



**POLITECNICO
MILANO 1863**

**DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA**



2024

Dipartimento di Elettronica, Informazione e Bioingegneria
Computer Graphics

Milano, 2024

Computer Graphics

- Advanced Transformations and introduction to projections



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 a point p , then point p can be obtained back from point p' by multiplying it with M^{-1} , the inverse of the matrix M .

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

$$p' = M \times p$$

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

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

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

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

Inversion of transformations

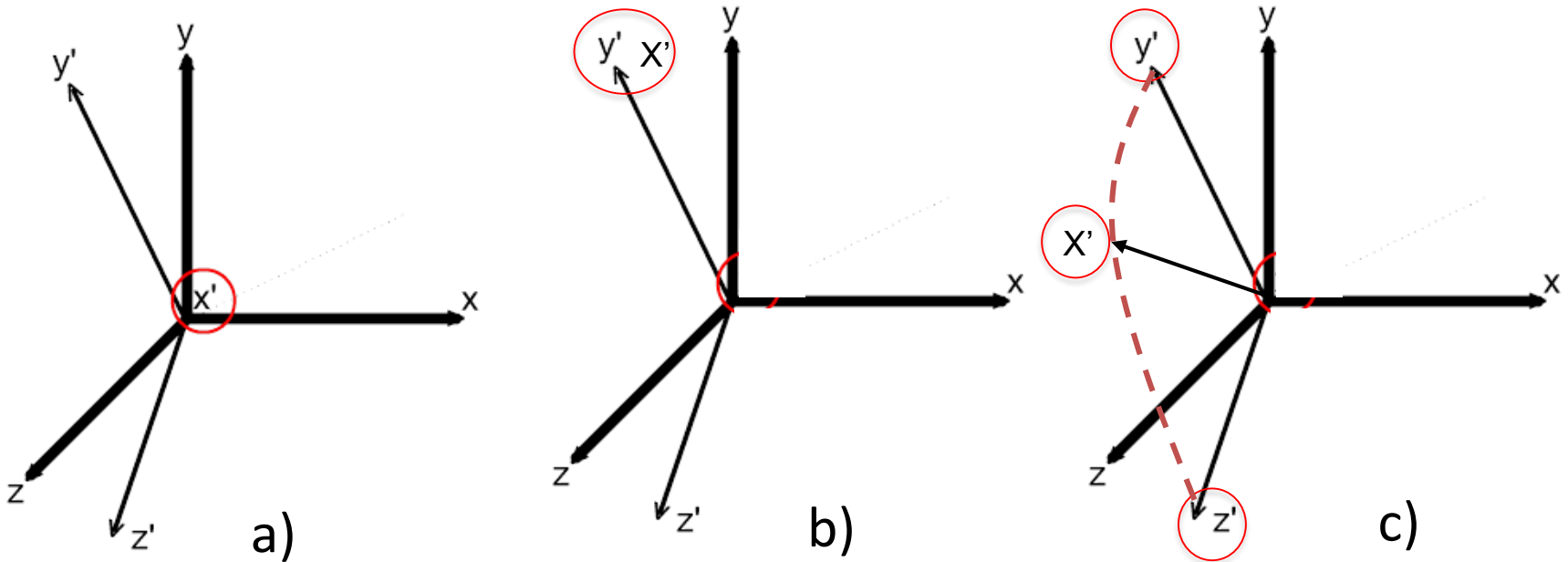
Since the last row is $|0\ 0\ 0\ 1|$, it can be proven that a transform matrix M is invertible if its sub-matrix composed by the first 3 rows and 3 columns is invertible.

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

Inversion of transformations

This is generally the case, except when:

- a) one or two of the projected axis degenerates to be of zero length,
- b) when two axis perfectly overlaps, or
- c) 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.

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

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

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

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

$$A = R_C$$

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

$$\det(A) = ad - bc$$

$$\det(A) = a_{11}(a_{22}a_{33} - a_{32}a_{23}) - a_{21}(a_{12}a_{33} - a_{32}a_{13}) + a_{31}(a_{12}a_{23} - a_{22}a_{13})$$

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

However, if we want to the invert some of the transformations previously presented, we have simple matrix patterns that we can use to compute their inverse 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 $s \neq 0$.

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

Inversion of transformations

For rotation:

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

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

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

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

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

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

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:

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

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

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 a **composition**.

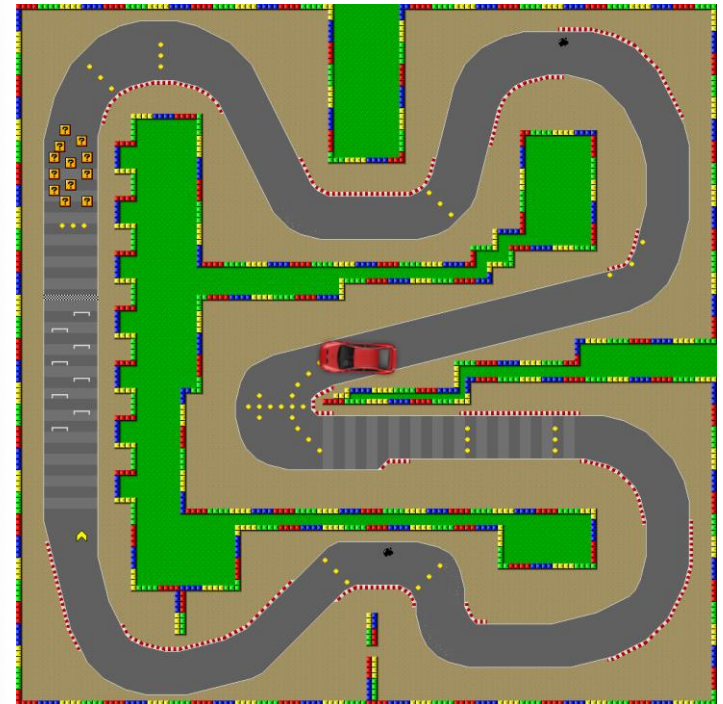
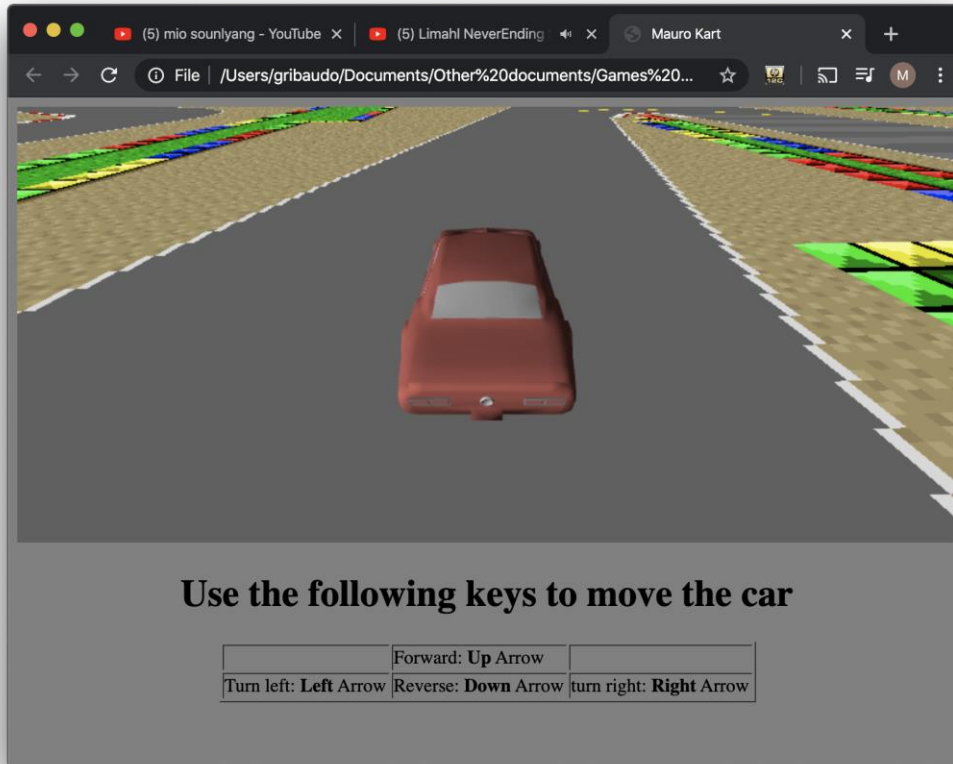
Usually, an object should be translated and rotated in all directions to be positioned in the scene.

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 done in a very efficient way.

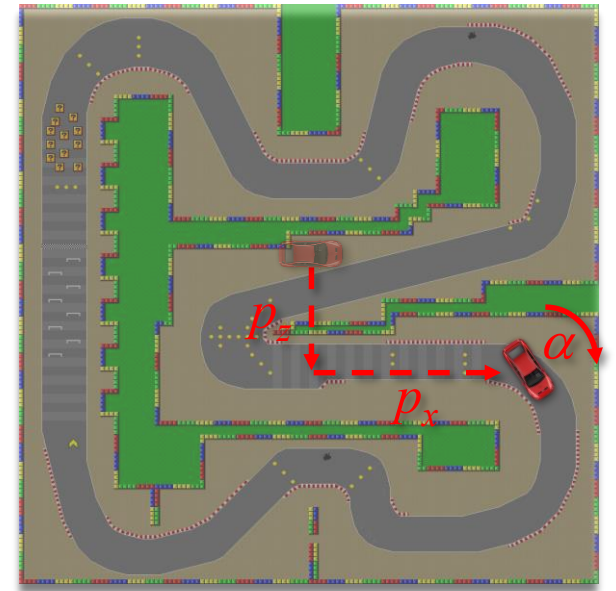
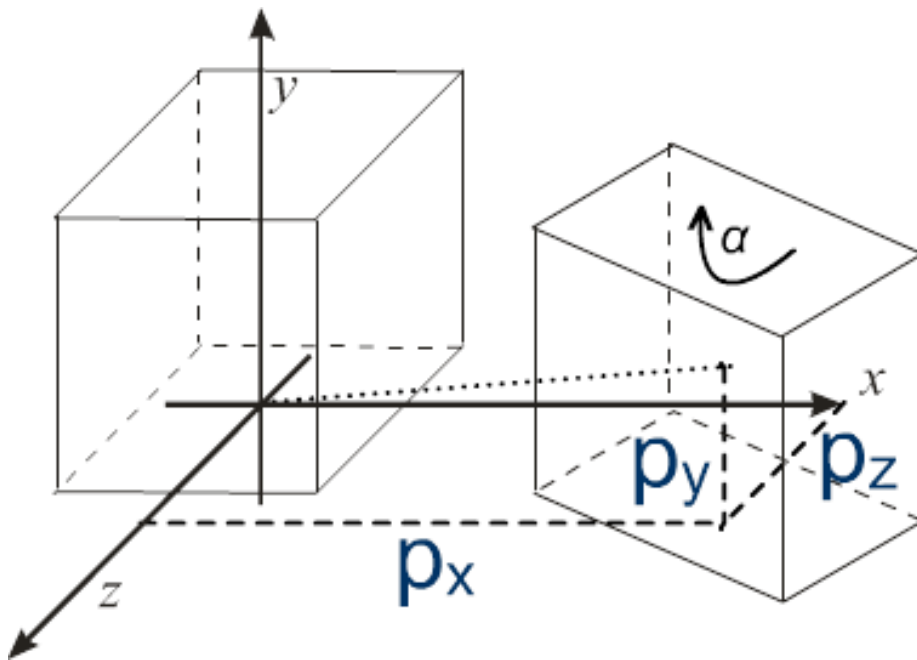
Composition of transformations

