POLITECNICO MILANO 1863

DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

DEIB

2023

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2023

Computer Graphics

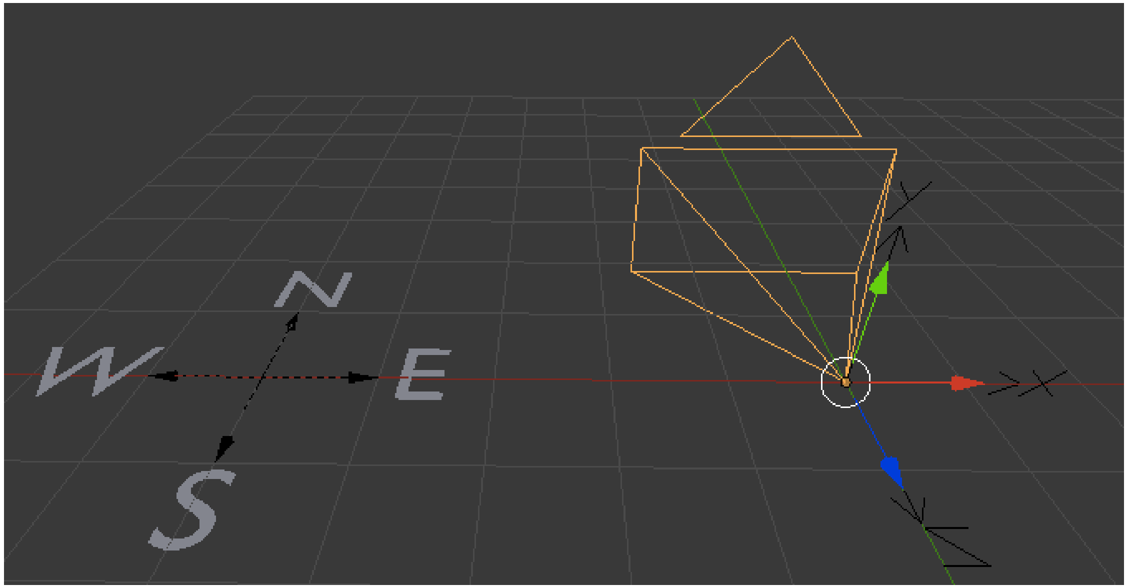
- View Matrix

Marco Gribaudo - DEIB

Axis and orientation

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

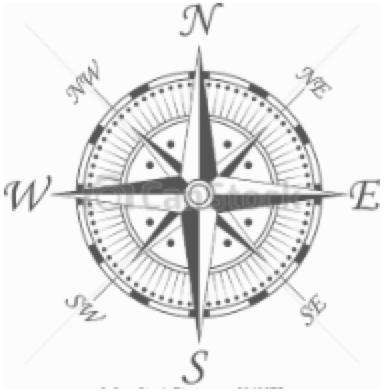
We have defined the *world coordinates* as a reference system to globally specify the positions of an object in the 3D space.

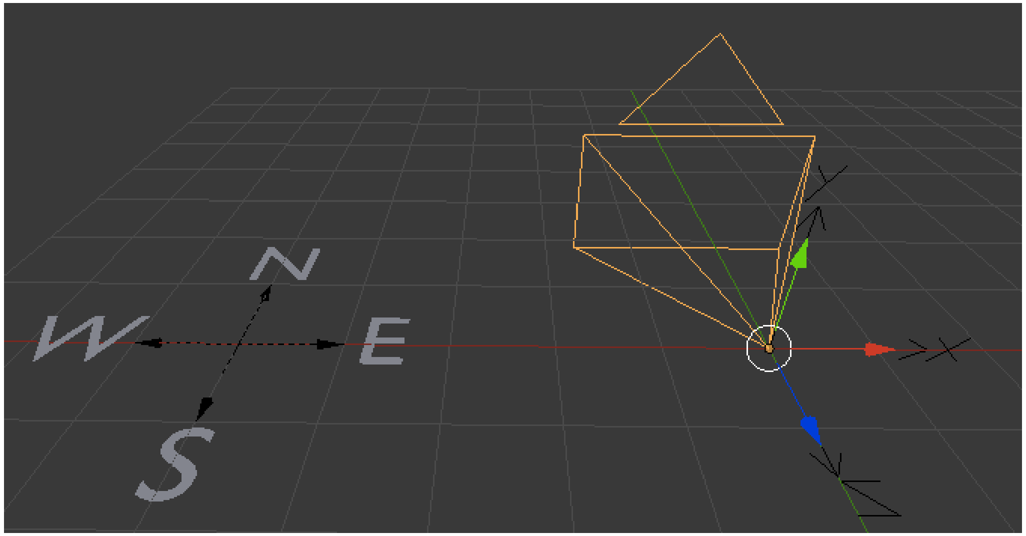


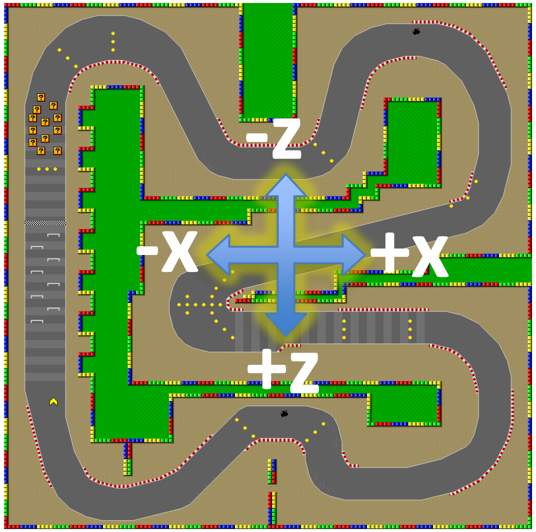
Axis and orientation

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

In particular, we will call the *negative z-axis* the *North* direction in the 3D world, and that the *positive x-axis* the *East*.







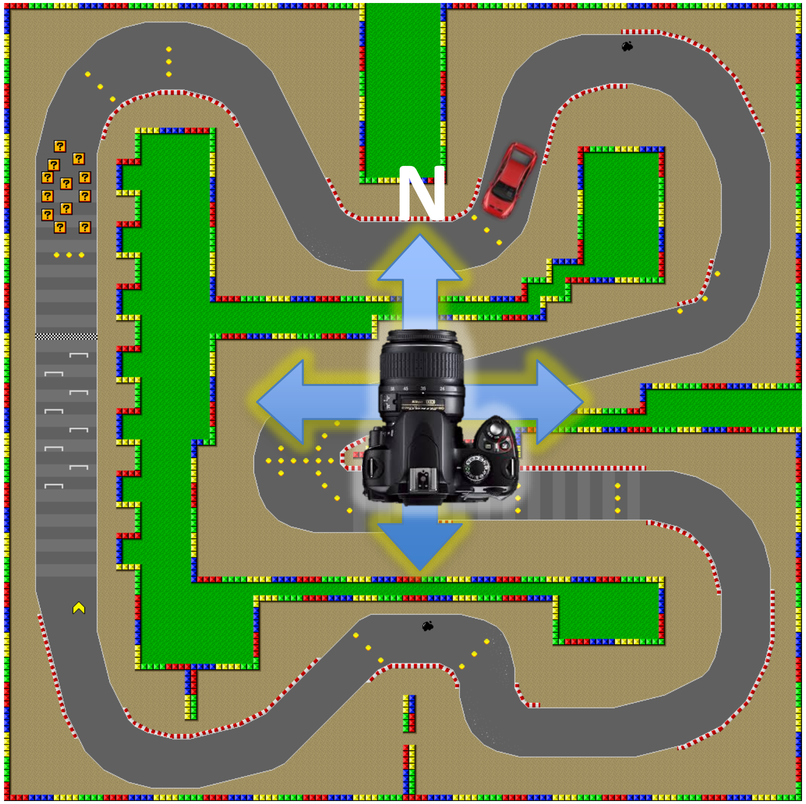
The View Matrix

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

For parallel projections, we also assume that the projection rays are oriented along the negative *z-axis*.

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

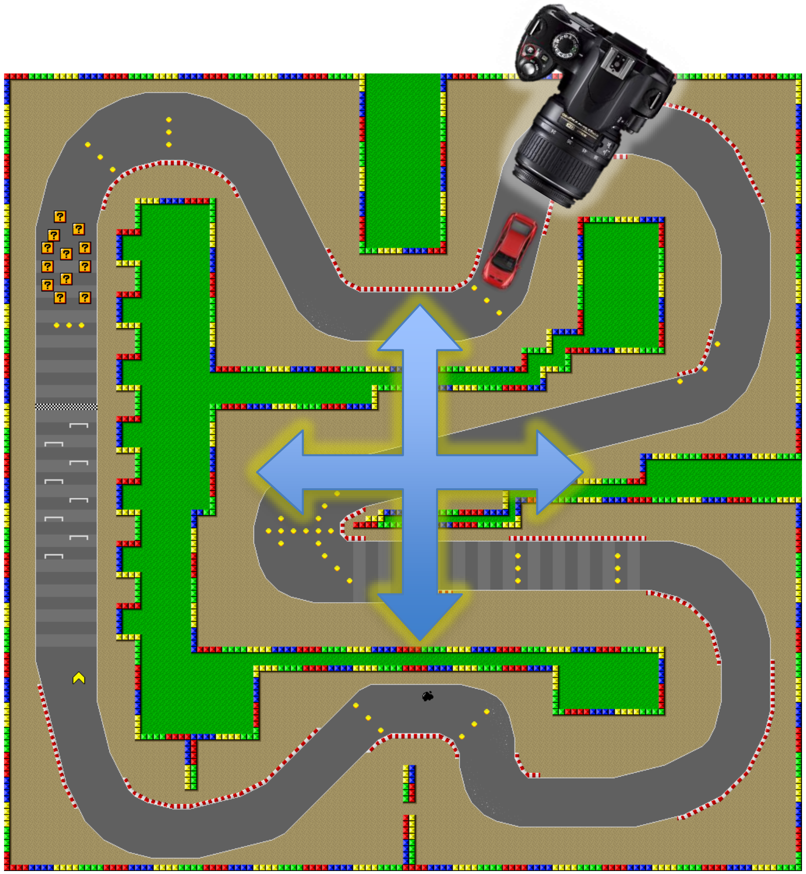
In both cases, it corresponds to a camera looking *North*.



The View Matrix

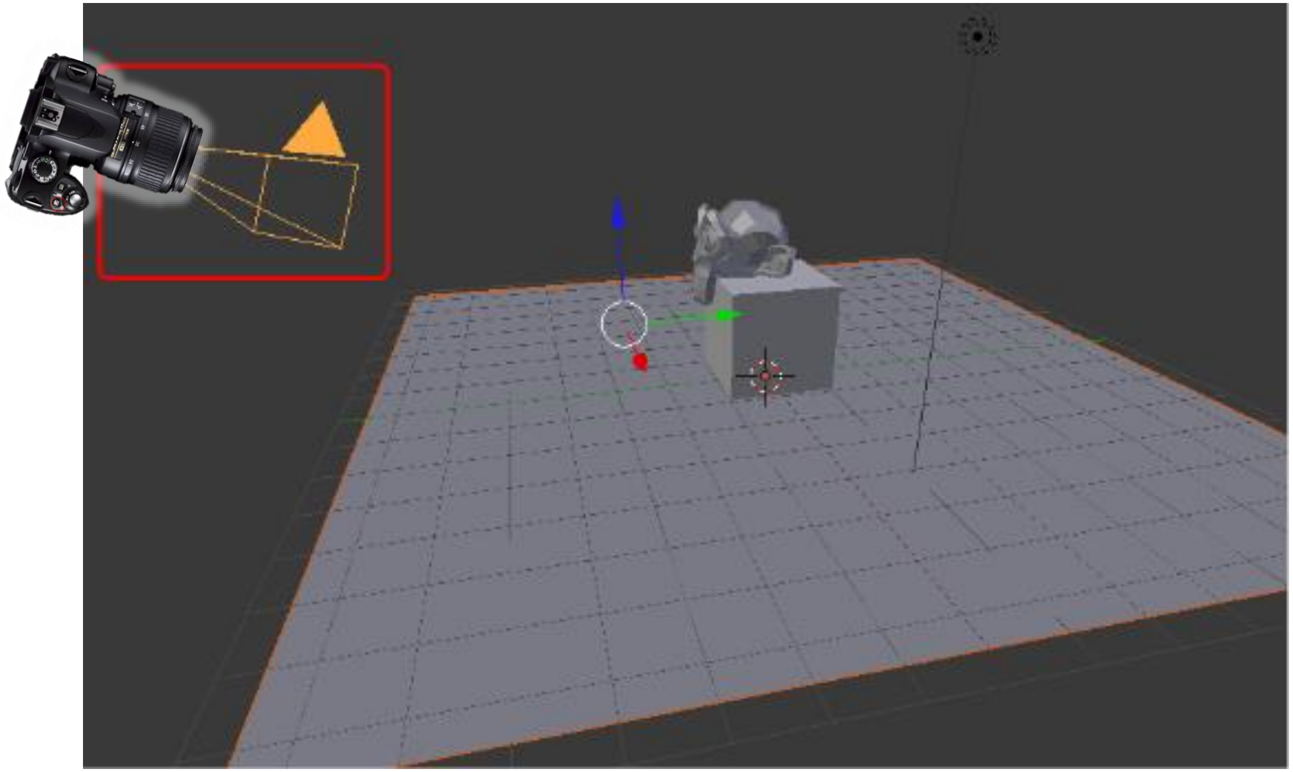
In actual applications, we are interested in generating a 2D view for a plane arbitrarily positioned in the space.

