./testGLM
1       0       0       1               1       5       9       13
0       1       0       2               2       6       10      14
0       0       1       3               3       7       11      15
0       0       0       1               4       8       12      16

5       13      21      29
10      22      34      46
15      31      47      63
4       8       12      16

gribaudo@Marcos-MBP other stuff for lessons %
```

glm::mat4

Function `glm::mat4(…)` can be used to create a matrix from its element. Elements are specified *per column*.

```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
          for(int j = 0; j < 4; j++) {
            std::cout << M2[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```



glm::mat4

Elements can be accessed with the double [j][i] syntax, specifying first the column and next the row.

```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
          for(int j = 0; j < 4; j++) {
            std::cout << M2[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```

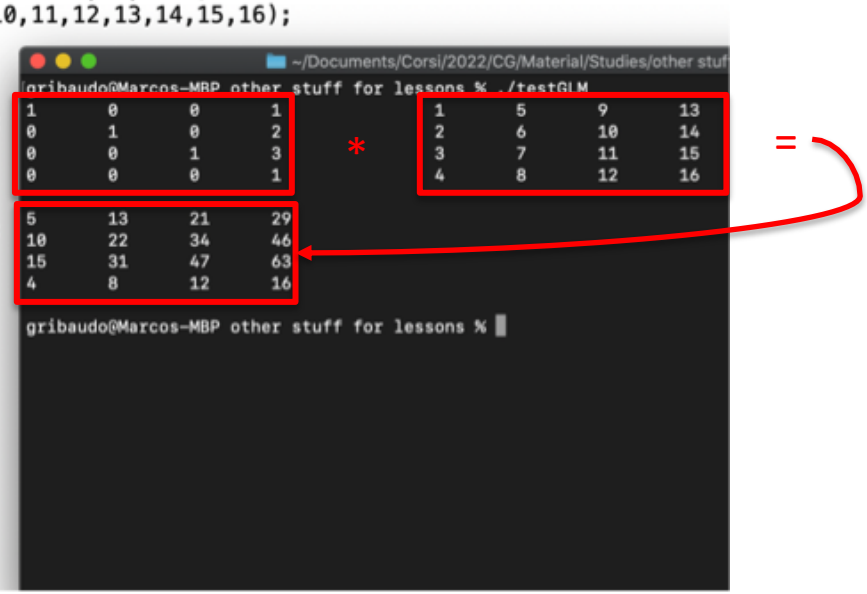
column row

```
~/Documents/Corsi/2022/CG/Material/Studies/other stuf
gribaudo@Marcos-MBP other stuff for lessons % ./testGLM
1       0       0       1               1       5       9       13
0       1       0       2               2       6       10      14
0       0       1       3               3       7       11      15
0       0       0       1               4       8       12      16

5       13      21      29
10      22      34      46
15      31      47      63
4       8       12      16

gribaudo@Marcos-MBP other stuff for lessons %
```

glm::mat4

Matrix product can be computed with the standard * notation.

```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
          for(int j = 0; j < 4; j++) {
            std::cout << M2[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```