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



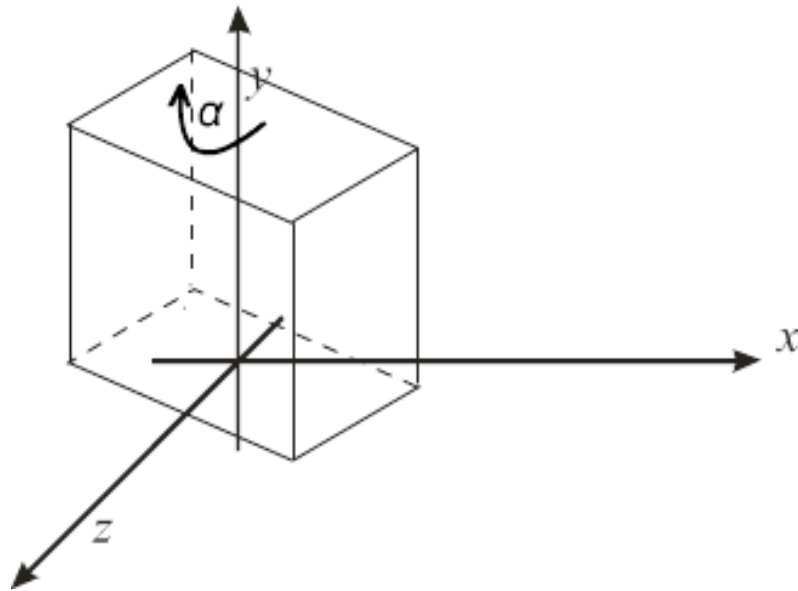
Composition of transformations

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



Composition of transformations

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

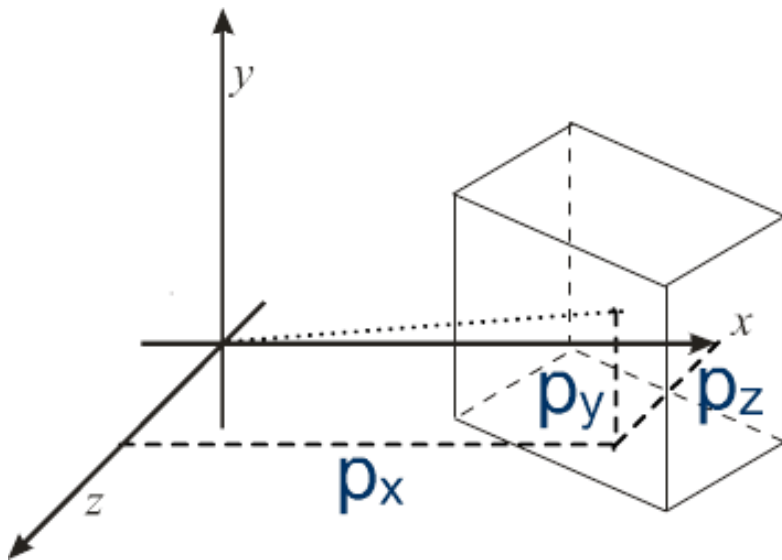


$$R_y(a) = \begin{vmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$p' = R_y(a) \times p$$

Composition of transformations

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



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

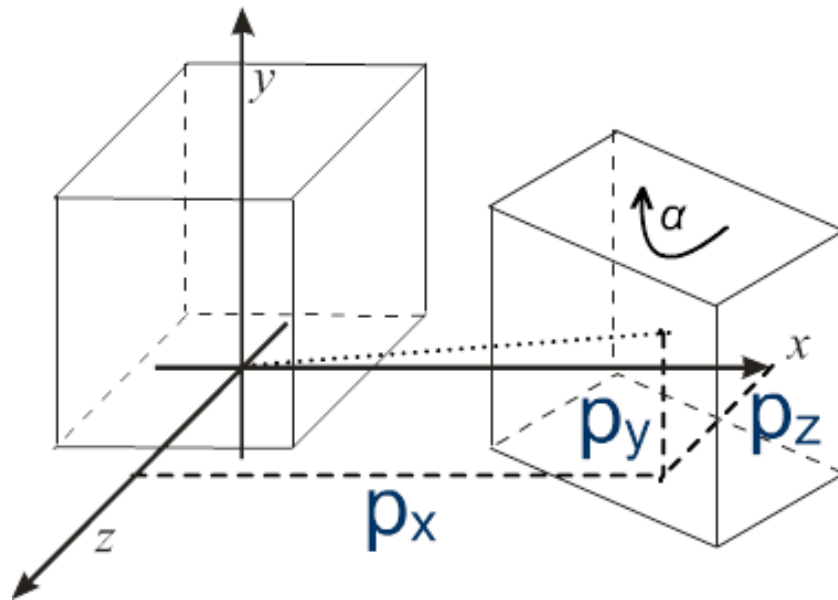
$$p' = R_y(a) \times p$$

$$p'' = T(p_x, p_y, p_z) \times p' = T(p_x, p_y, p_z) \times R_y(a) \times p$$

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

Composition of transformations

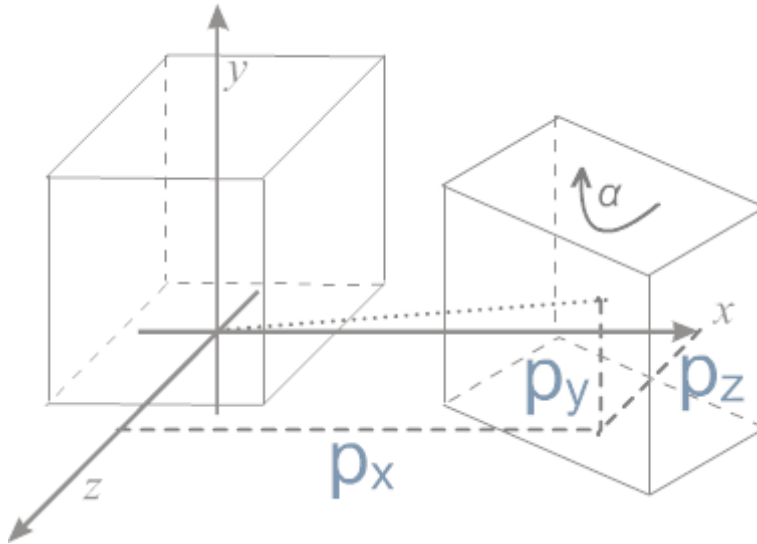
Note that, as in functional programming, matrices appear inside the expression in reverse order with respect to the transformations they represent.



$$p'' = T(p_x, p_y, p_z) \times R_y(a) \times p$$

Composition of transformations

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

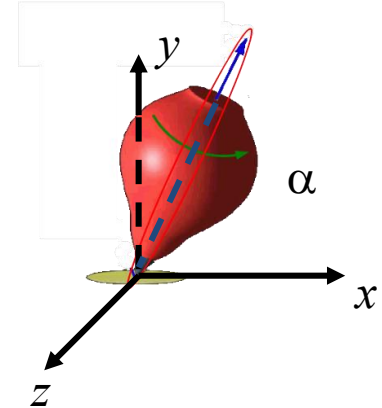
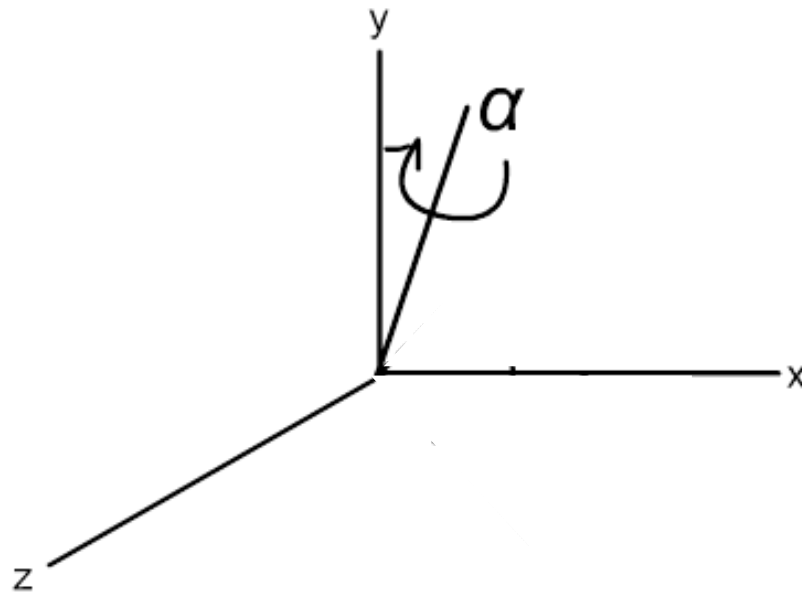


In this course, we will never use the *matrix-on-the-right* convention.

$$p'' = p \times R_y(a)^T \times T(p_x, p_y, p_z)^T$$

Transformations around an arbitrary axis or center

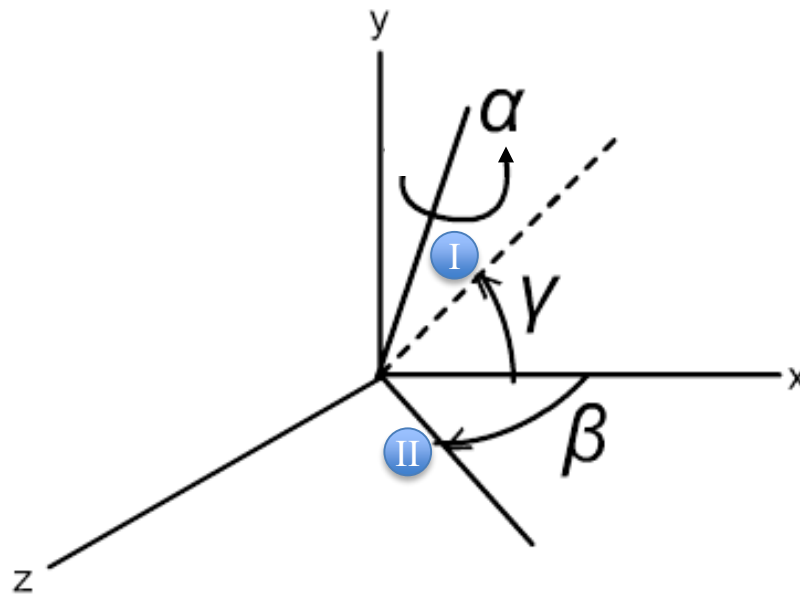
Now consider the rotation of an object of an angle α about an arbitrary axis that passes through the origin.



Transformations around an arbitrary axis or center

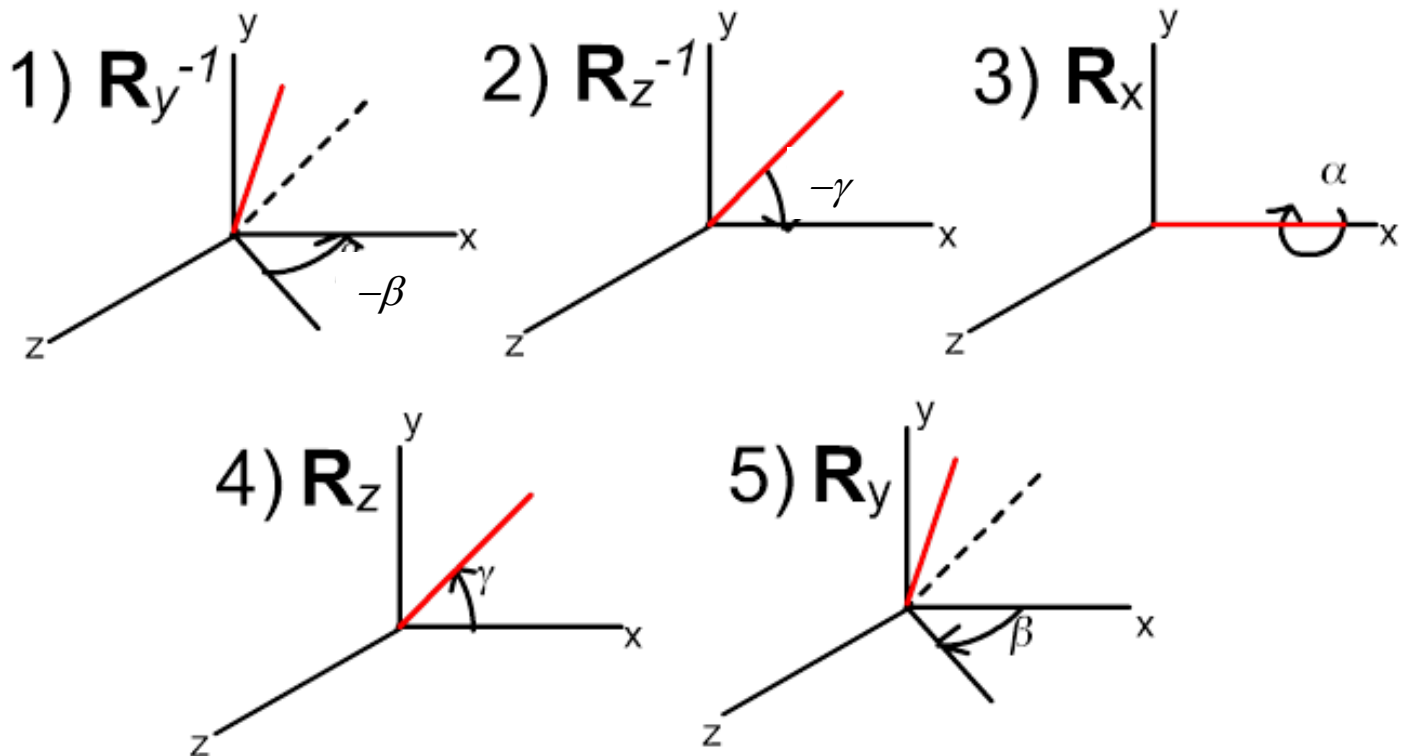
Let us focus on a case where the direction of the arbitrary axis can be expressed with a couple of angles.