This is achieved by adding a set of additional transformations before the projection.



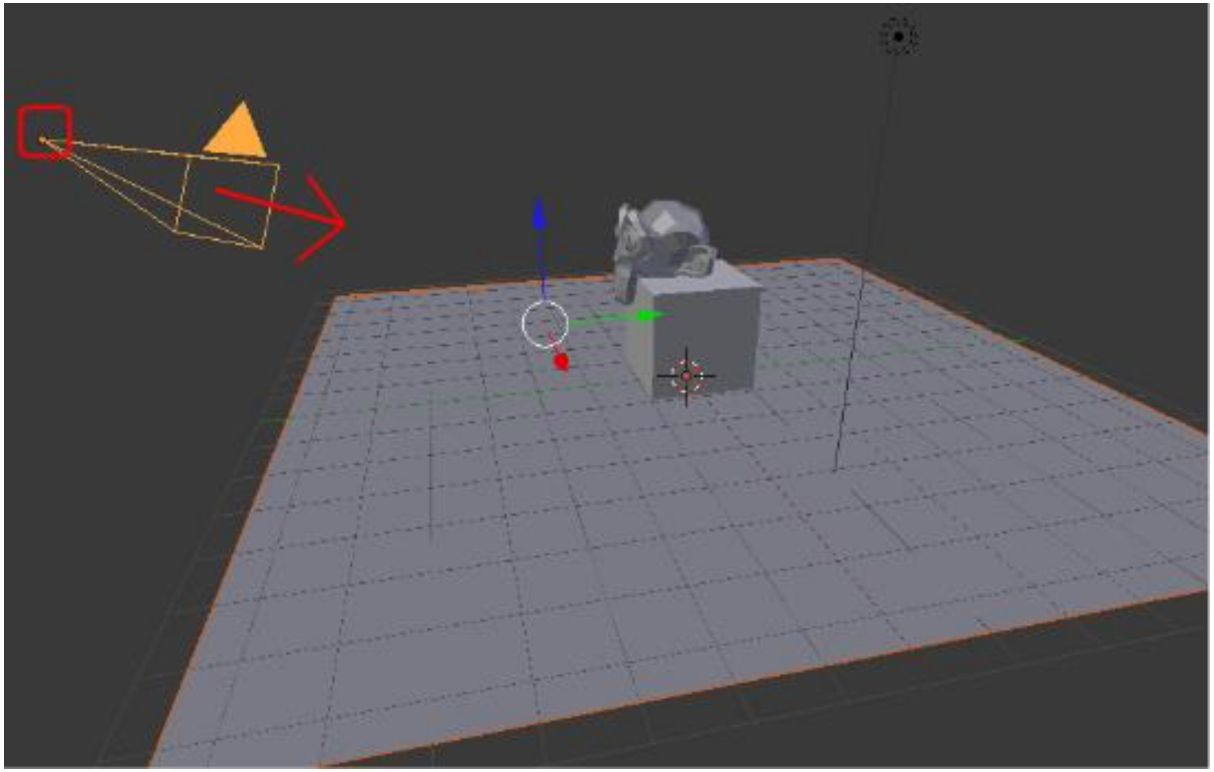
The View Matrix

We will focus on perspective, but the same reasoning applies also to parallel projection. The projection plane can be seen as the view of a *camera* that looks at the scene from the center of projection.



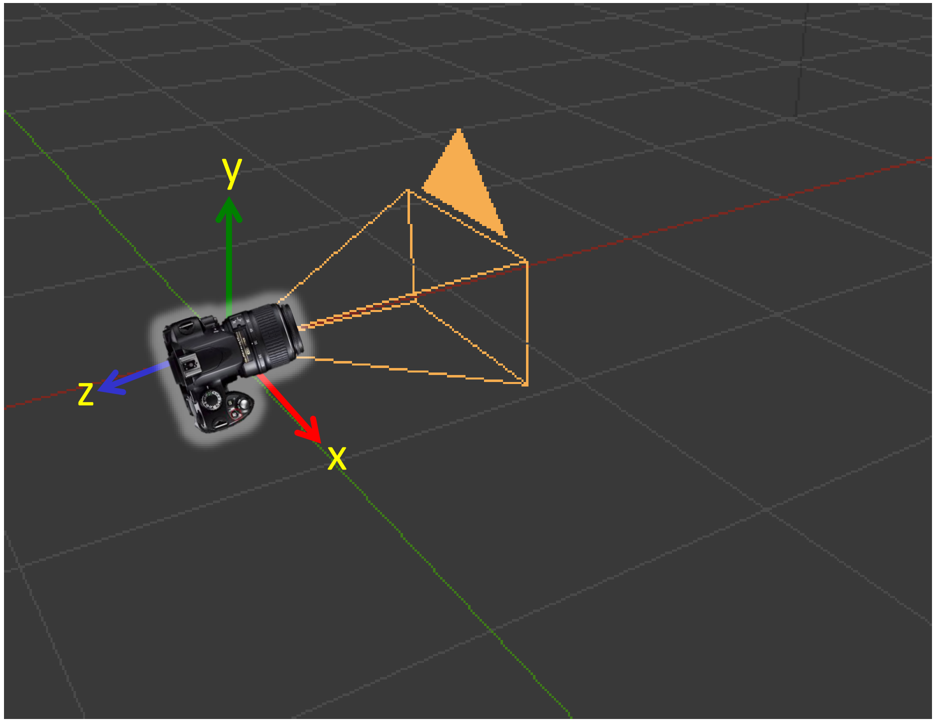
The View Matrix

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



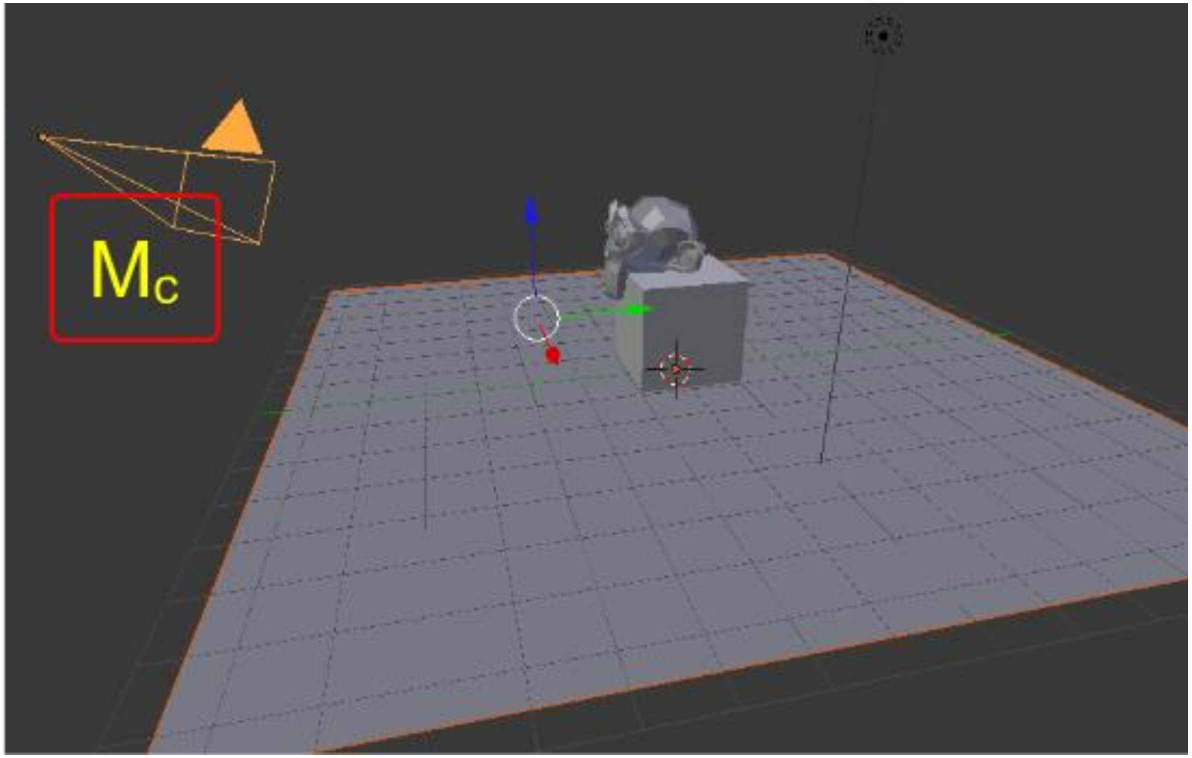
The View Matrix

The projection matrices that we have seen, compute the view of a *camera* initially positioned in the origin, and aiming along the negative *z-axis*. We can consider this camera as a 3D object.



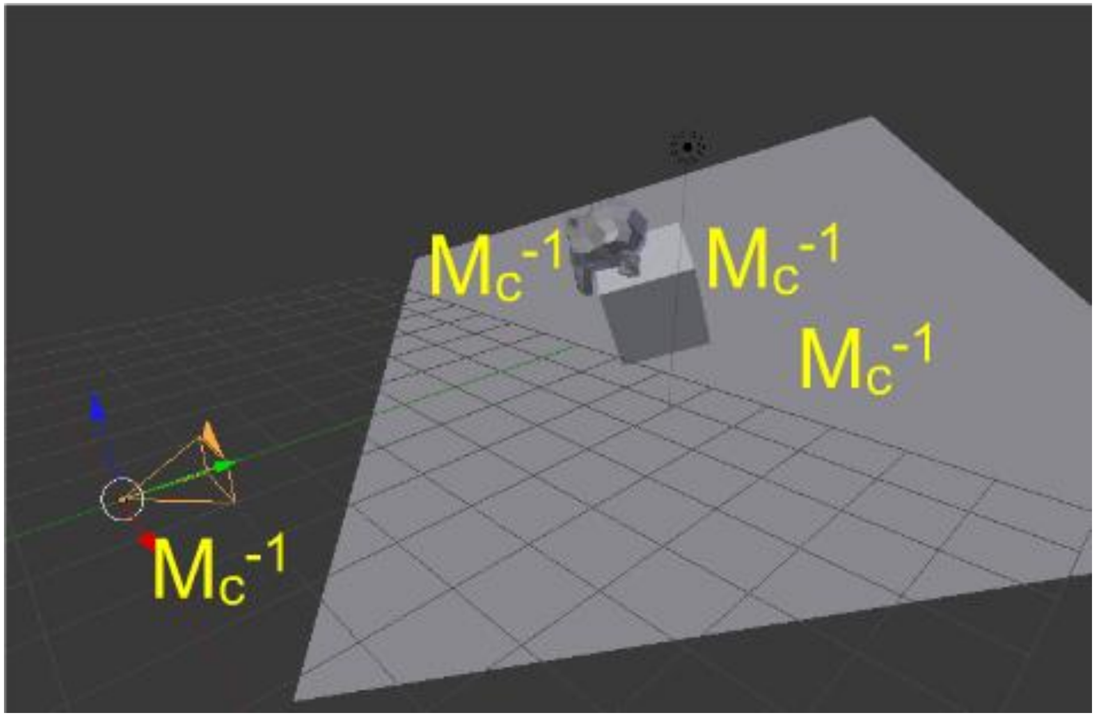
The View Matrix

We can then define a transformation matrix M_c that moves this camera object to its target position that orient the negative *z-axis* along the desired direction. We will call this matrix the *Camera Matrix*.