glm::mat4

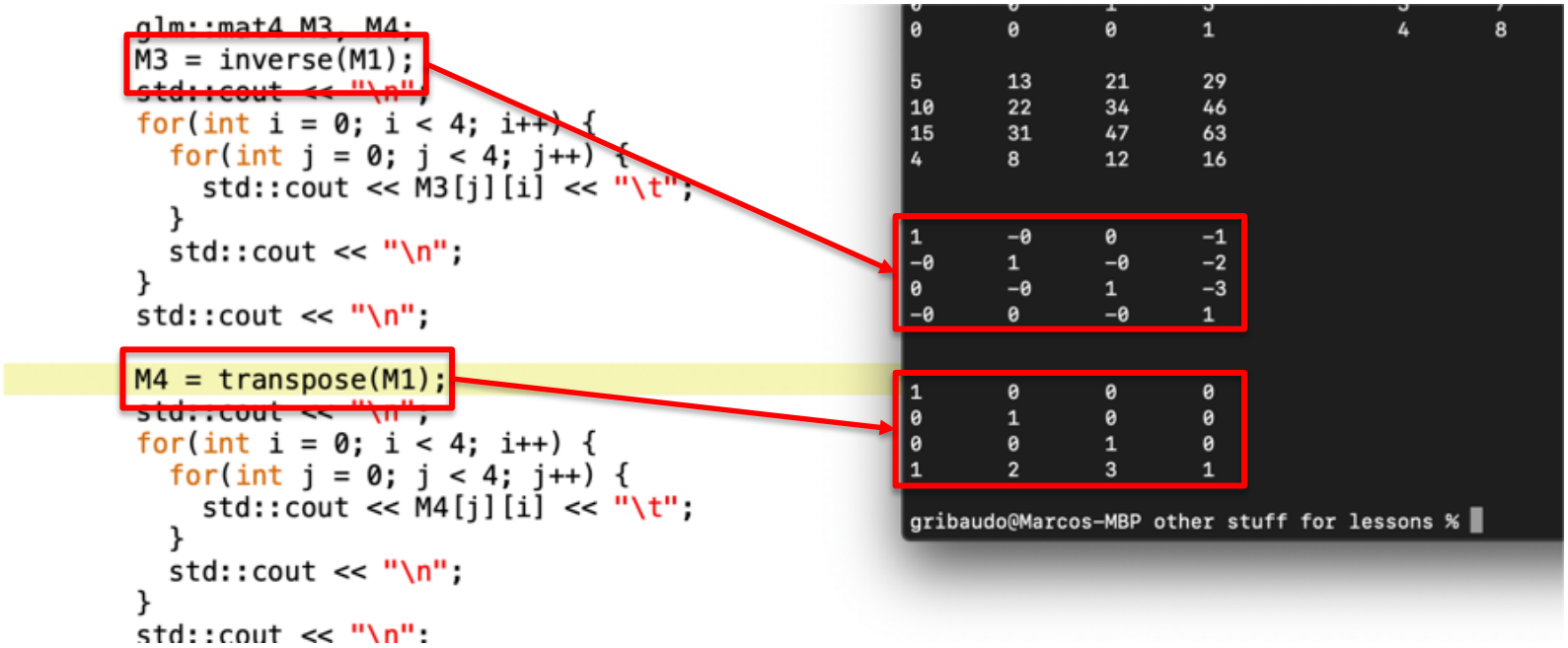
The special syntax `glm::mat4(1)` creates an identity matrix – that is, one on the diagonal and zero elsewhere.

```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
          for(int j = 0; j < 4; j++) {
            std::cout << M2[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```

```
~/Documents/Corsi/2022/CG/Material/Studies/other stuf
gribaudo@Marcos-MBP other stuff for lessons % ./testGLM
1       0       0       1               1       5       9       13
0       1       0       2               2       6       10      14
0       0       1       3               3       7       11      15
0       0       0       1               4       8       12      16

5       13      21      29
10      22      34      46
15      31      47      63
4       8       12      16

gribaudo@Marcos-MBP other stuff for lessons %
```

glm::mat4

Matrix can be inverted with the `inverse(…)` function, and can be transposed with the `transpose(…)` function.



glm::vec3 and glm:vec4

In a similar way, types `glm::vec3` and `glm::vec4` respectively represents three and four components vectors.

```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
          for(int j = 0; j < 4; j++) {
            std::cout << M2[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```

```
~/Documents/Corsi/2022/CG/Material/Studies/other stuf
gribaudo@Marcos-MBP other stuff for lessons % ./testGLM
1       0       0       1               1       5       9       13
0       1       0       2               2       6       10      14
0       0       1       3               3       7       11      15
0       0       0       1               4       8       12      16

5       13      21      29
10      22      34      46
15      31      47      63
4       8       12      16

gribaudo@Marcos-MBP other stuff for lessons %
```

Matrix and vector operations

Standard algebraic operations between matrices and vectors can be computed using the conventional symbols + - and *

```
glm::vec4 V = glm::vec4(5,2,1,1);
glm::vec4 x;

x = M1 * V;
for(int j = 0; j < 4; j++) {
    std::cout << x[j] << "\t";
}
std::cout << "\n\n";

glm::mat4 MT = M1 + M2 - M3;
for(int i = 0; i < 4; i++) {
  for(int j = 0; j < 4; j++) {
    std::cout << MT[j][i] << "\t";
  }
  std::cout << "\n";
}
std::cout << "\n";
```

```
6       4       4       1

1       5       9       15
2       6       10      18
3       7       11      21
4       8       12      16

gribaudo@Marcos-MBP other stuff for lessons %
```