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

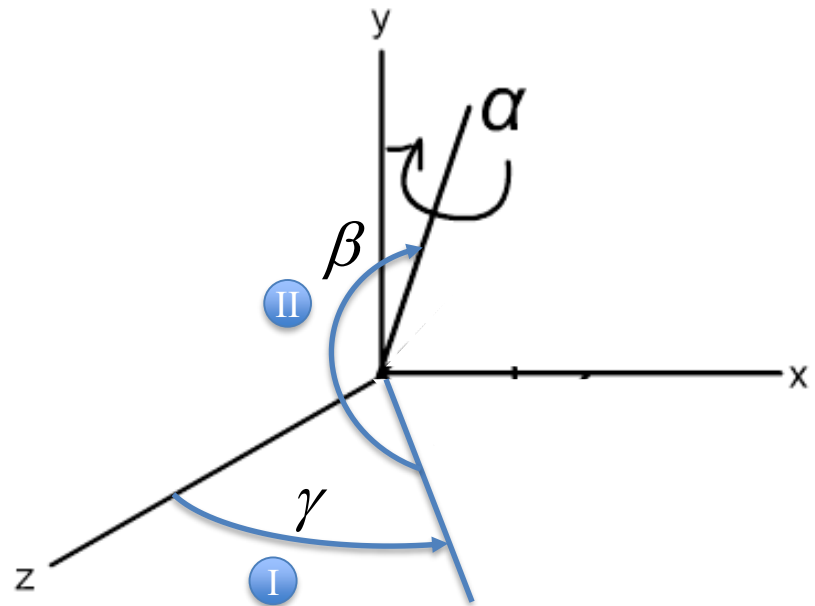
The arbitrary axis rotation can be obtained by combining 5 elementary transformations:



Transformations around an arbitrary axis or center

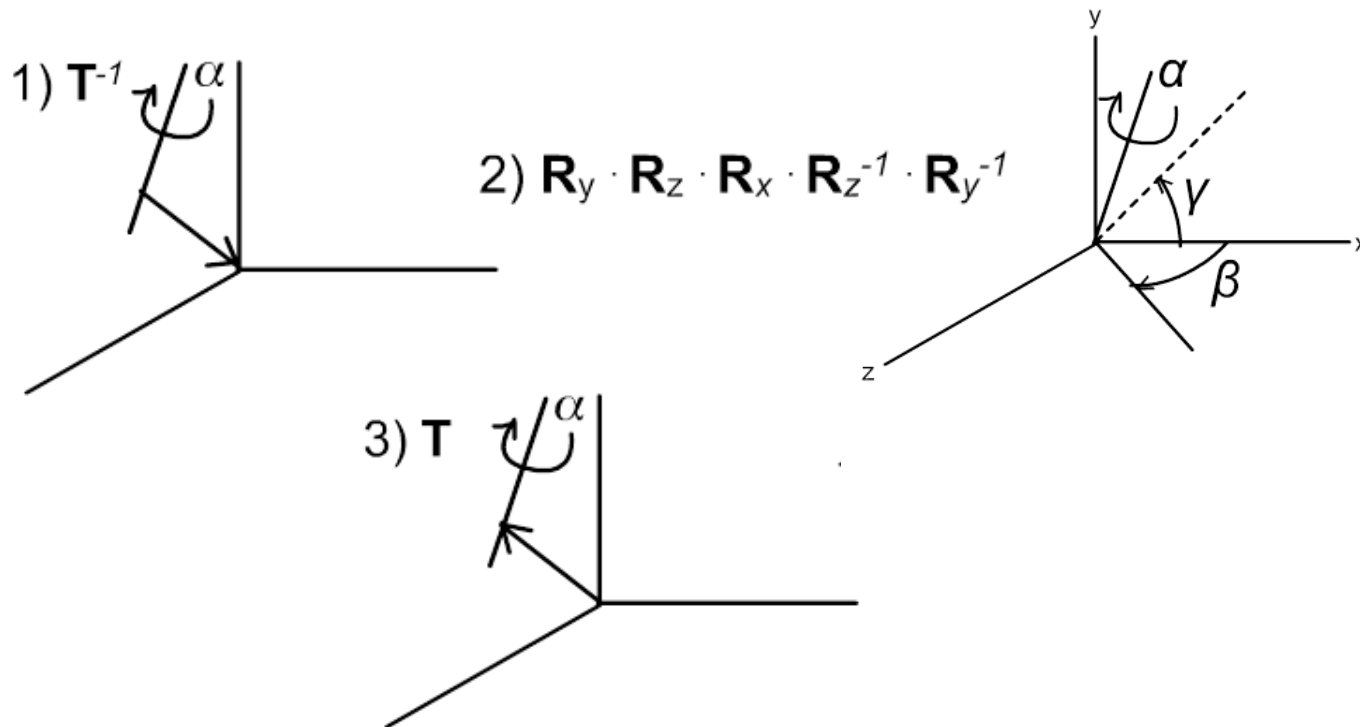
Please note that the same reasoning can be extended to many different ways in which an axis can be aligned to the arbitrary one, with a set of elementary rotations.

For example, here we see a sequence of rotations, of γ around the y -axis, then β around the x -axis, that align the z -axis with the arbitrary one.



Transformations around an arbitrary axis or center

If an axis does not pass through the origin, but through a point (p_x, p_y, p_z) , a translation $T^{-1}(p_x, p_y, p_z)$ must be applied to move it to $(0,0,0)$, and then use $T(p_x, p_y, p_z)$ at the end to return the axis to its initial position.



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:

$$p' = T(p_x, p_y, p_z) \times R_y(b) \times R_z(g) \times R_x(a) \times R_z(g)^{-1} \times R_y(b)^{-1} \times T(p_x, p_y, p_z)^{-1} \times p$$

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 very application dependent, and its derivation 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:

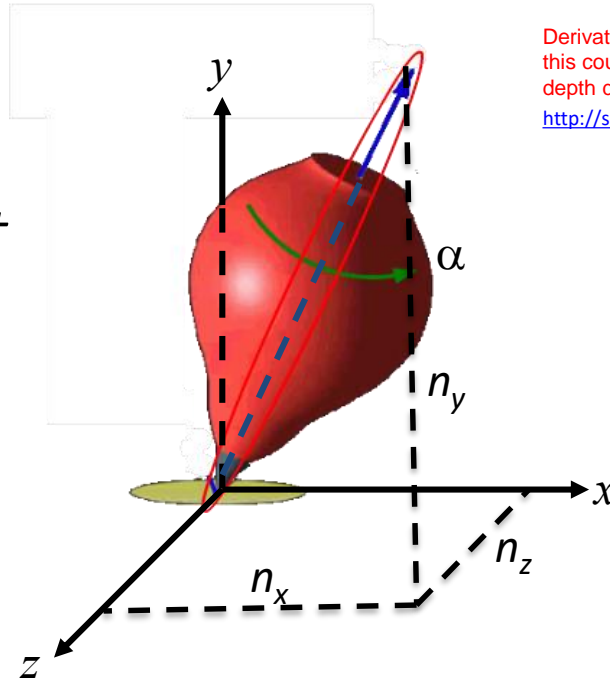
$$p' = T(p_x, p_y, p_z) \times S(s_x, s_y, s_z) \times T(p_x, p_y, p_z)^{-1} \times p$$

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

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

Axis-angle rotation matrix

In many circumstances, the rotation axis can be expressed with a unitary vector $\mathbf{n} = (n_x, n_y, n_z)$ 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

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

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.

$$p' = T(p_x, p_y, p_z) \times R_y(b) \times R_z(g) \times R_x(a) \times R_z(g)^{-1} \times R_y(b)^{-1} \times T(p_x, p_y, p_z)^{-1} \times p$$

$$R_{p_x, p_y, p_z, b, g}(a) = T(p_x, p_y, p_z) \times R_y(b) \times R_z(g) \times R_x(a) \times R_z(g)^{-1} \times R_y(b)^{-1} \times T(p_x, p_y, p_z)^{-1}$$

$$p' = R_{p_x, p_y, p_z, b, g}(a) \times p$$

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:

$$\begin{aligned} \left[R_{p_x, p_y, p_z, b, g}(a) \right]^{-1} &= \left[T(p_x, p_y, p_z) \cdot R_y(b) \cdot R_z(g) \cdot R_x(a) \cdot R_z(g)^{-1} \cdot R_y(b)^{-1} \cdot T(p_x, p_y, p_z)^{-1} \right]^{-1} = \\ &= \left[T(p_x, p_y, p_z)^{-1} \right]^{-1} \cdot \left[R_y(b)^{-1} \right]^{-1} \cdot \left[R_z(g)^{-1} \right]^{-1} \cdot R_x(a)^{-1} \cdot R_z(g)^{-1} \cdot R_y(b)^{-1} \cdot T(p_x, p_y, p_z)^{-1} = \\ &= T(p_x, p_y, p_z) \cdot R_y(b) \cdot R_z(g) \cdot R_x(a)^{-1} \cdot R_z(g)^{-1} \cdot R_y(b)^{-1} \cdot T(p_x, p_y, p_z)^{-1} = R_{p_x, p_y, p_z, b, g}(-a) \end{aligned}$$

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:

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

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

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

Composition of transformation: not commutative

Note that matrix product is not commutative.

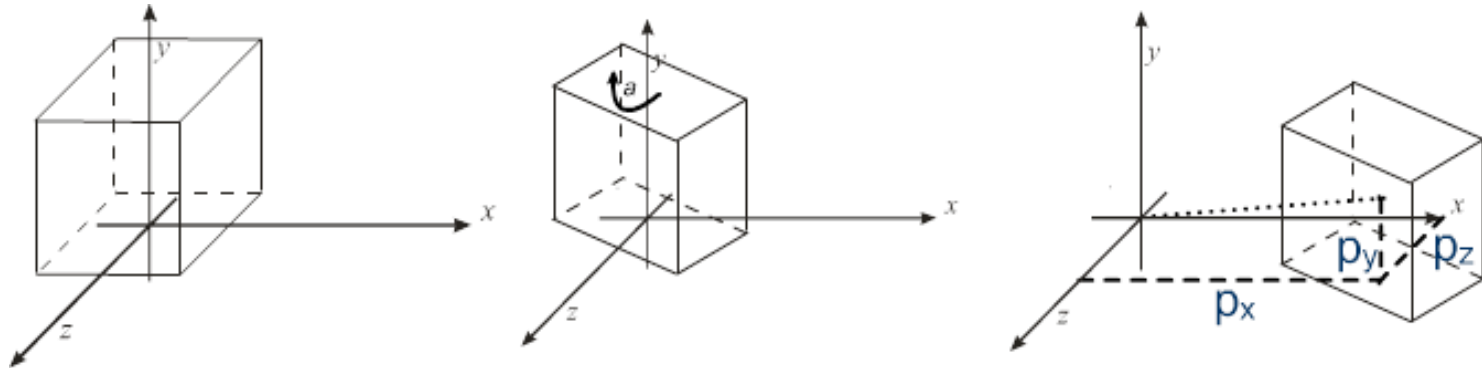
$$A \times B \neq B \times A$$

This means that the order of transformation matters.