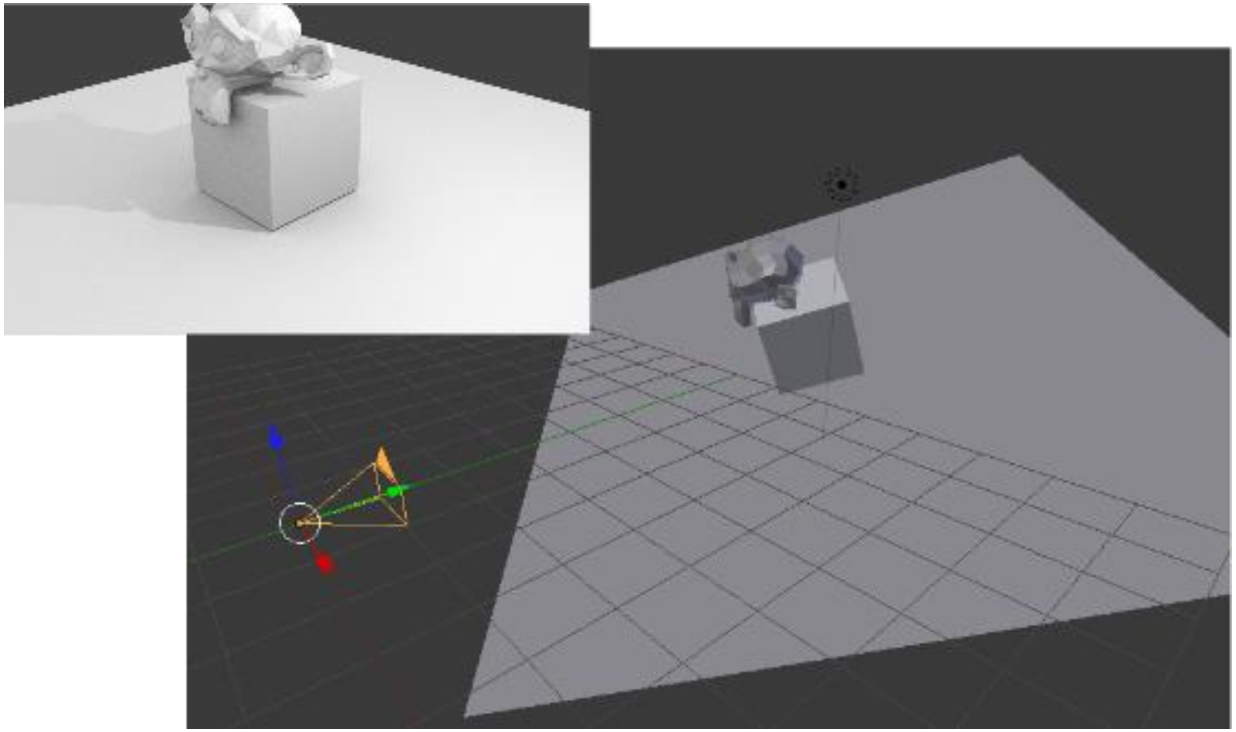
The View Matrix

If we apply the inverse of M_c to all the objects in the scene, we obtain a new 3D world where the projection plane is placed exactly as required by the projection transformations seen in the previous lessons.



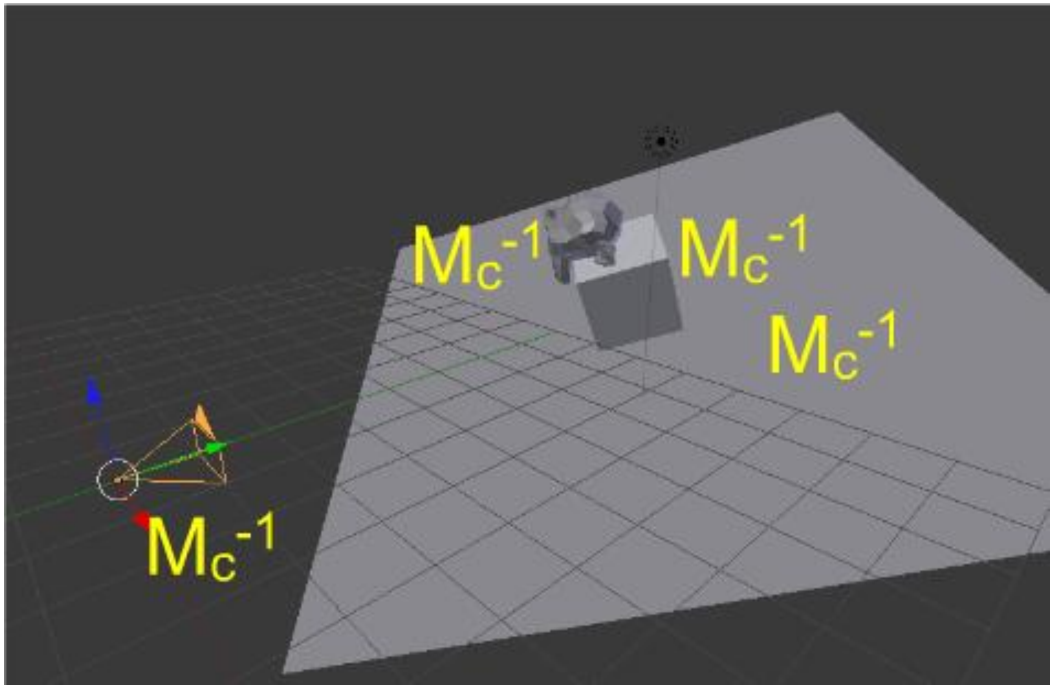
The View Matrix

In this new space, the view of the 3D world as seen from the arbitrarily oriented camera can be computed by applying the projections matrices we have seen.



The View Matrix

Matrix M_c^{-1} is known as the *View Matrix*, and we will denote it as M_v.



$$M_V = M_C^{-1}$$

The View-Projection Matrix

The view matrix is placed before the projection matrix M_{Prj} previously introduced: in this way, it allows us to find the normalized screen coordinates of the points in space, as seen by the considered camera.

This matrix is sometimes known as the *view-projection matrix*.

$$M_{VP} = M_{Prj} \cdot M_V$$

It transforms a point from world coordinates (homogeneous coordinates), to 3D normalized screen coordinates (cartesian) as seen by a given camera characterized by matrix M_{Prj}, in a given location and direction in space according to matrix M_V.

Creating a View Matrix

Several techniques exist to create a view matrix in a user-friendly way. The two most popular are:

- The *look-in-direction* technique
- The *look-at* technique

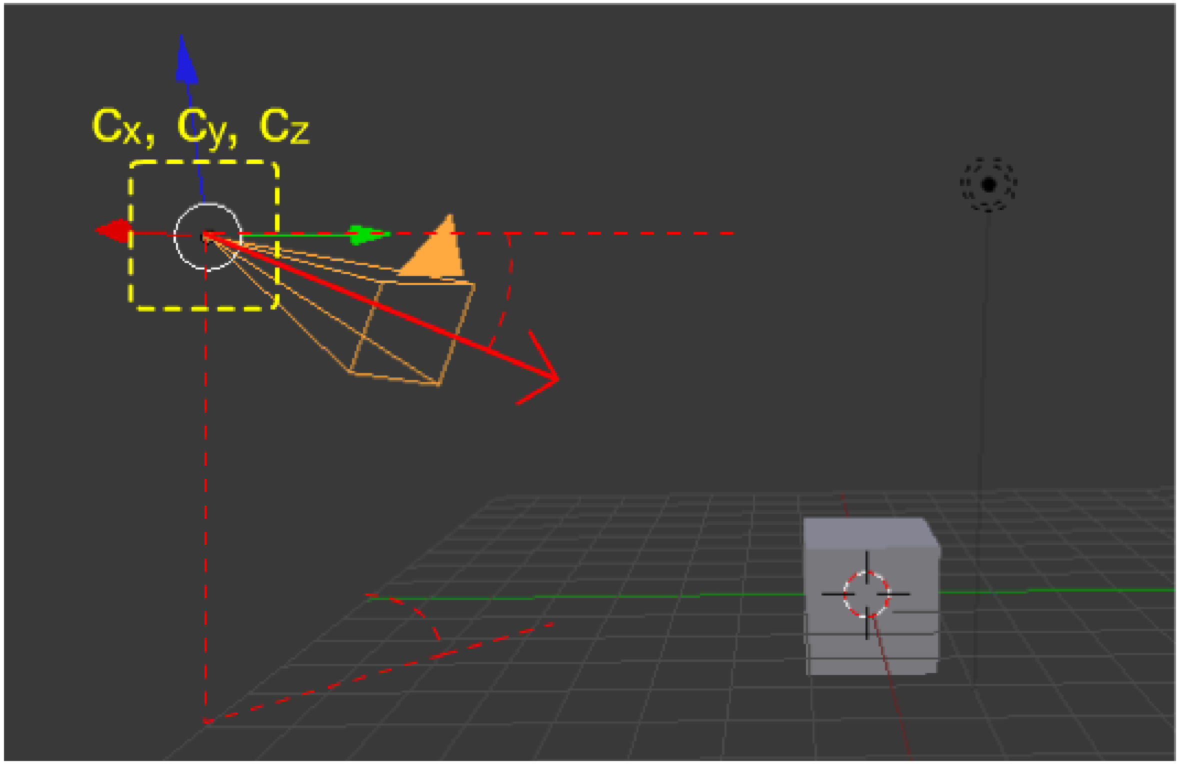
View Matrix: look-in-direction

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

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

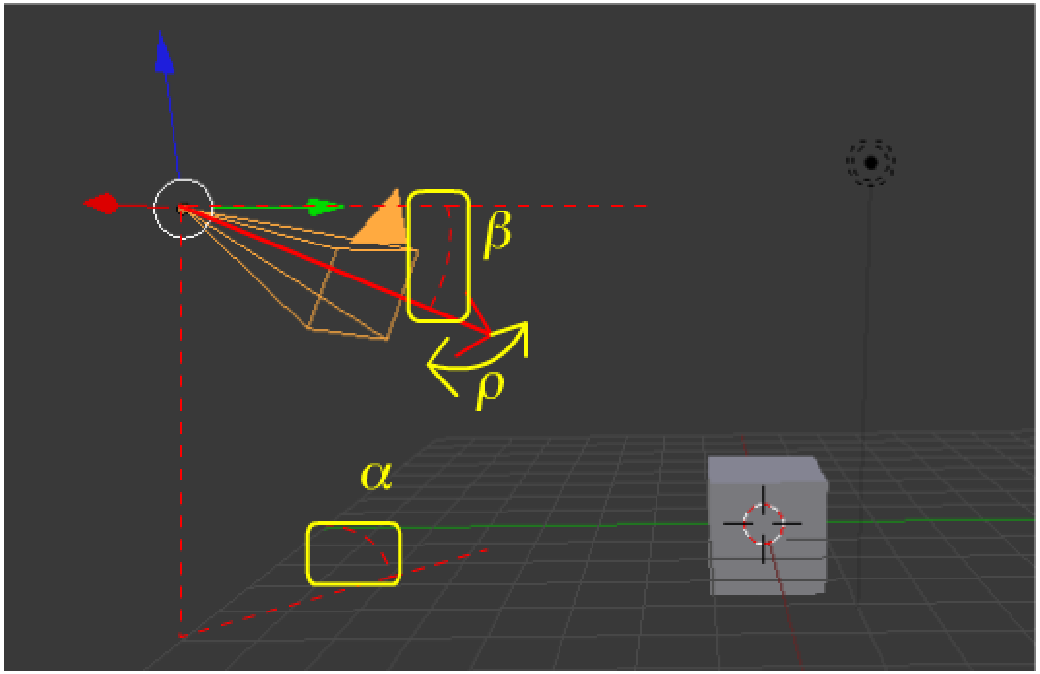
View Matrix: look-in-direction

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



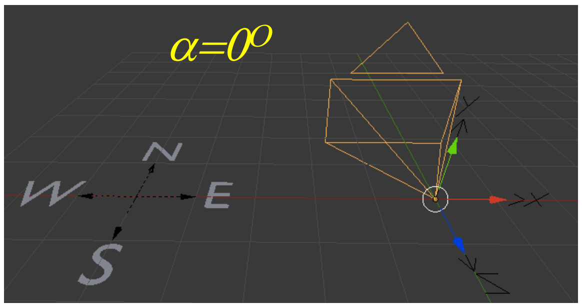
View Matrix: look-in-direction

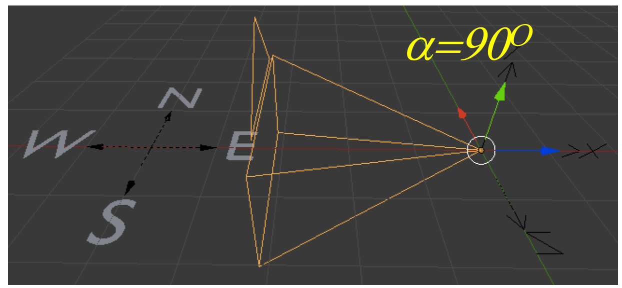
The direction where the camera is looking is specified with three angles: the “compass” direction (angle α), the elevation (angle β), and the roll over the viewing direction (angle ρ). This last parameter however is rarely used.

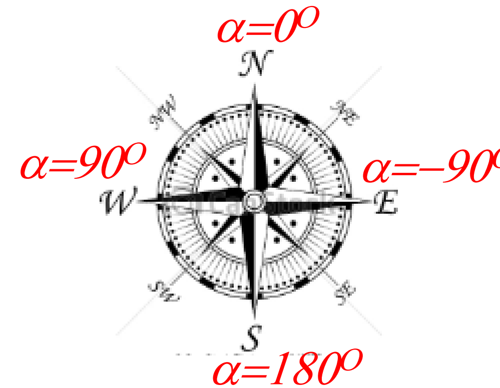


View Matrix: look-in-direction

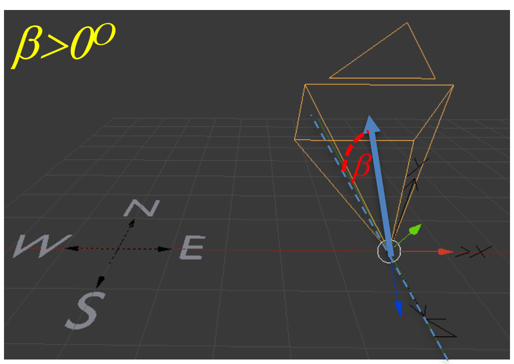
In particular, with $\alpha=0^O$ the camera looks *North*, while with $\alpha=90^O$ the camera looks *West*. *South* is $\alpha=180^O$ and *East* is $\alpha=-90^O=270^O$.