Transform matrix creation with GLM

Transform matrices can be created with the following functions:

```
glm::mat4 T = glm::translate(glm::mat4(1), glm::vec3(dx, dy, dz));
```

Puts in variable T a 4x4 translation matrix of displacement d_x, d_y and d_z.

```
class testGLM {
    public:
    void run() {
        glm::mat4 M1, M2, M;
        M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
        M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
        M = M1 * M2;
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << M1[j][i] << "\t";
          }
          std::cout << "\t";
```

```
~/Documents/Corsi/2022/CG/Material/Studies/other stuf
gribaudo@Marcos-MBP other stuff for lessons % ./testGLM
1       0       0       1               1       5       9       13
0       1       0       2               2       6       10      14
0       0       1       3               3       7       11      15
0       0       0       1               4       8       12      16
```

Transform matrix creation with GLM

```
glm::mat4 S = glm::scale(glm::mat4(1), glm::vec3(sx, sy, sz));
```

Puts in variable S a 4x4 scaling matrix with scaling factors s_x, s_y and s_z.

A shortcut for proportional scaling of factor s is the following:

- `glm::mat4 Sp = glm::scale(glm::mat4(1), glm::vec3(s));`

```
glm::mat4 R = glm::rotate(glm::mat4(1), ang, glm::vec3(ax, ay, az));
```

Puts in variable R a 4x4 rotation matrix of an angle *ang* along an axis oriented according to vector a_x, a_y and a_z.

Rotations along the x, y and z axis, can be respectively performed in this way:

- `glm::mat4 Rx = glm::rotate(glm::mat4(1), ang, glm::vec3(1, 0, 0));`
- `glm::mat4 Ry = glm::rotate(glm::mat4(1), ang, glm::vec3(0, 1, 0));`
- `glm::mat4 Rz = glm::rotate(glm::mat4(1), ang, glm::vec3(0, 0, 1));`

Transform matrix creation with GLM

It is convenient to specify angles in radians. This is obtained by adding the `#define GLM_FORCE_RADIANS` specification before including the library:

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEFAULT_ALIGNED_GENTYPES
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/gtx/hash.hpp>
```

In this case, angles can be converted from degrees to radians using the `glm::radians()` function:

```
const float ROT_SPEED = glm::radians(60.0f);
const float MOVE_SPEED = 0.75f;
const float MOUSE_RES = 500.0f;
```

Please note: due to the strange way in which floating point type works in C++, if your angle is integer, you have to add `.0f` at the end to explicitly make it floating point. If it is fractional value, you have to add and `f` at the end. For example:

60° -> `60.0f`

22.5° -> `22.5f`

Transform matrix creation with GLM

Shear requires the inclusion of a special header, and can be performed with the following procedures:

- `glm::mat4 Rx = glm::shearX3D(glm::mat4(1), hy, hz);`
- `glm::mat4 Rx = glm::shearY3D(glm::mat4(1), hx, hz);`
- `glm::mat4 Rx = glm::shearZ3D(glm::mat4(1), hx, hy);`

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

#define GLM_FORCE_RADIANS

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform2.hpp>

....

        MSh = shearY3D(glm::mat4(1), 0.5f, -0.25f);
        for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; j++) {
            std::cout << MSh[j][i] << "\t";
          }
          std::cout << "\n";
        }
        std::cout << "\n";
    }
};
```

```
1       -0      0       -1
-0      1       -0      -2
0       -0      1       -3
-0      0       -0      1


1       0       0       0
0       1       0       0
0       0       1       0
1       2       3       1

6       4       4       1

1       5       9       15
2       6       10      18
3       7       11      21
4       8       12      16

1       0.5     0       0
0       1       0       0
0       -0.25   1       0
0       0       0       1

gribaudo@Marcos-MacBook-Pro E01 %
```

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)