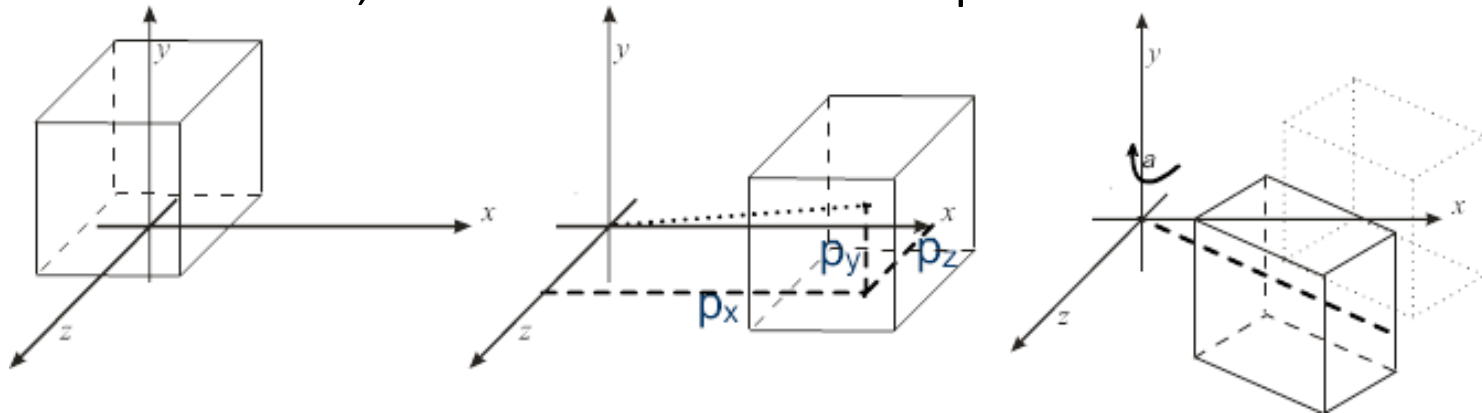
In other words, if we swap two transformations we obtain a different result.

Composition of transformation: not commutative

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

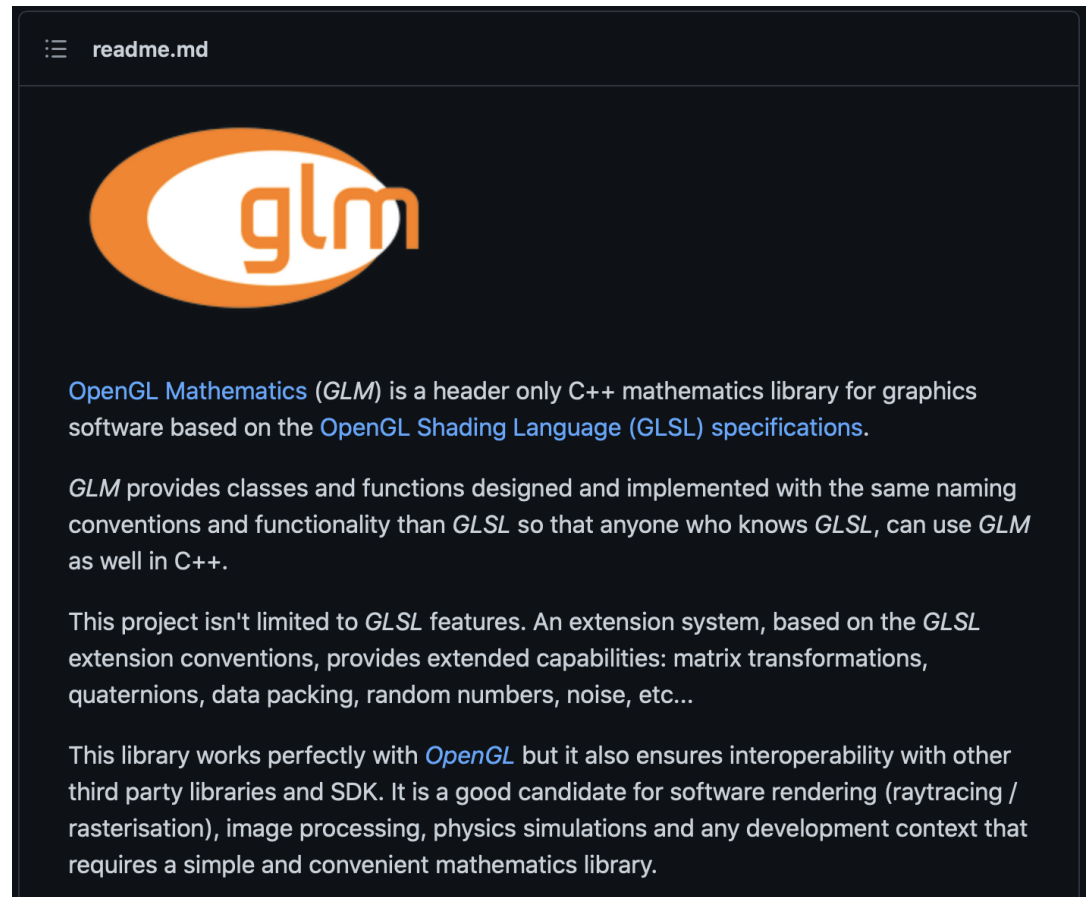


If we maintain the same parameters, but invert the order of the transformations, we obtain a different final position.



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.



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

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

By exploiting the C++ operator overloading features, it can express complex mathematical 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

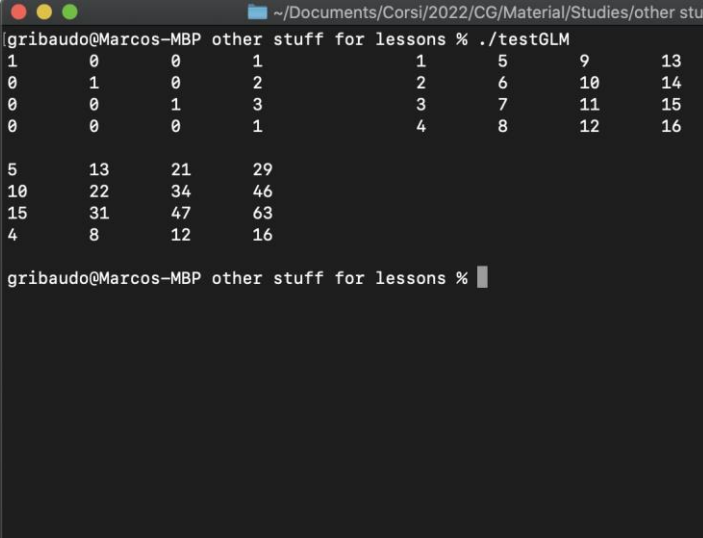
To use the library, it must be included at the beginning of the .cpp file.

- `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>
```

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 = glm::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 << "\n";
            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";
    }
};
```



```
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 %
```

Function `glm::mat4(...)` can be used to create a matrix from its elements, which must be passed *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";
    }
};
```

```
gribaudo@Marcos-MBP other stuff for lessons % ./testGLM
1 0 0 1 5 6 7 8 13 14 15 16 21 22 23 24 29 30 31 32
0 1 0 2 13 14 15 16 22 23 24 25 31 32 33 34 40 41 42 43
0 0 1 3 21 22 23 24 31 32 33 34 40 41 42 43 49 50 51 52
0 0 0 1 29 30 31 32 40 41 42 43 49 50 51 52 59 60 61 62

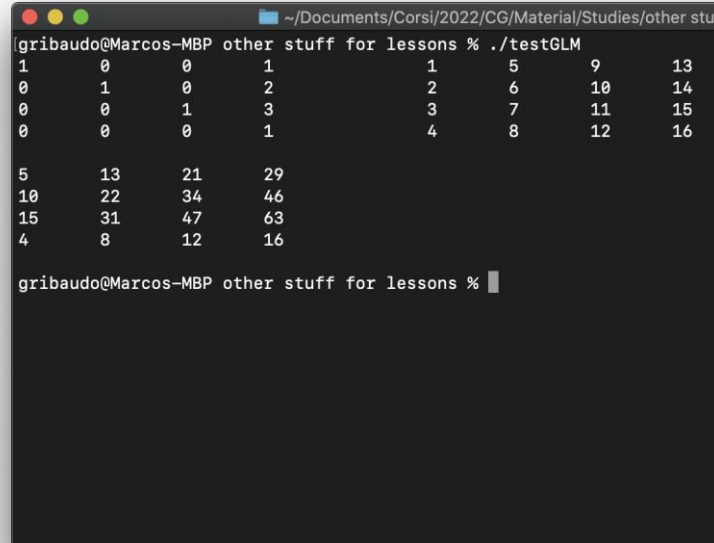
gribaudo@Marcos-MBP other stuff for lessons %
```

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 << "\n";
            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



```
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 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";
    }
};
```

The terminal output shows the result of the matrix multiplication. The first two 4x4 matrices are highlighted with red boxes, followed by a red asterisk (*). The resulting 4x4 matrix is also highlighted with a red box. A red arrow points from the result box to the right, where a red equals sign (=) is located.

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

*

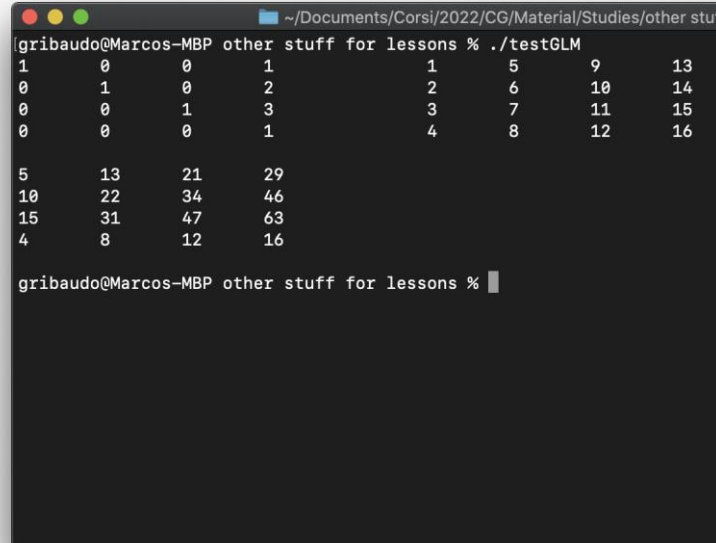
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

=

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

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

```
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 << "\n";
            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";
    }
};
```



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

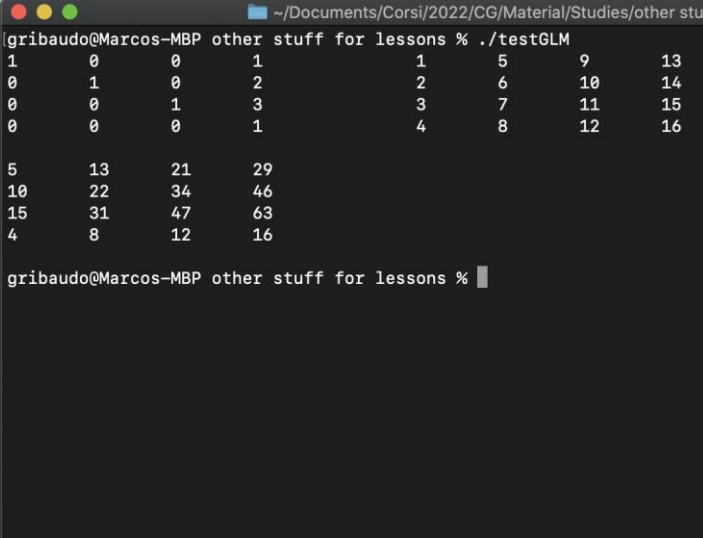
```
glm::mat4 M3, M4;  
M3 = inverse(M1);  
std::cout << "\n";  
for(int i = 0; i < 4; i++) {  
    for(int j = 0; j < 4; j++) {  
        std::cout << M3[j][i] << "\t";  
    }  
    std::cout << "\n";  
}  
std::cout << "\n";  
M4 = transpose(M1);  
std::cout << "\n";  
for(int i = 0; i < 4; i++) {  
    for(int j = 0; j < 4; j++) {  
        std::cout << M4[j][i] << "\t";  
    }  
    std::cout << "\n";  
}  
std::cout << "\n";
```

```
0 0 1 0 3 7  
0 0 0 1 4 8  
  
5 13 21 29  
10 22 34 46  
15 31 47 63  
4 8 12 16  
  
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  
  
gribaudo@Marcos-MBP other stuff for lessons %
```

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 << "\n";
            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";
    }
};
```



```
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 %
```