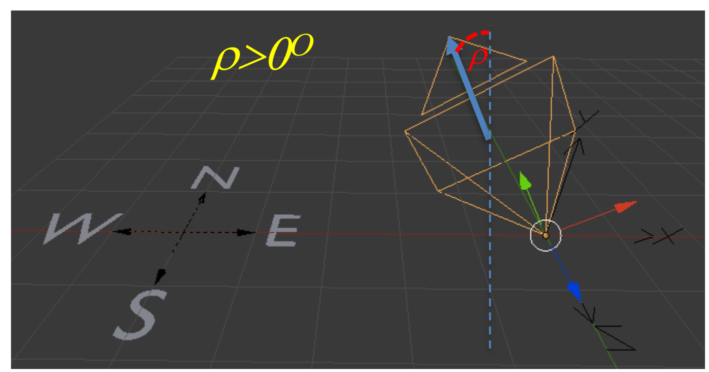




A positive angle $\beta>0^O$ makes the camera look up.

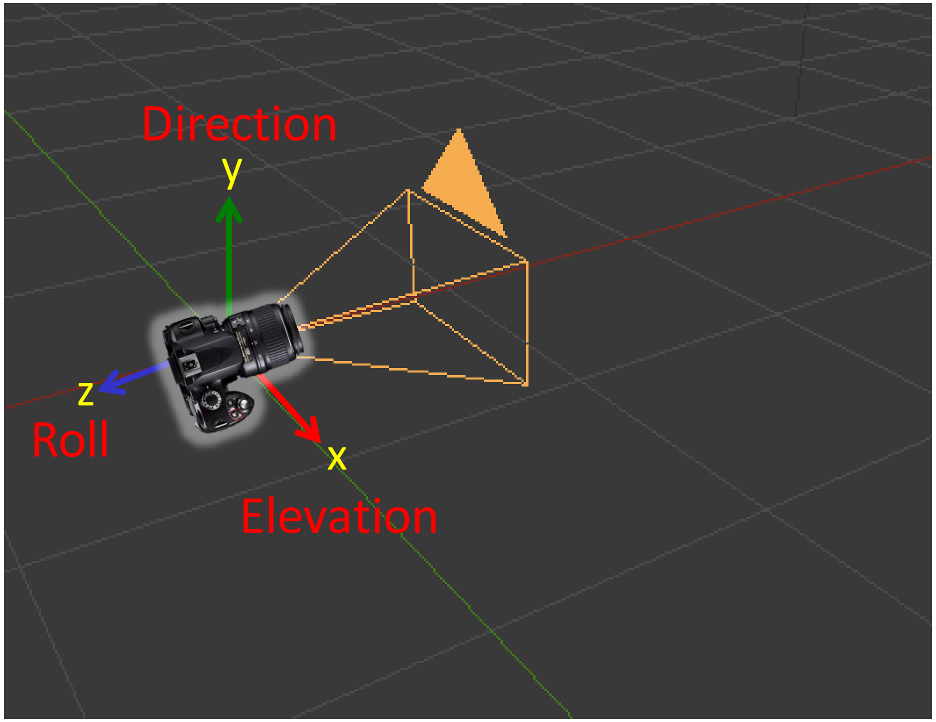


A positive angle $\rho>0^O$ turns the camera counterclockwise.



The View Matrix

Considering the camera object, roll corresponds to a rotation around the *z-axis*, elevation (also known as *pitch*) along the *x-axis*, and direction *(also known as yaw)* around the *y-axis*.



View Matrix: look-in-direction

Rotations must be performed in a specific order (we will return on this next lesson): roll must be performed first, then the elevation, and finally the direction. Translation is performed after the rotations. The *Camera Matrix* is then composed in this way:

$$M_C = T(c_x, c_y, c_z) \times R_y(a) \times R_x(b) \times R_z(r)$$

The *View Matrix*, is the inverse of the *Camera Matrix*. Remembering the rules and pattern for inverting a chain of transformations, we have:

$$M_V = \left(M_C\right)^{-1} = R_z(-r) \times R_x(-b) \times R_y(-a) \times T(-c_x, -c_y, -c_z)$$

Look in matrix in GLM

GLM does not provide any special support to build a *Look-in-direction* matrix. However, due to its simplicity, it can be easily implemented using what we have seen in the previous lessons:

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

```
glm::mat4 Mv =
    glm::rotate(glm::mat4(1.0),   -rho, glm::vec3(0,0,1)) *
    glm::rotate(glm::mat4(1.0),  -beta, glm::vec3(1,0,0)) *
    glm::rotate(glm::mat4(1.0), -alpha, glm::vec3(0,1,0)) *
    glm::translate(glm::mat4(1.0), glm::vec3(-cx, -cy ,-cz));
```

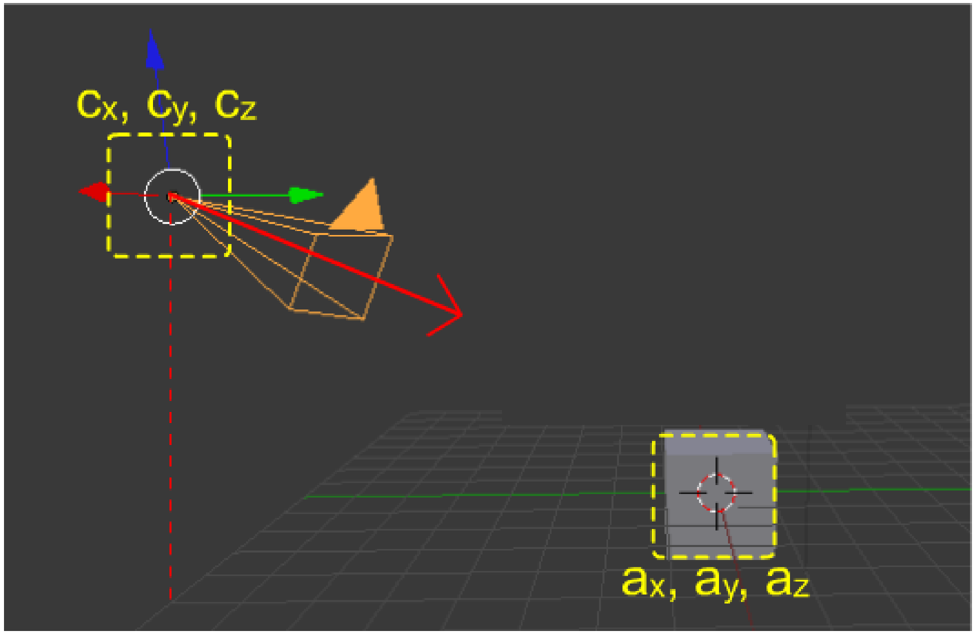
View Matrix: look-at

The *look-at* model is instead generally employed in *third person applications*.

In this case, the camera tracks a point (or an object) aiming at it.

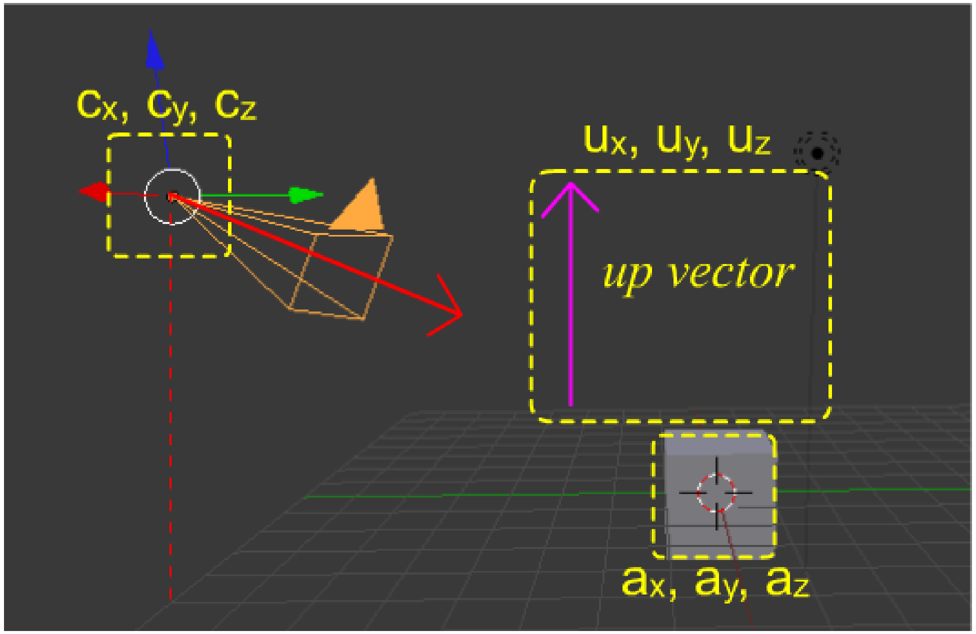
View Matrix: look-at

In the *look-at* model, the center of the camera is positioned at $c=(c_x, c_y, c_z)$, and the target is a point of coordinates $a=(a_x, a_y, a_z)$.



View Matrix: look-at

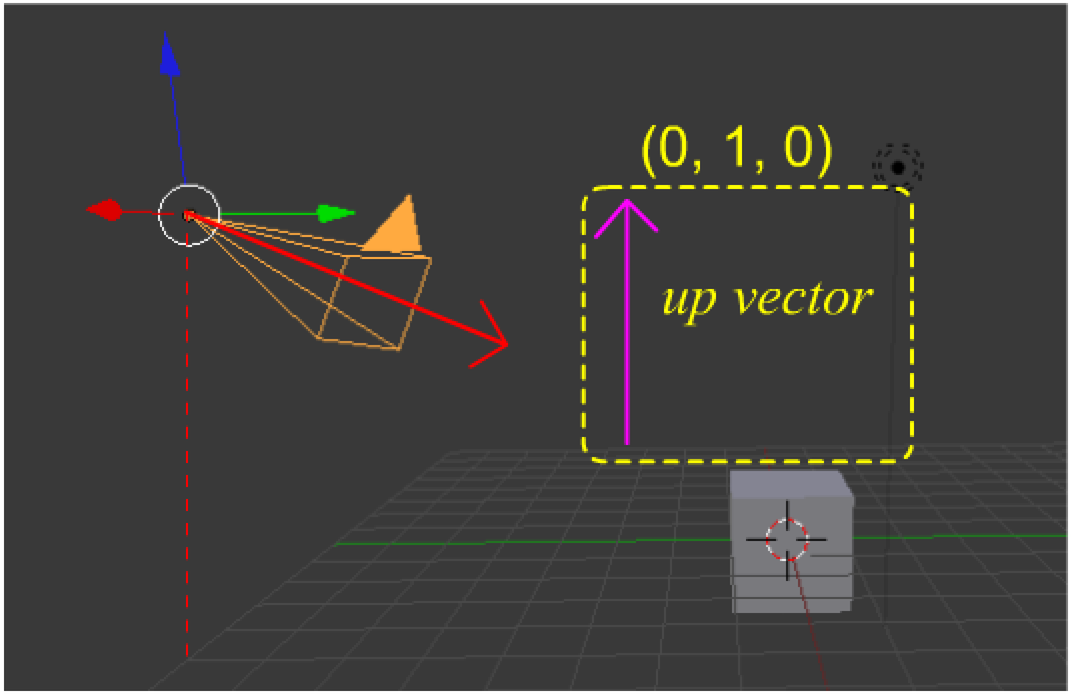
The technique also requires the *up vector*: the direction $u=(u_x, u_y, u_z)$ perpendicular to "*the ground*" of the scene.



View Matrix: look-at

In *y-up* coordinate systems, it is usually set to *u=(0, 1, 0).*

In this way, the camera oriented with the y-axis perpendicular to the horizon.



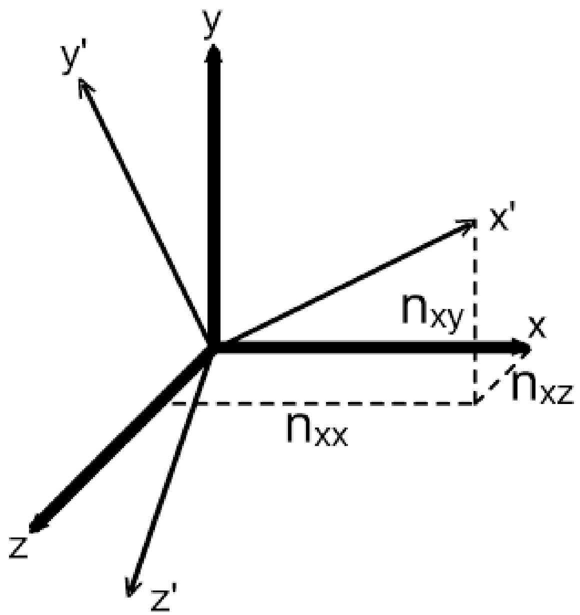
View Matrix: look-at

In some applications however, the direction of the up vector is changed to obtain interesting effects on the game plays.

Super Mario Galaxy, 2007, Nintendo Wii, or 2020 Nintendo Switch

View Matrix: look-at

The *View Matrix* is computed by first determining the direction of its axis in World coordinates, and then using the corresponding information to build the *Camera Matrix*.



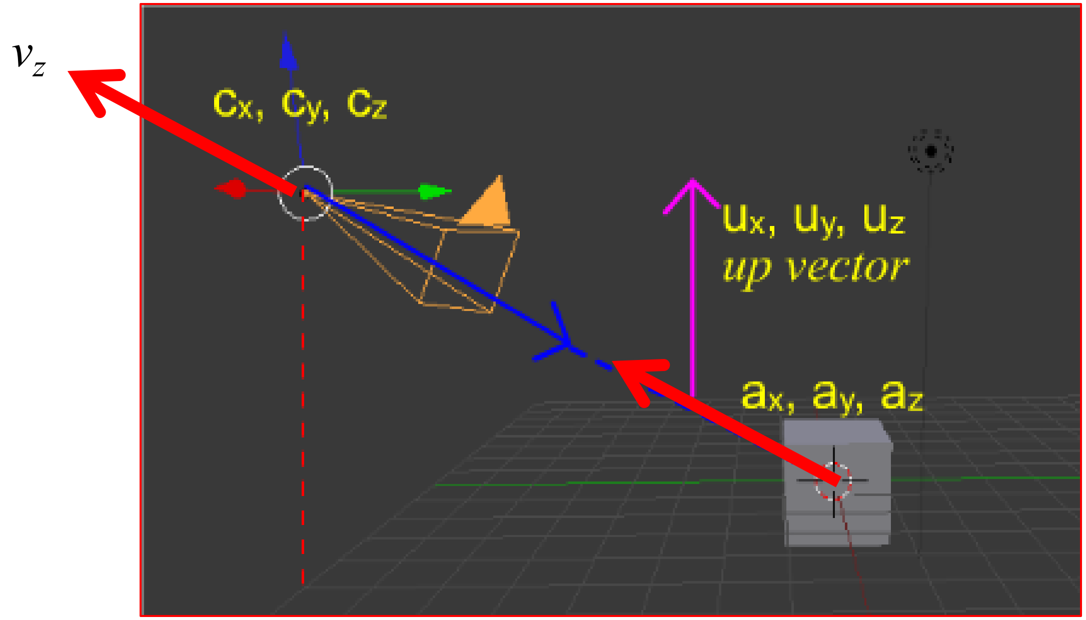
$$M_R = \begin{vmatrix} \boxed{\begin{matrix} n_{xx} \\ n_{xy} \\ n_{xz} \end{matrix}} & \begin{matrix} n_{yx} \\ n_{yy} \\ n_{yz} \end{matrix} & \begin{matrix} n_{zx} \\ n_{zy} \\ n_{zz} \end{matrix} \end{vmatrix}$$

View Matrix: look-at

We first determine the transformed (negative) *z-axis* as the normalized vector that ends into the camera center and that starts from the point that it is looking.

Normalization (unit size) is obtained by dividing for the length of the resulting vector.

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

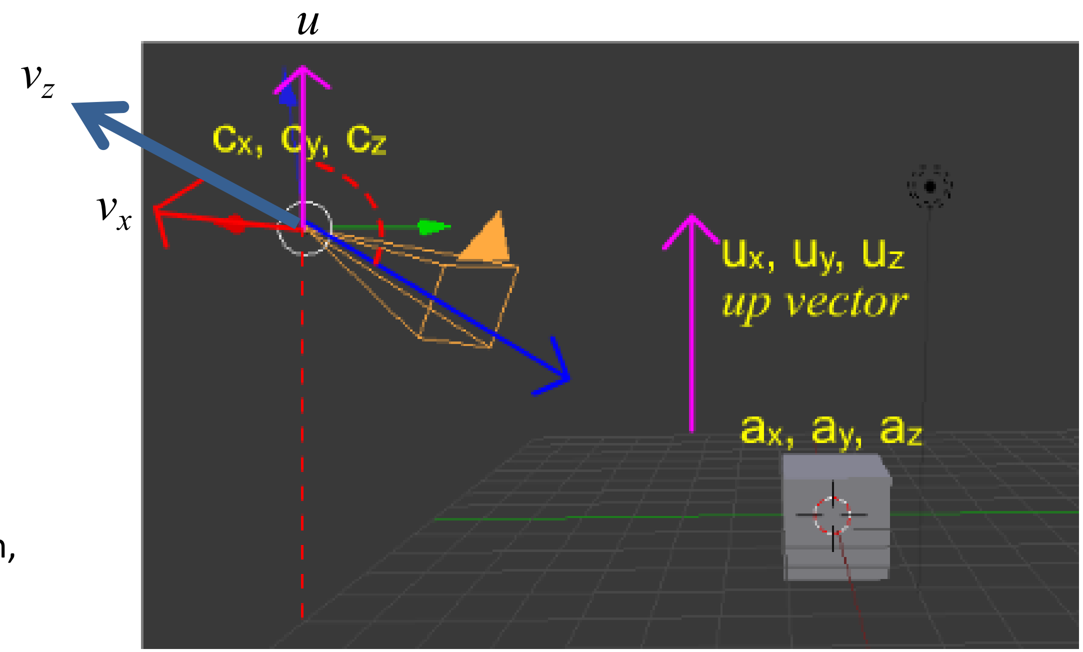


The new x-axis must be perpendicular to both the new *z-axis*, and the *up-vector*: it can be computed via the normalized cross product of the two.

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

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

Even if both vectors are unitary, they are not always perpendicular. Without normalization, the result will be of a non-unitary size.



For convenience, the cross product of two 3 components vectors is defined as:

$$|u_x \quad u_y \quad u_z| \times |v_x \quad v_y \quad v_z| = |u_y v_z - u_z v_y \quad u_z v_x - u_x v_z \quad u_x v_y - u_y v_x|$$

View Matrix: look-at

Note that the cross product returns zero if the two vectors u and v_z are aligned.

This makes it impossible to determine vector v_x, and thus to find a proper camera matrix.

Such problem occurs when the viewer is perfectly vertical, and thus it is impossible to align the camera with the ground: the simplest solution is to use the previously computed matrix, or select a random orientation for the *x-axis*.

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

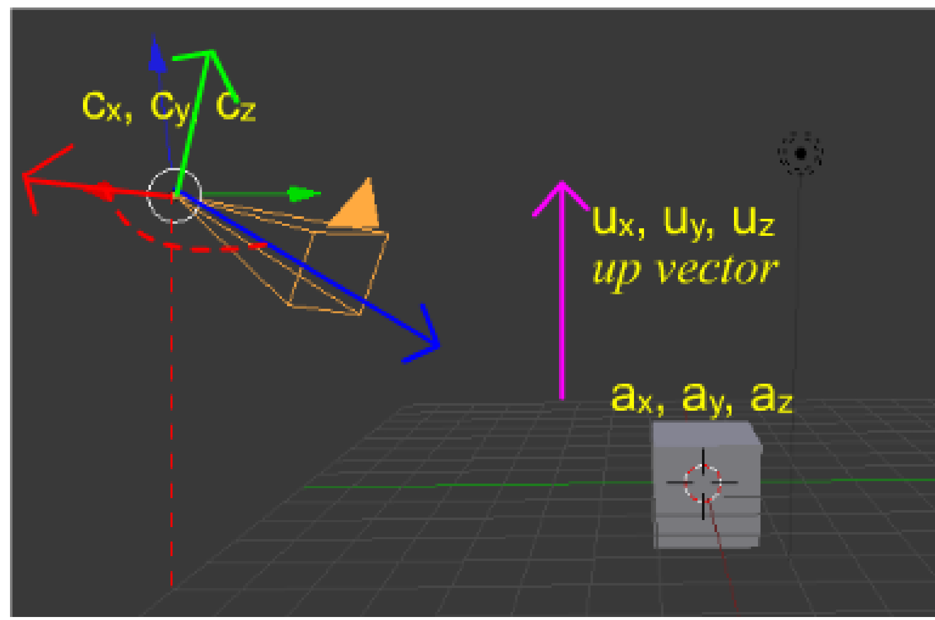
View Matrix: look-at

Finally, the new *y-axis* should be perpendicular to both the new *z-axis* and the new *x-axis*. This could be computed via the cross product of the two vectors just obtained.

Since both the new *z-axis* and the new *x-axis* are by construction perpendicular and unit vectors, normalization is not required.

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

$$v_y = v_z \times v_x$$



View Matrix: look-at

The *Camera Matrix* M_C can then be computed by placing vectors v_x, v_y and v_z in the first three columns and the position of the center c in the fourth.

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

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

$$v_y = v_z \times v_x$$

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

View Matrix: look-at

The *View Matrix* can be computed inverting M_C. Since the vectors are orthogonal, the inversion of a look-at camera matrix can be computed very easily with a transposition and a matrix-vector product:

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

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

$$v_y = v_z \times v_x$$

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

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

Cross product, normalization and definition by column in GLM

GLM can compute the cross product of two vectors using the `cross()` function:

```
glm::vec3 uXvz = glm::cross(u, vz);
```

A vector can be made unitary with the `normalize()` function:

```
glm::vec3 vx = glm::normalize(uXvz);
```

Finally, there is a constructor to build a 4x4 matrix starting from 4 column vectors:

```
glm::mat4 Mc = glm::mat4(vx, vy, vz, glm::vec4(0,0,0,1));
```

Cross product, normalization and definition by column in GLM

In particular, it can be implemented as:

```
glm::vec3 vz = glm::normalize(c - a);
glm::vec3 vx = glm::normalize(glm::cross(u, vz));
glm::vec3 vy = glm::cross(vz, vx);
glm::mat4 Mc = glm::mat4(vx, vy, vz, glm::vec4(0,0,0,1));
Glm::mat4 Mv = glm::inverse(Mc);
```

Where `a`, `c` and `u` are three `glm::vec3` representing respectively the position of the camera, the target point and the up vector, and `Glm::mat4 Mv` is the computed view matrix.

Look at matrix in GLM

Moreover, GLM has the `lookAt()` functions that creates a Look-at matrix starting from three `glm::vec3` vectors representing respectively the center of the camera, the point it targets, and it up-vector:

```
glm::mat4 Mv = glm::lookAt(glm::vec3(cx, cy, cz),
                           glm::vec3(ax, ay, az),
                           glm::vec3(ux, uy, uz));
```

```
glm::mat4 Mv = glm::lookAt(c, a, u);
```

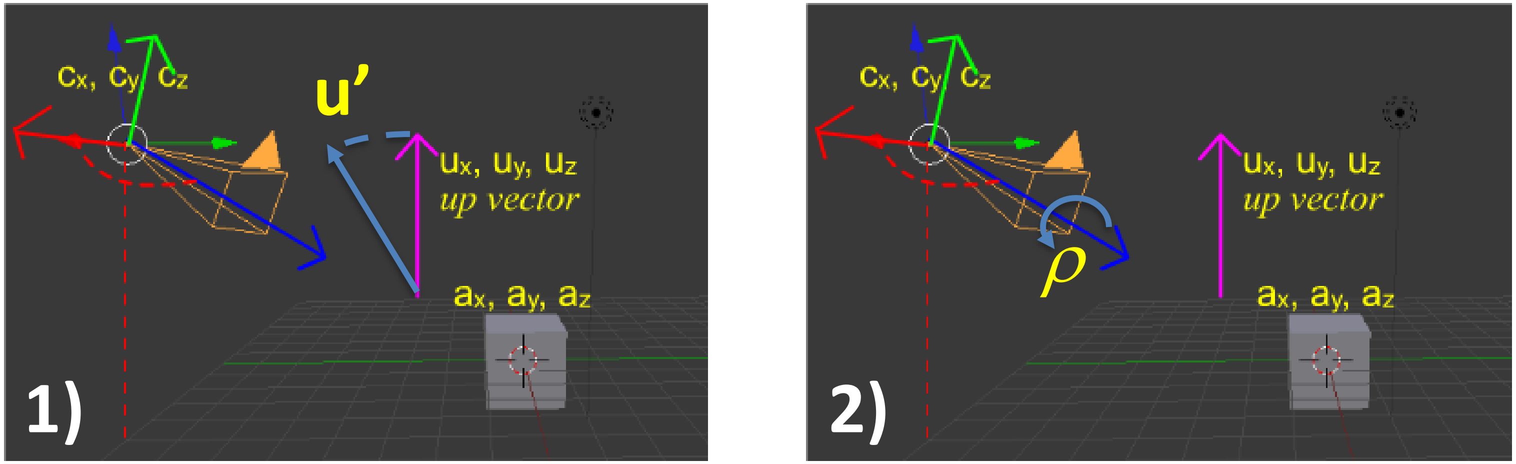
Look at matrix in GLM

Roll in Look At matrices can be implemented in two ways:

1. Rotating the **u** vector
2. Adding a rotation of an angle ρ around the z-axis.

In the last case, it can be implemented as follows:

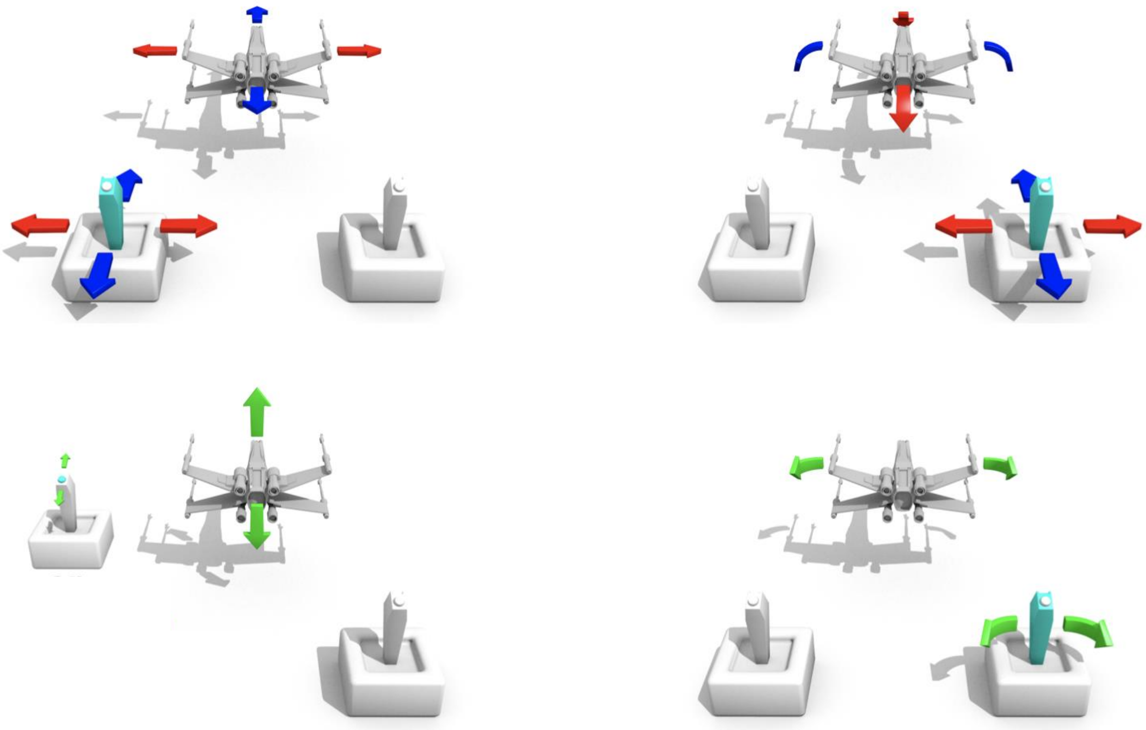
```
glm::mat4 Mv = glm::rotate(glm::mat4(1.0), -Roll, glm::vec3(0,0,1)) *
               glm::lookAt(c, a, u);
```



Camera navigation models

The motion of the camera is characterized by six axes:

Three motion axes



Three rotation axes

Camera navigation models

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

- Walk
- Fly

As the name suggests, the first works best in environments where there is a reference "ground" and some gravity that anchors the user over it, while the other is better when the viewer can fly in an open space without specific reference points (i.e. controlling a starship in orbit).

Camera navigation models

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

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

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

Before presenting how to move the camera, let's see how we can get input from the user.

Controls

All controls – keyboard, joysticks, gamepads and mouse – should be wrapped to return values in the -1 and +1 range for each axis:

+1 : the direction along the considered axis is selected

0 : this axis is not being changed

-1 : the opposite direction along the target axis is selected.

Discrete sources such as *keyboards*, *buttons*, *hat-swtiches* or *DPads* return boolean values, that can be mapped exactly one of these three value per axis.

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

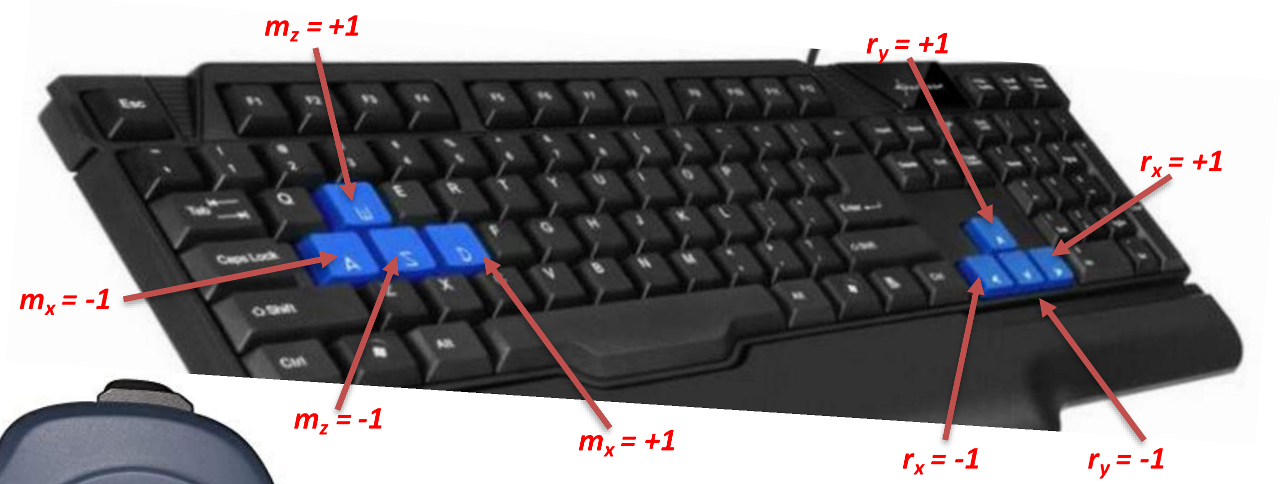
Controls

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

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

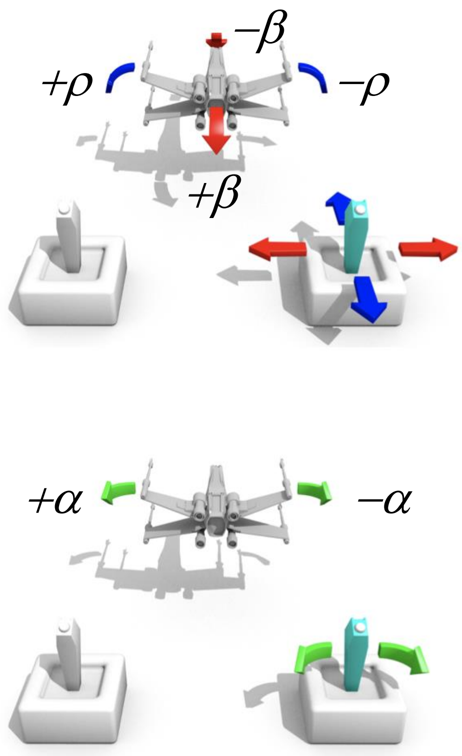
Controls

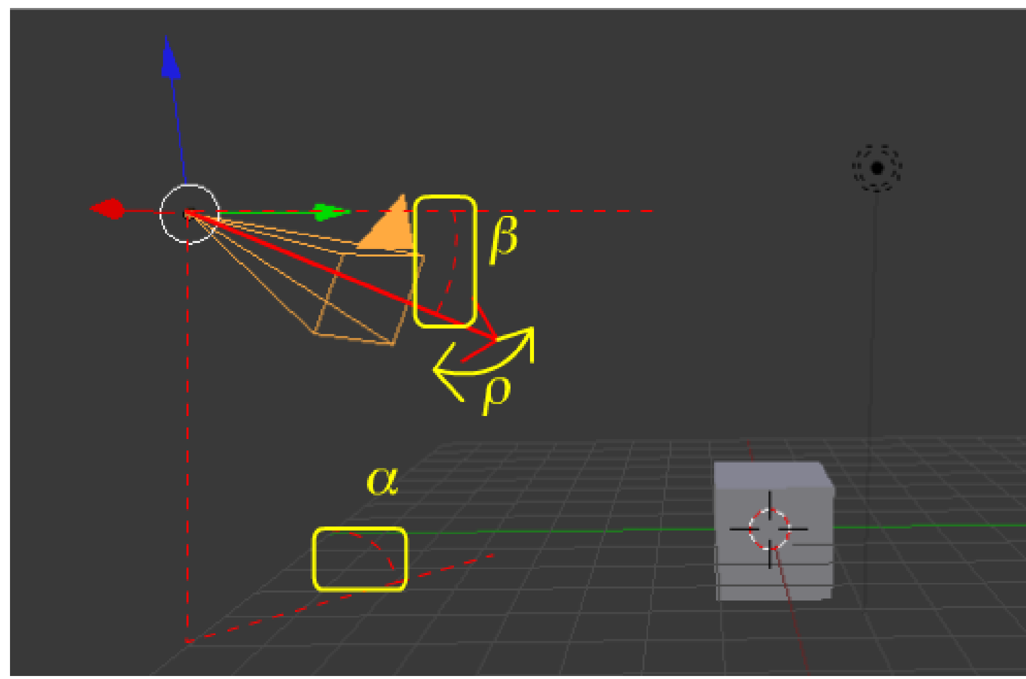
Examples:



The Walk navigation model

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





The Walk navigation model

In this case we usually use a vector variable to hold the position, plus three floating points variables to store the rotations.

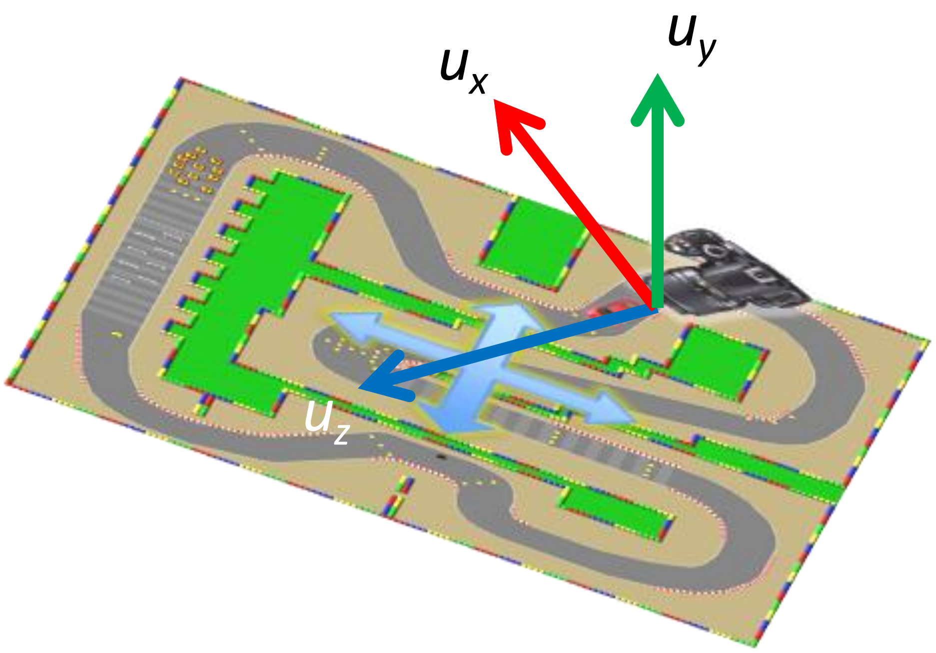
```
// external variables to hold
// the camera position
float alpha, beta, rho;
glm:vec3 pos;
```

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

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

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

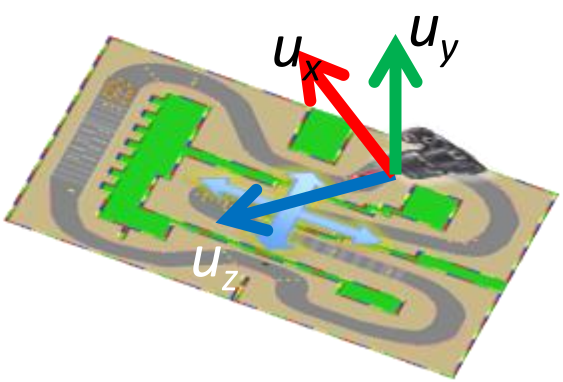
$$u_z = \left[R_y(\alpha) \cdot |0 \quad 0 \quad -1 \quad 1|\right].xyz$$

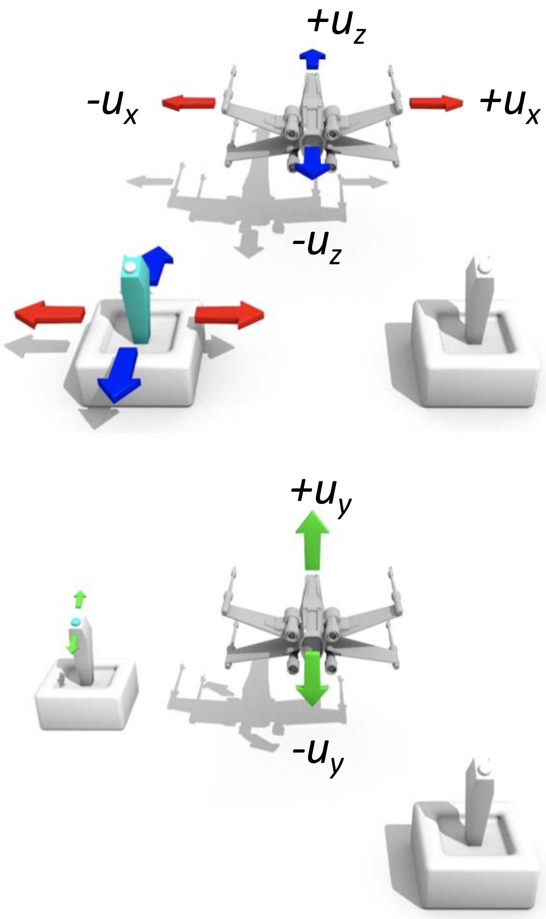


Here the *[…].xyz* notation is used to denote the cartesian coordinate corresponding the the homogeneous one.