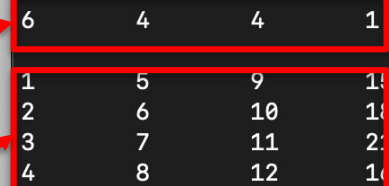
Matrix and vector operations

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 %

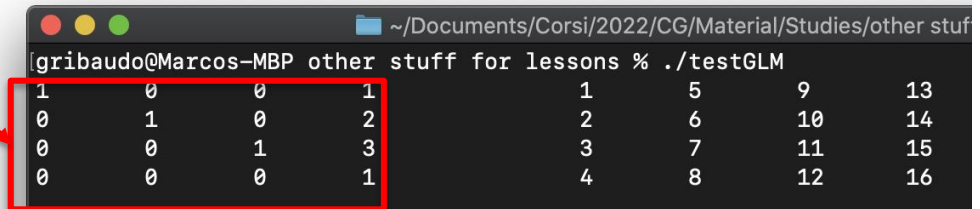
Transform matrix creation with GLM

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



```
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, `<glm/gtx/transform2.hpp>`, 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 %

Projections

In 3D computer graphics, the goal is to represent a three-dimensional space on a screen.

The screen has only two dimensions: even when considering stereoscopic images, the sensation of the third dimension is given by the way in which the human brain interprets two separate 2D images sent to the left and to the right eye.

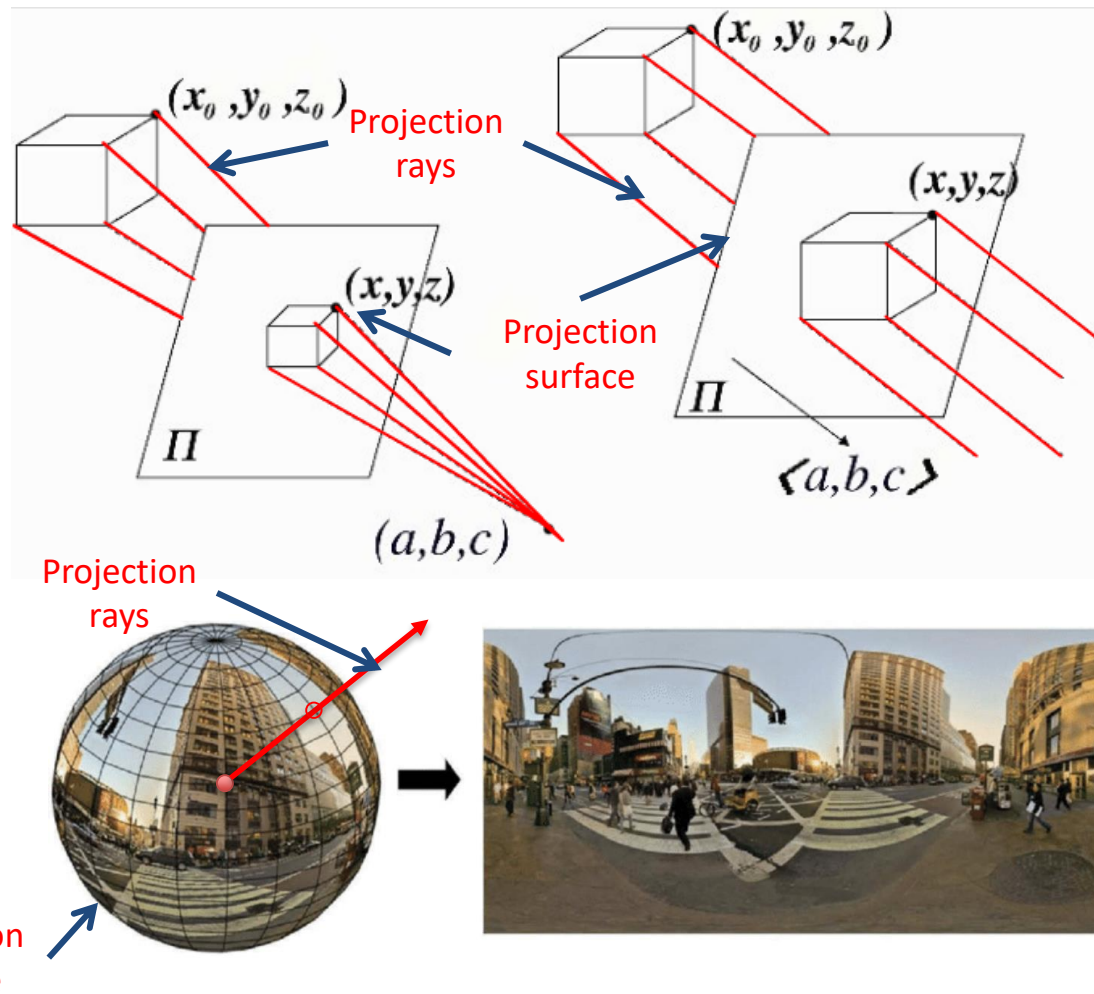
3D computer graphics uses geometrical primitives to describe objects in a three dimensional space.

Then it produces a 2D representation of this space, and shows it on the screen.

Projections

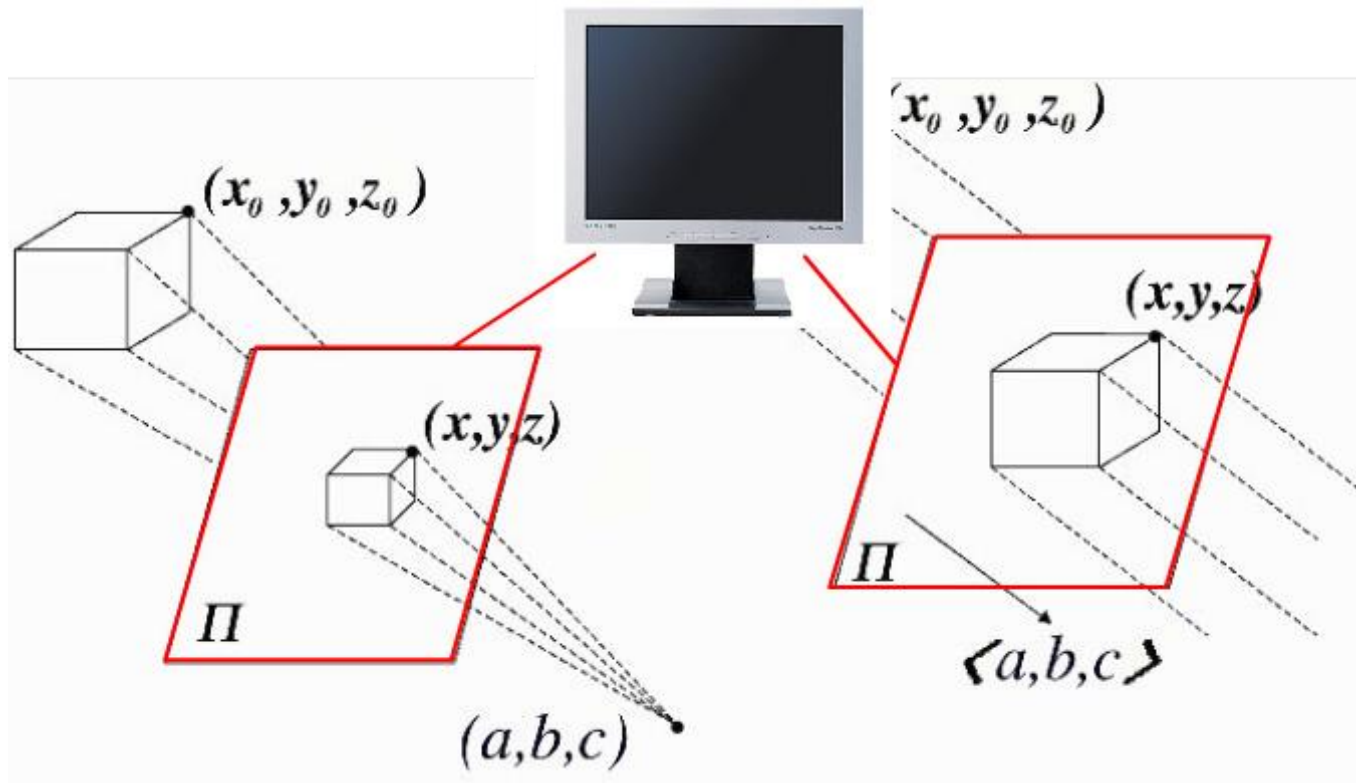
The technique used to construct a 2D image from a 3D scene is called *projection*.

The 2D representation of the 3D object is defined by the intersection of a set of *projection rays* with a surface.



Projections

This surface is generally a plane. It is called the *projection plane*, and a rectangular area defined on it corresponds to the screen. We will only consider planar projections.



Projections

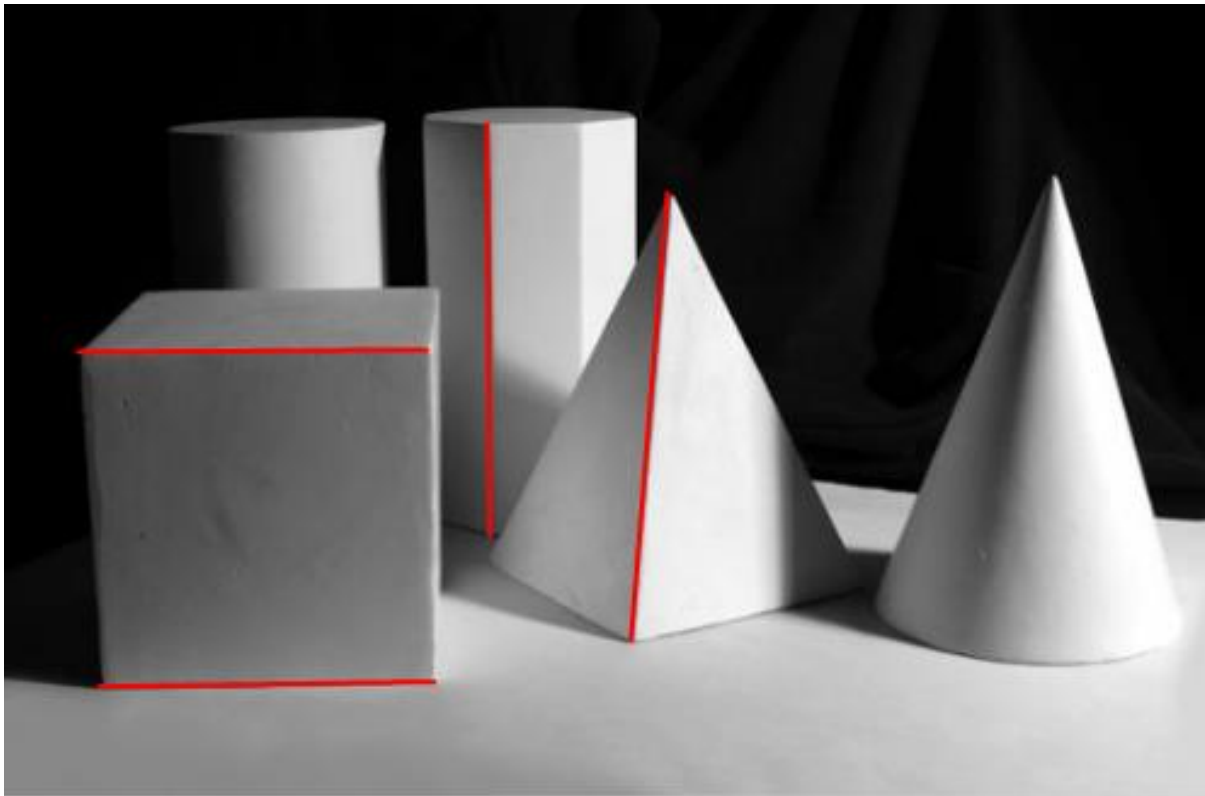