The Walk navigation model

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





Speed

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

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

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

Update

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

```
// external variables to hold
// the camera position
float alpha, beta, rho;
glm:vec3 pos;

...

// The Walk model update procedure
glm::mat4 ViewMatrix;
glm::vec3 ux = glm::vec3(glm::rotate(glm::mat4(1),
                         alpha, glm::vec3(0,1,0)) *
                         glm::vec4(1,0,0,1));
glm::vec3 uy = glm::vec3(0,1,0);
glm::vec3 uz = glm::vec3(glm::rotate(glm::mat4(1),
                         alpha, glm::vec3(0,1,0)) *
                         glm::vec4(0,0,-1,1));
alpha += omega * rx * dt;
beta  += omega * ry * dt;
rho   += omega * rz * dt;
pos += ux * mu * mx * dt;
pos += uy * mu * my * dt;
pos += uz * mu * mz * dt;

ViewMatrix = MakeLookAt(pos,
                          alpha, beta, rho);
```

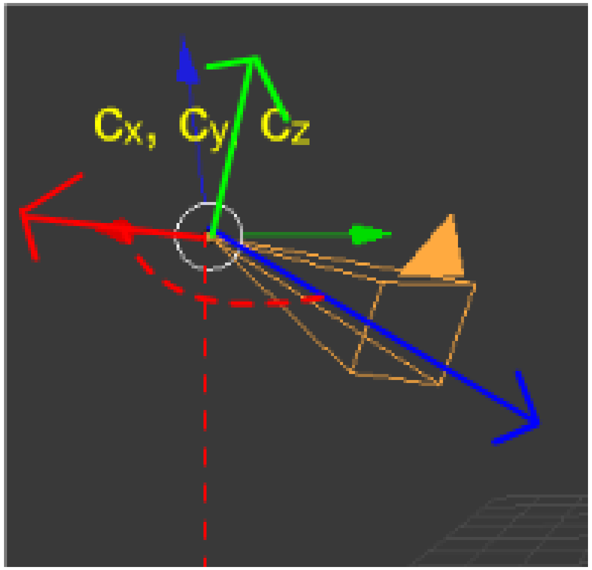
In the Walk model, it is easier to have variables containing the position and direction of the camera, and use them to recreate a new view matrix at each frame update.

The Fly navigation model

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

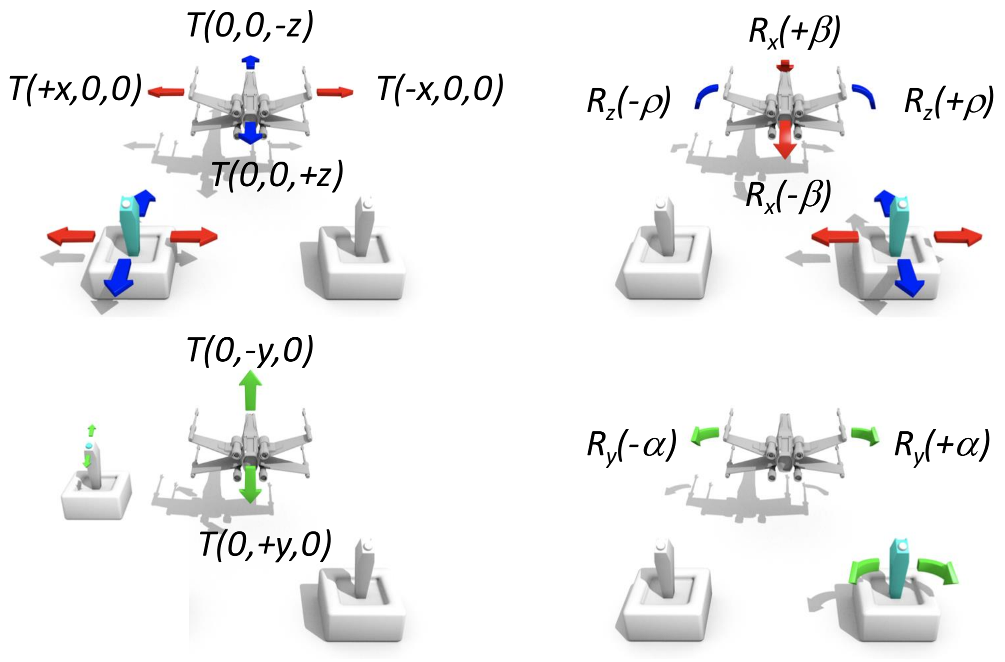
Since the view matrix brings the three axes of the camera along the *x*, *y* and *z-axes*, and its center to the origin, things are much simpler. We store only the view matrix and we update it directly.

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



The Fly navigation model

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



Update

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

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

...

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

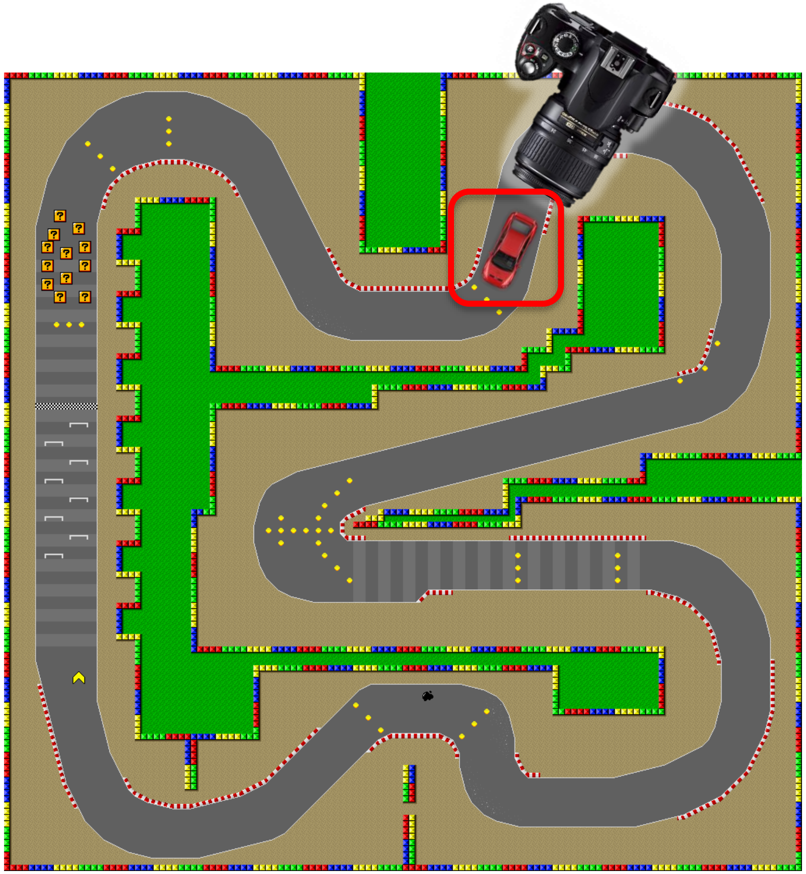
Please note that (as introduced in the beginning of the course) since matrix product is not commutative, the order of transformations matters. However, in this particular case, it usually does not have a visible impact because:

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

Third person view

In the next lesson, we will see how to move objects in the 3D world.

Third person view camera motion techniques, applies the procedures that we have just seen to the target object, instead of the camera.

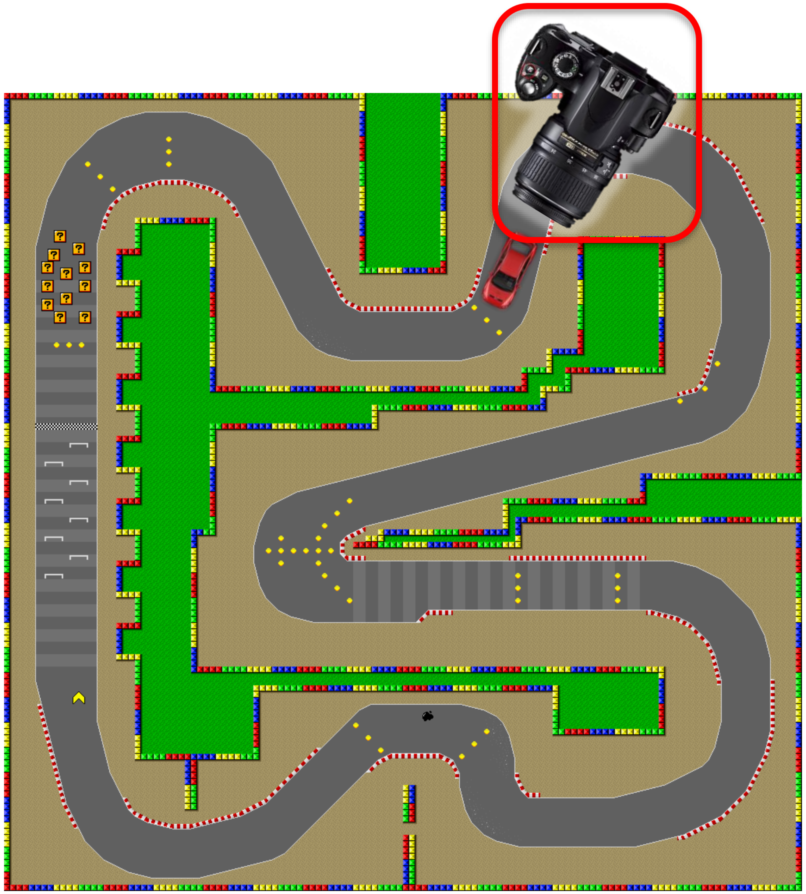


Third person view

Then, using appropriate algorithms, they determine the point of the camera that follows the object.

This algorithm is quite complex, since it needs to find a proper “spot” from which the target is visible (i.e. avoiding floor, ceilings and walls), and it can be influenced by the pitch and the roll of the camera.

Finally, the *LookAt* technique is used to determine the camera matrix.



Interaction with the Host O.S.

In order to perform motion, the input from controllers must be retrieved.

Each O.S. has its own way of getting the input from the devices connected to its host: allowing interaction in a platform independent way can be a very complex task.

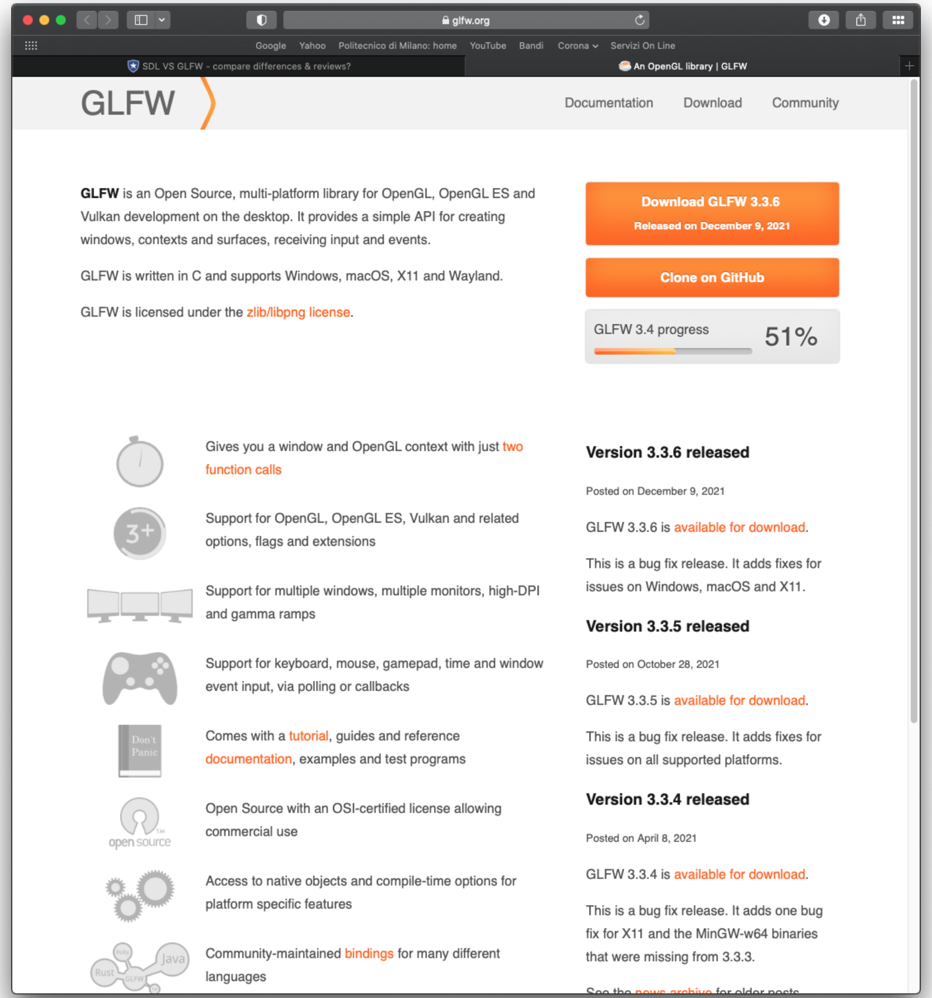
Several libraries have been developed for this task: GLFW and SDL are the two most popular for desktop applications.

In this course we will focus on GLFW.

GLFW

GLFW is an Open Source, multi-platform library for OpenGL and Vulkan development. It provides an API for creating windows, receiving input and events.

GLFW is written in C and supports Windows, macOS and Linux.



GLFW

One of the main features of GLFW is the ability of opening windows and handling events such as resizing.

We will see how this can be done in a future lesson.

```
    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
        glfwSetWindowUserPointer(window, this);
        glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
    }

    static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {
        auto app = reinterpret_cast<Assignment0*>
                        (glfwGetWindowUserPointer(window));
        app->framebufferResized = true;
    }
```

GLFW

Whenever a window is created, a `GLFWwindow*` object is returned. Such object must be stored in a variable that will be used in subsequent calls to the library.

```
GLFWwindow* window;
```

```

    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
        glfwSetWindowUserPointer(window, this);
        glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
    }

    static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {
        auto app = reinterpret_cast<Assignment0*>
                        (glfwGetWindowUserPointer(window));
        app->framebufferResized = true;
    }

```

GLFW

Key presses can be detected using the `glfwGetKey(window, GLFW_KEY_xxx)` function.

It returns `true` if the requested key has been pressed since the last call to the same method.

```
if(glfwGetKey(window, GLFW_KEY_A)) {
    CamPos -= MOVE_SPEED * glm::vec3(CamDir[0]) * deltaT;
}
if(glfwGetKey(window, GLFW_KEY_D)) {
    CamPos += MOVE_SPEED * glm::vec3(CamDir[0]) * deltaT;
}
if(glfwGetKey(window, GLFW_KEY_S)) {
    CamPos += MOVE_SPEED * glm::vec3(CamDir[2]) * deltaT;
}
if(glfwGetKey(window, GLFW_KEY_W)) {
    CamPos -= MOVE_SPEED * glm::vec3(CamDir[2]) * deltaT;
}
if(glfwGetKey(window, GLFW_KEY_F)) {
    CamPos -= MOVE_SPEED * glm::vec3(CamDir[1]) * deltaT;
}
if(glfwGetKey(window, GLFW_KEY_R)) {
    CamPos += MOVE_SPEED * glm::vec3(CamDir[1]) * deltaT;
}
```

GLFW

Each key has a different name. A complete reference can be found here:

https://www.glfw.org/docs/3.3/group__keys.html

```
#define GLFW_KEY_UNKNOWN -1
#define GLFW_KEY_SPACE 32
#define GLFW_KEY_APOSTROPHE 39 /* ' */
#define GLFW_KEY_COMMA 44 /* , */
#define GLFW_KEY_MINUS 45 /* - */
#define GLFW_KEY_PERIOD 46 /* . */
#define GLFW_KEY_SLASH 47 /* / */
#define GLFW_KEY_0 48
#define GLFW_KEY_1 49
#define GLFW_KEY_2 50
#define GLFW_KEY_3 51
#define GLFW_KEY_4 52
#define GLFW_KEY_5 53
#define GLFW_KEY_6 54
#define GLFW_KEY_7 55
#define GLFW_KEY_8 56
#define GLFW_KEY_9 57
#define GLFW_KEY_SEMICOLON 59 /* ; */
#define GLFW_KEY_EQUAL 61 /* = */
#define GLFW_KEY_A 65
#define GLFW_KEY_B 66
#define GLFW_KEY_C 67
#define GLFW_KEY_D 68
#define GLFW_KEY_E 69
#define GLFW_KEY_F 70
#define GLFW_KEY_G 71
#define GLFW_KEY_H 72
#define GLFW_KEY_I 73
```

```
#define GLFW_KEY_I 73
#define GLFW_KEY_J 74
#define GLFW_KEY_K 75
#define GLFW_KEY_L 76
#define GLFW_KEY_M 77
#define GLFW_KEY_N 78
#define GLFW_KEY_O 79
#define GLFW_KEY_P 80
#define GLFW_KEY_Q 81
#define GLFW_KEY_R 82
#define GLFW_KEY_S 83
#define GLFW_KEY_T 84
#define GLFW_KEY_U 85
#define GLFW_KEY_V 86
#define GLFW_KEY_W 87
#define GLFW_KEY_X 88
#define GLFW_KEY_Y 89
#define GLFW_KEY_Z 90
#define GLFW_KEY_LEFT_BRACKET 91 /* [ */
#define GLFW_KEY_BACKSLASH 92 /* \ */
#define GLFW_KEY_RIGHT_BRACKET 93 /* ] */
#define GLFW_KEY_GRAVE_ACCENT 96 /* ` */
#define GLFW_KEY_WORLD_1 161 /* non-US #1 */
#define GLFW_KEY_WORLD_2 162 /* non-US #2 */
#define GLFW_KEY_ESCAPE 256
#define GLFW_KEY_ENTER 257
#define GLFW_KEY_TAB 258
#define GLFW_KEY_BACKSPACE 259
```

```
#define GLFW_KEY_INSERT 260
#define GLFW_KEY_DELETE 261
#define GLFW_KEY_RIGHT 262
#define GLFW_KEY_LEFT 263
#define GLFW_KEY_DOWN 264
#define GLFW_KEY_UP 265
#define GLFW_KEY_PAGE_UP 266
#define GLFW_KEY_PAGE_DOWN 267
#define GLFW_KEY_HOME 268
#define GLFW_KEY_END 269
#define GLFW_KEY_CAPS_LOCK 280
#define GLFW_KEY_SCROLL_LOCK 281
#define GLFW_KEY_NUM_LOCK 282
#define GLFW_KEY_PRINT_SCREEN 283
#define GLFW_KEY_PAUSE 284
#define GLFW_KEY_F1 290
#define GLFW_KEY_F2 291
#define GLFW_KEY_F3 292
#define GLFW_KEY_F4 293
#define GLFW_KEY_F5 294
#define GLFW_KEY_F6 295
#define GLFW_KEY_F7 296
#define GLFW_KEY_F8 297
#define GLFW_KEY_F9 298
#define GLFW_KEY_F10 299
#define GLFW_KEY_F11 300
#define GLFW_KEY_F12 301
#define GLFW_KEY_F13 302
```

```
#define GLFW_KEY_KP_0 320
#define GLFW_KEY_KP_1 321
#define GLFW_KEY_KP_2 322
#define GLFW_KEY_KP_3 323
#define GLFW_KEY_KP_4 324
#define GLFW_KEY_KP_5 325
#define GLFW_KEY_KP_6 326
#define GLFW_KEY_KP_7 327
#define GLFW_KEY_KP_8 328
#define GLFW_KEY_KP_9 329
#define GLFW_KEY_KP_DECIMAL 330
#define GLFW_KEY_KP_DIVIDE 331
#define GLFW_KEY_KP_MULTIPLY 332
#define GLFW_KEY_KP_SUBTRACT 333
#define GLFW_KEY_KP_ADD 334
#define GLFW_KEY_KP_ENTER 335
#define GLFW_KEY_KP_EQUAL 336
#define GLFW_KEY_LEFT_SHIFT 340
#define GLFW_KEY_LEFT_CONTROL 341
#define GLFW_KEY_LEFT_ALT 342
#define GLFW_KEY_LEFT_SUPER 343
#define GLFW_KEY_RIGHT_SHIFT 344
#define GLFW_KEY_RIGHT_CONTROL 345
#define GLFW_KEY_RIGHT_ALT 346
#define GLFW_KEY_RIGHT_SUPER 347
#define GLFW_KEY_MENU 348
#define GLFW_KEY_LAST GLFW_KEY_MENU
```

The current position of the mouse can be detected with the `glfwGetCursorPos(window, &x, &y)` function.

It requires a pointer to two double precision floating point variables where it stores the current position of the mouse in pixels.

```
static double old_xpos = 0, old_ypos = 0;
double xpos, ypos;
glfwGetCursorPos(window, &xpos, &ypos);
double m_dx = xpos - old_xpos;
double m_dy = ypos - old_ypos;
old_xpos = xpos; old_ypos = ypos;

glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, GLFW_TRUE);
if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) {
    CamAng.y += m_dx * ROT_SPEED / MOUSE_RES;
    CamAng.x += m_dy * ROT_SPEED / MOUSE_RES;
}
```

GLFW

The pressure of a mouse key can be checked with the `glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_xx)` function, which returns `GLFW_PRESS` if the event occurred.

Each mouse button has a different name (for a complete list, see https://www.glfw.org/docs/3.3/group__buttons.html).

```
static double old_xpos = 0, old_ypos = 0;
double xpos, ypos;
glfwGetCursorPos(window, &xpos, &ypos);
double m_dx = xpos - old_xpos;
double m_dy = ypos - old_ypos;
old_xpos = xpos; old_ypos = ypos;

glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, GLFW_TRUE);
if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) {
    CamAng.y += m_dx * ROT_SPEED / MOUSE_RES;
    CamAng.x += m_dy * ROT_SPEED / MOUSE_RES;
}
```

GLFW

To convert the mouse motion into two axis values (i.e. in the [-1..+1] range), a simple difference with the previous location, divided by a larger value representing the movement resolution, can be implemented. This should depend on the size of the window, but in simpler applications can be a constant.

Static variables allow to remember the previous mouse position each time the procedure is executed.

```
static double old_xpos = 0, old_ypos = 0;
double xpos, ypos;
glfwGetCursorPos(window, &xpos, &ypos);
double m_dx = xpos - old_xpos;
double m_dy = ypos - old_ypos;
old_xpos = xpos; old_ypos = ypos;

glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, GLFW_TRUE);
if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) {
    CamAng.y += m_dx * ROT_SPEED / MOUSE_RES;
    CamAng.x += m_dy * ROT_SPEED / MOUSE_RES;
}
```

GLFW

The library supports also features to read joystick and gamepad controls, in an (almost) device independent way.

The procedures are however a bit complex, and outside the scope of this course.

If you are interested, a nice description can be found here:

Joystick input

The joystick functions expose connected joysticks and controllers, with both referred to as joysticks. It supports up to sixteen joysticks, ranging from `GLFW_JOYSTICK_1`, `GLFW_JOYSTICK_2` up to and including `GLFW_JOYSTICK_16` or `GLFW_JOYSTICK_LAST`. You can test whether a **joystick** is present with **glfwJoystickPresent**.

```
int present = glfwJoystickPresent(GLFW_JOYSTICK_1);
```

Each joystick has zero or more axes, zero or more buttons, zero or more hats, a human-readable name, a user pointer and an SDL compatible GUID.

When GLFW is initialized, detected joysticks are added to the beginning of the array. Once a joystick is detected, it keeps its assigned ID until it is disconnected or the library is terminated, so as joysticks are connected and disconnected, there may appear gaps in the IDs.

Joystick axis, button and hat state is updated when polled and does not require a window to be created or events to be processed. However, if you want joystick connection and disconnection events reliably delivered to the **joystick callback** then you must **process events**.

To see all the properties of all connected joysticks in real-time, run the `joysticks` test program.

Joystick axis states

The positions of all axes of a joystick are returned by **glfwGetJoystickAxes**. See the reference documentation for the lifetime of the returned array.

https://www.glfw.org/docs/3.3/input_guide.html#joystick

Other useful S.O. calls

As introduced, it is important to know the time passed since the last call to the procedure for performing a platform independent motion update.

Although GLFW has functions for accessing the system clock, C++ has a standard interface called `<chrono>` that can be used to read the current time in high resolution.

```
#include <chrono>
```

```
static auto startTime = std::chrono::high_resolution_clock::now();
static float lastTime = 0.0f;

auto currentTime = std::chrono::high_resolution_clock::now();
float time = std::chrono::duration<float, std::chrono::seconds::period>
            (currentTime - startTime).count();
float deltaT = time - lastTime;
lastTime = time;
```

Other useful S.O. calls

Similarly to what done for mouse motion, the time since the last call to the procedure (named `deltaT` below) measured in seconds can be computed memorizing the previous value (in a static variable) and computing the difference.

```
#include <chrono>
```

```
static auto startTime = std::chrono::high_resolution_clock::now();
static float lastTime = 0.0f;

auto currentTime = std::chrono::high_resolution_clock::now();
float time = std::chrono::duration<float, std::chrono::seconds::period>
            (currentTime - startTime).count();
float deltaT = time - lastTime;
lastTime = time;
```

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)