The idea of correctly represent a 3D object over a 2D surface with projections was studied in the XVI century.



DÜRER, Albrecht (1471-1528). Institutiones geometricae - 1522

Projections

One important properties of planar projections, is that linear segments in the 3D scene remain straight on the screen.



Projections

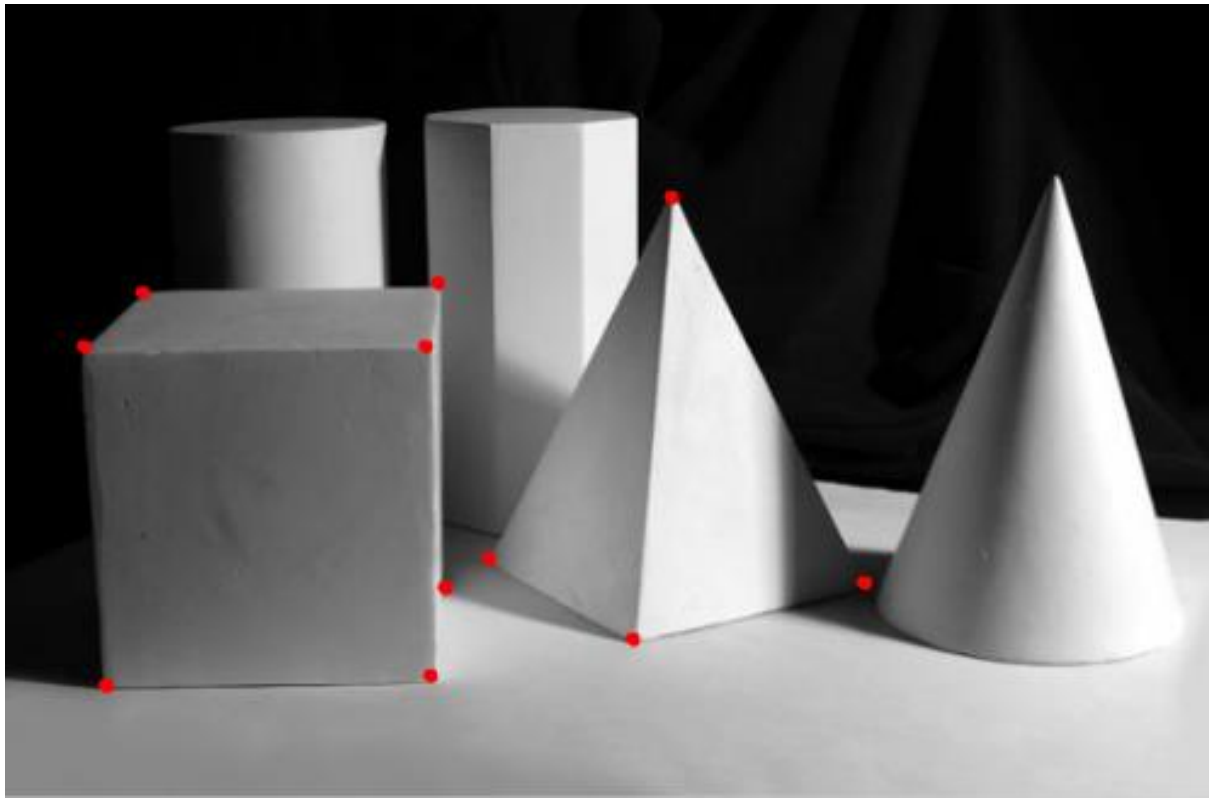
Note that this is no longer the case with projections on different surfaces

Straight lines
become curves



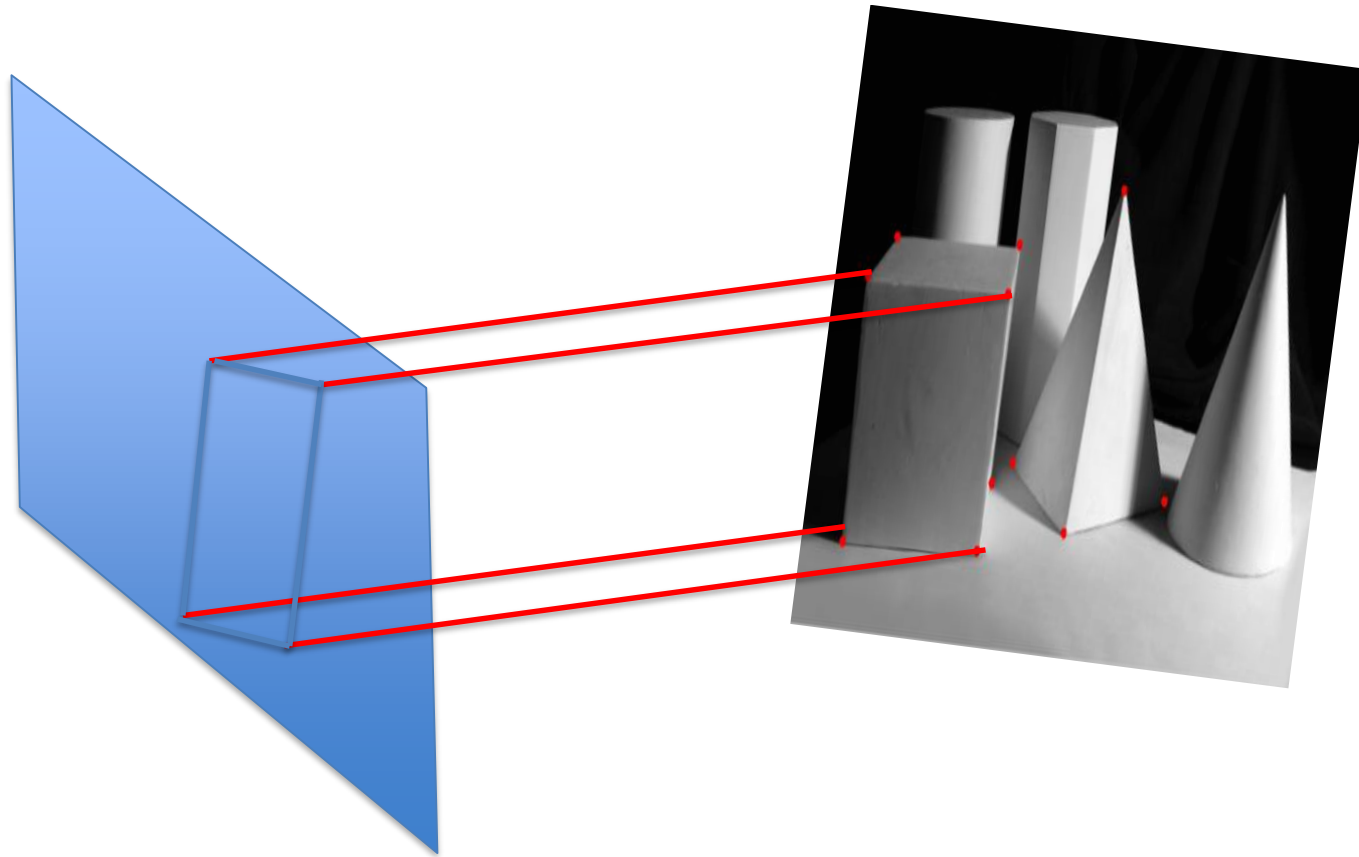
Projections

Since the projected segments connect the projections of their end points...



Projections

It is sufficient to project the vertices, and connect them to recreate a 2D representation of the corresponding 3D Objects.



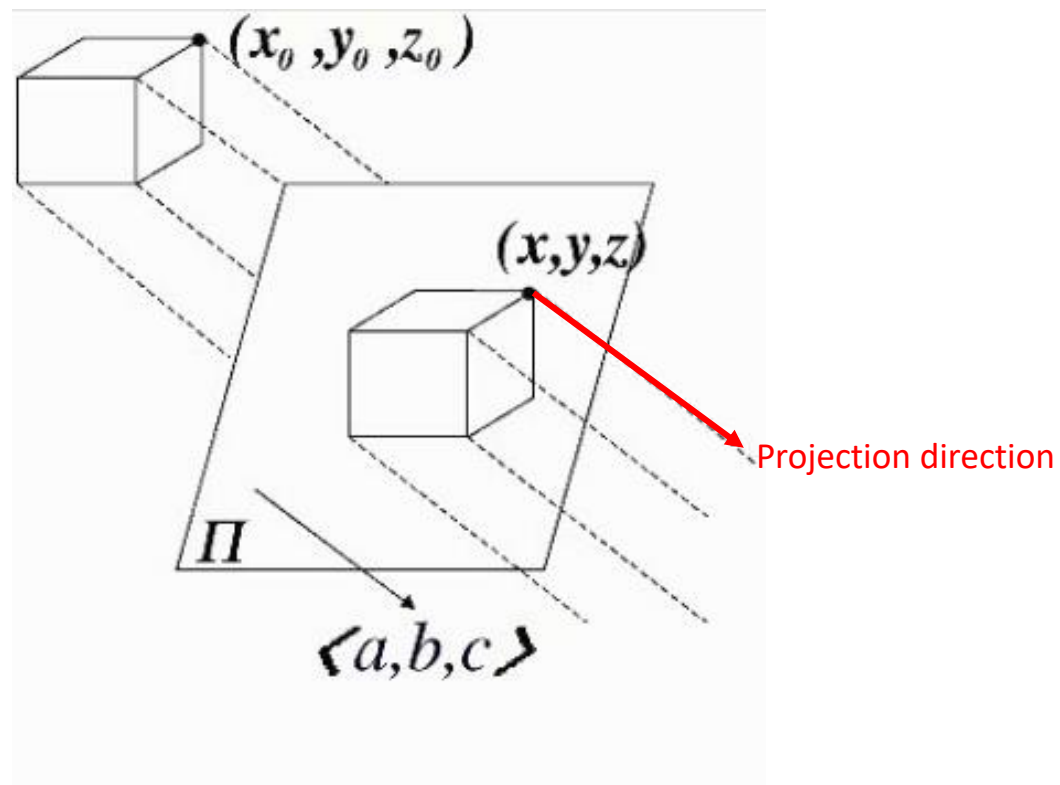
Projection types

We will consider two types of planar projections:

- *parallel projections*
- *perspective projections*

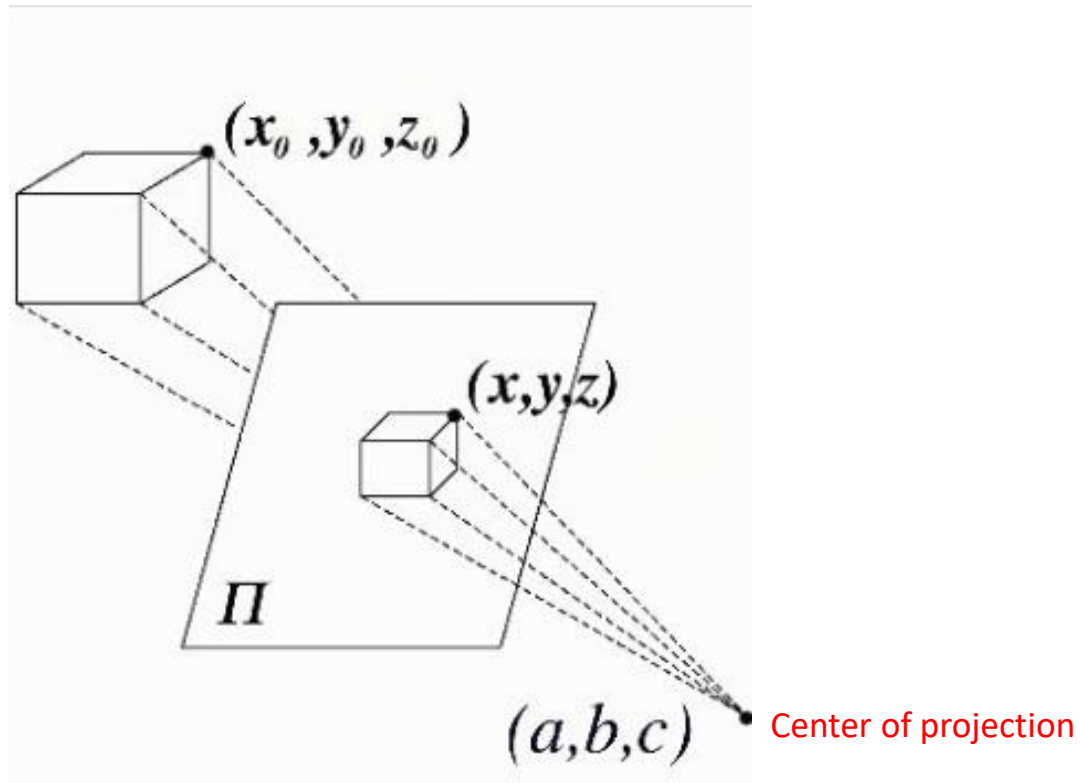
Projection types

In *parallel projections*, all rays are parallel to the same *direction*.



Projection types

In *perspective projections*, all the rays pass through a point, called the *center of projection*.



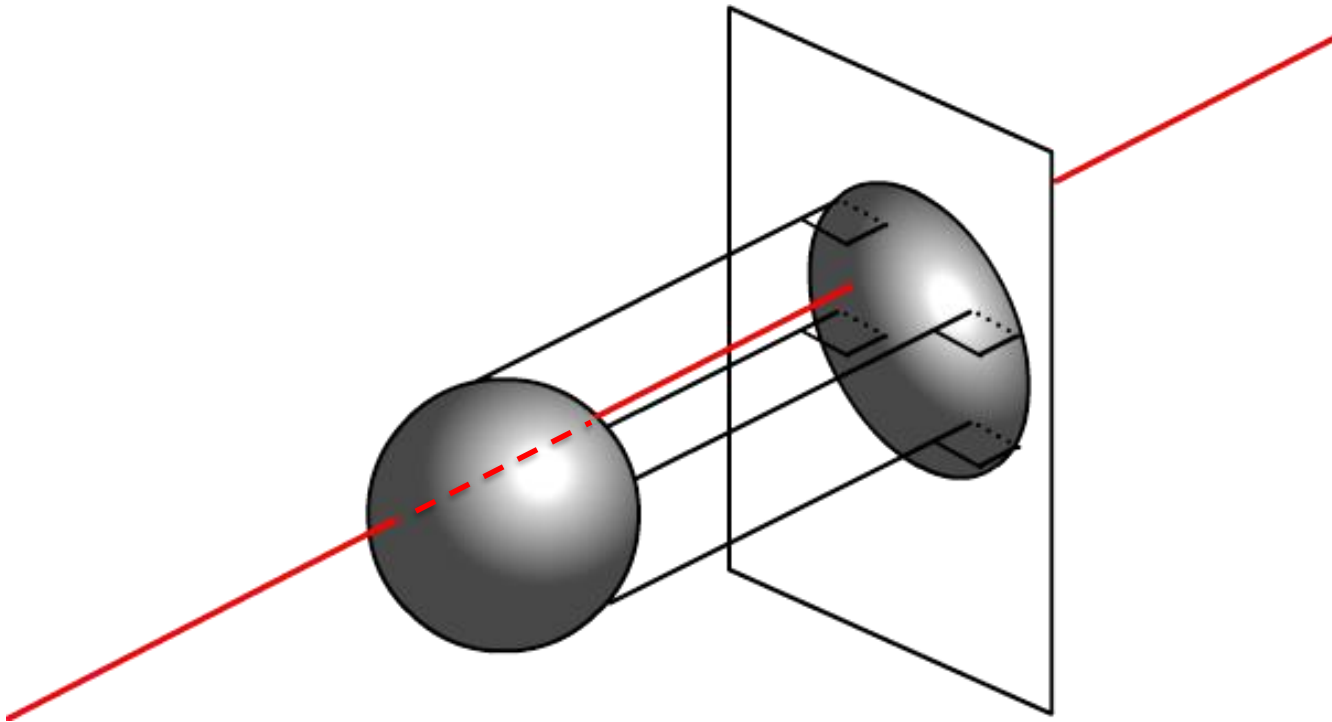
Projection properties

For both parallel and perspective projections, a point on the screen corresponds to an infinite number of coordinates in the space.

This is a direct consequence of losing a component when moving from a 3D spatial system, to a 2D screen system.

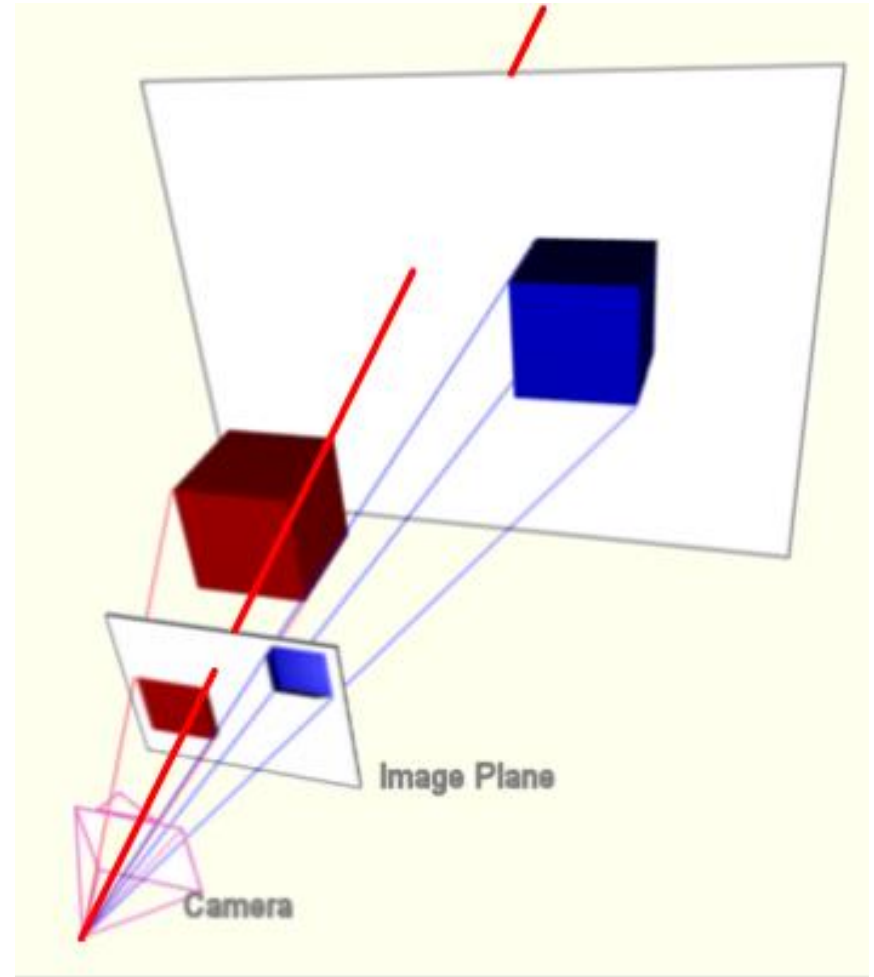
Projection properties

In parallel projections, all points that pass through a line parallel to the projection ray are mapped to same pixel.



Projection properties

In perspective projection, all points that are aligned with both the projected pixel and the center of projection are mapped to the same location.



Projection in 3D computer applications

In 3D computer applications, the projections are implemented with a conversion of 3D coordinates between two reference systems.

In particular *projections* transform:

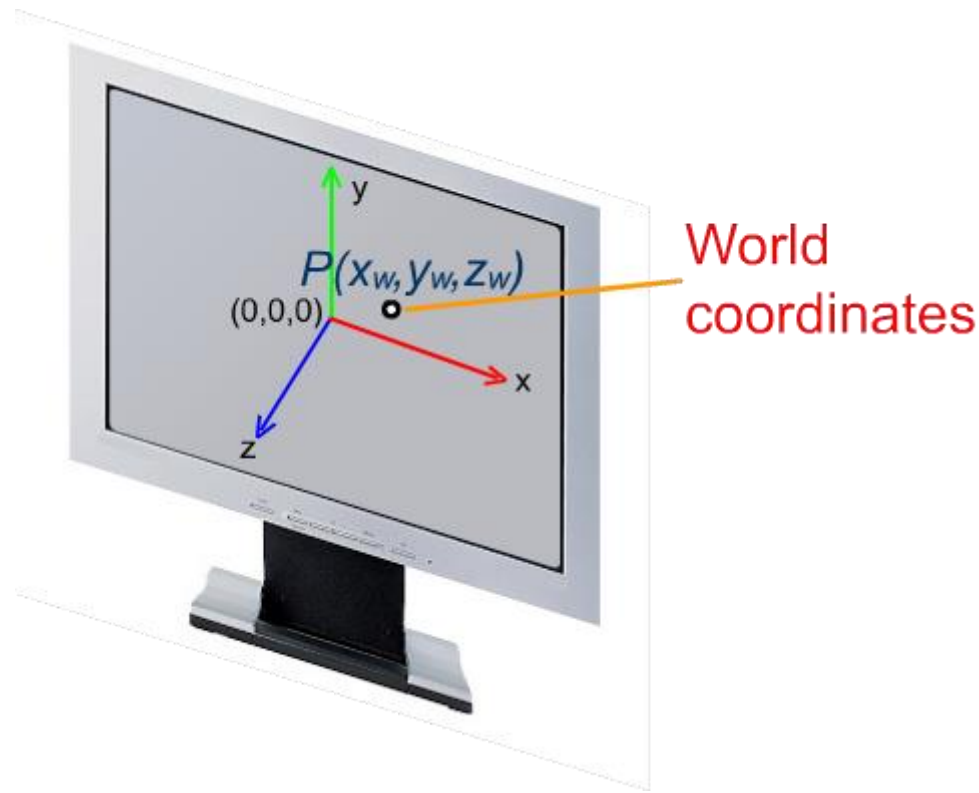
World Coordinates

Into:

3D Normalized Screen Coordinates

World Coordinates

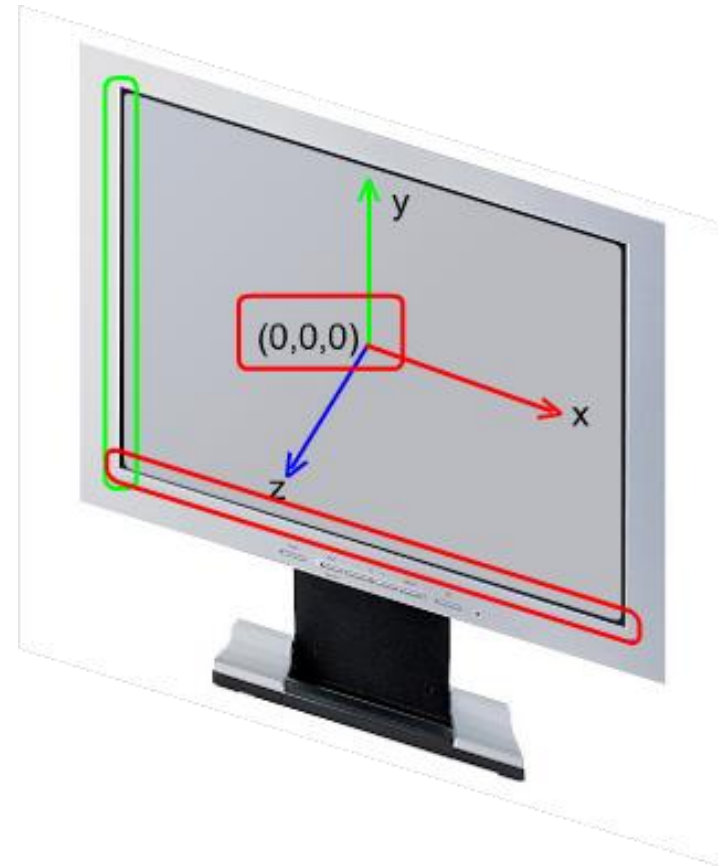
The system of coordinates that describes the objects in the 3D space is called *World Coordinates* (or *global coordinates*).



World Coordinates

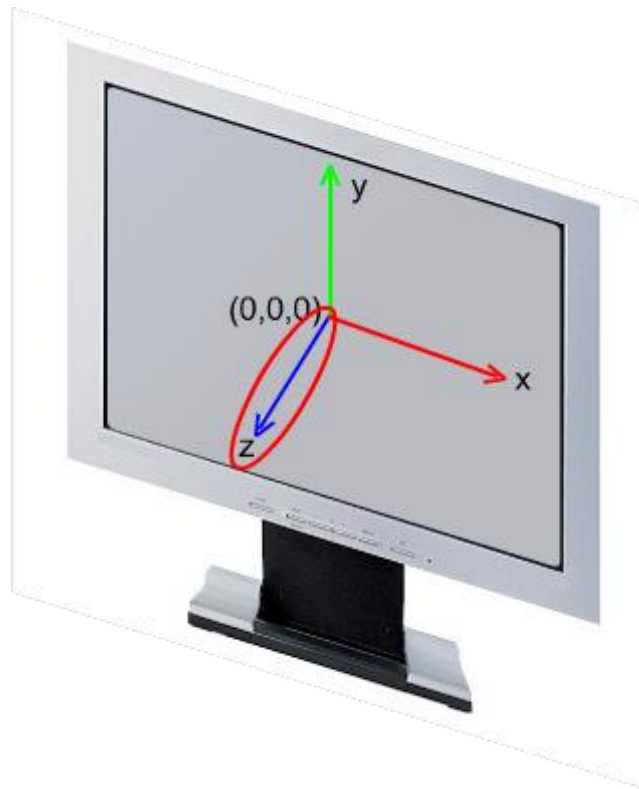
World Coordinates are a right-handed Cartesian coordinate system with the origin that is mapped to the center of the screen.

The version we will consider in this course is called “*y-up*”: it has the x and y -axes parallel respectively to the horizontal and vertical edges of the screen.



World Coordinates

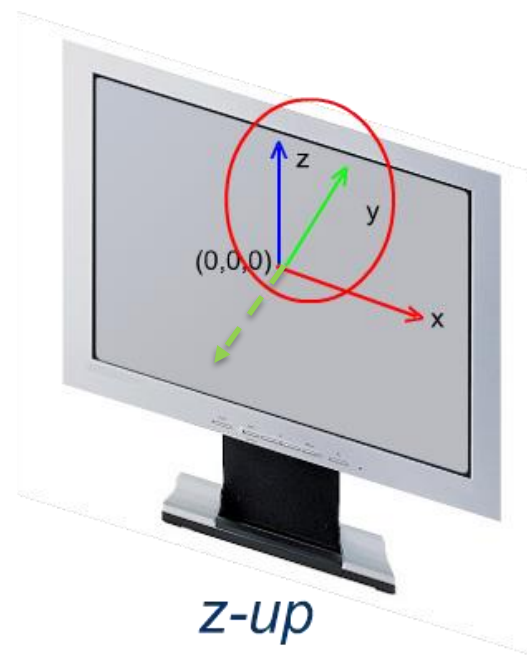
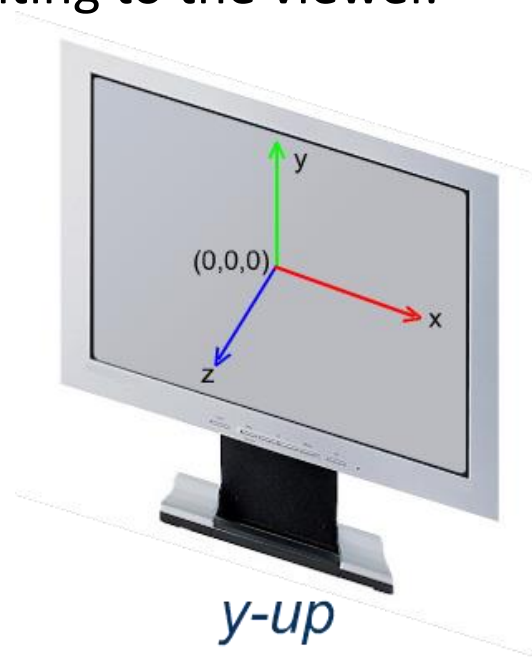
The *z-axis* is directed “out of the screen”, towards the viewer.



World Coordinates

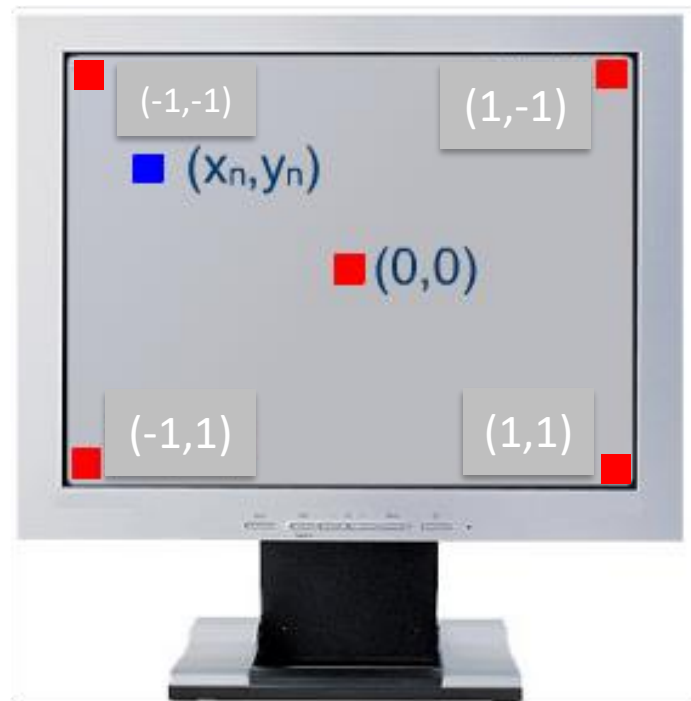
Note that many applications (i.e. *Blender*) use another convention for World Coordinates called “z-up”. In this case the *z-axis* is oriented along the vertical screen direction, and the *y-axis* points “inside the screen”. Some applications even use a left-handed system, with the *y* axis pointing to the viewer.

In this course we focus on the y-up system only.



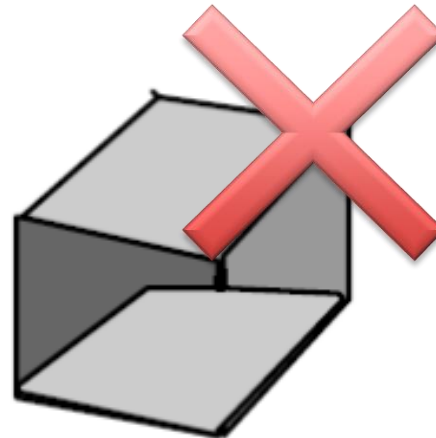
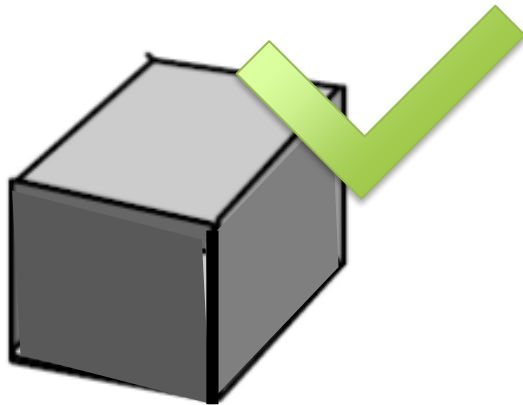
3D Normalized Screen Coordinates

As introduced, *Normalized Screen Coordinates* allow to specify the positions of points on a screen (or on a window) in a device independent way.



3D Normalized Screen Coordinates

However, even if the screen is a 2D surface, pixels coming from 3D images must be characterized by a “distance from the viewer” to allow sorting the surfaces in the correct way, and prevent the construction of unrealistic images (as we will see later).

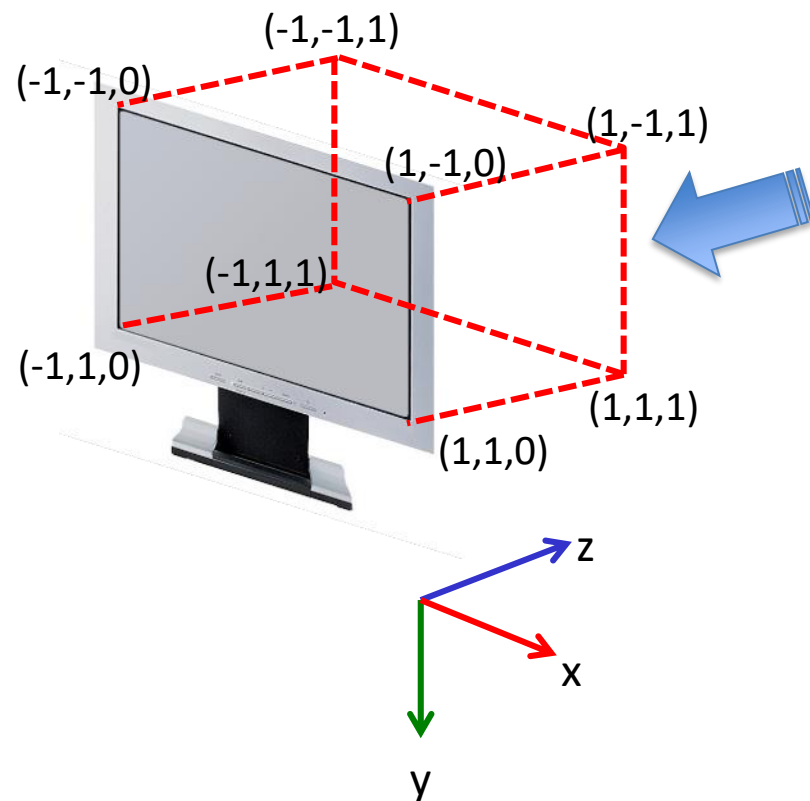


3D Normalized Screen Coordinates

3D Normalized Screen Coordinates have a third component that, when considering Vulkan, it is in the $[0,1]$ range.

Coordinates with smaller z value are considered to be closer to the viewer.

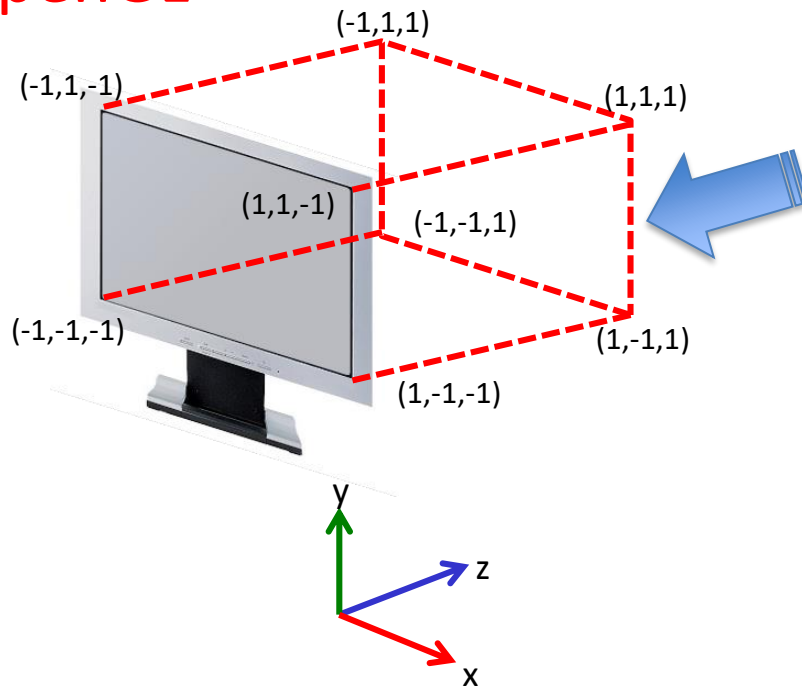
In the following, all normalized screen coordinates will be considered to be 3D.



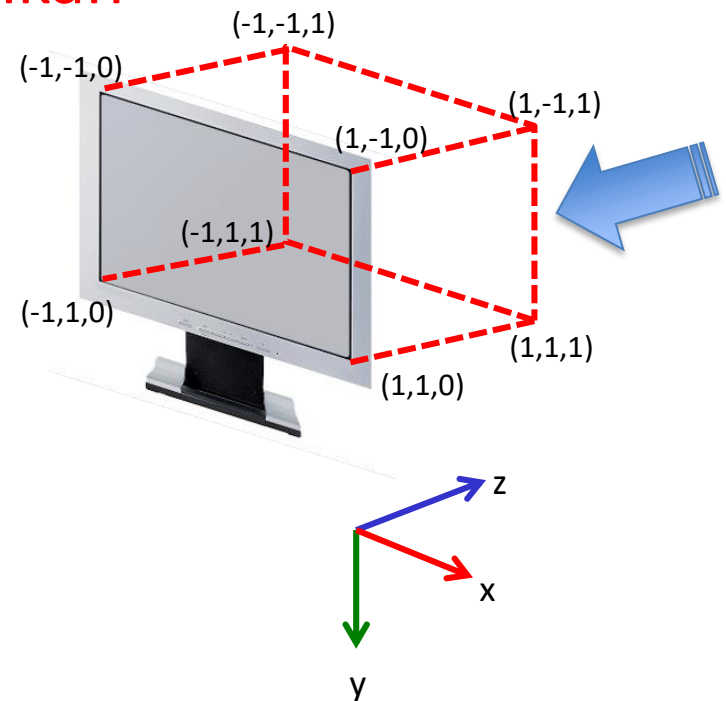
3D Normalized Screen Coordinates

Please note that other low level graphics engines, such as OpenGL, adopts completely different normalized screen coordinates systems.

OpenGL



Vulkan





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)