POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DEIB

2023

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2023

Computer Graphics

- Projections Wrap Up and Third Person Controller

Marco Gribaudo - DEIB

World-View-Projection Matrices

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

Each step performs a coordinate transformation from one space to another.

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

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

World-View-Projection Matrices

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

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

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

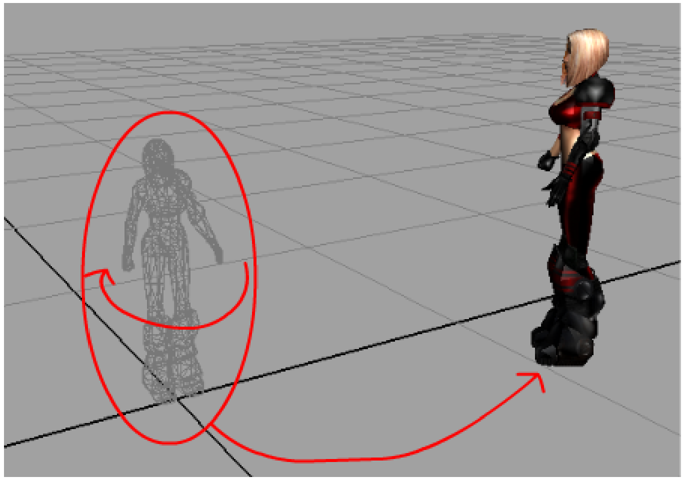
$$p_M = \begin{vmatrix} p_{Mx} & p_{My} & p_{Mz} \end{vmatrix}$$

Step I.a $\downarrow$

$$p_L = \begin{vmatrix} p_{Mx} & p_{My} & p_{Mz} & 1 \end{vmatrix}$$

Step I.b $\downarrow$

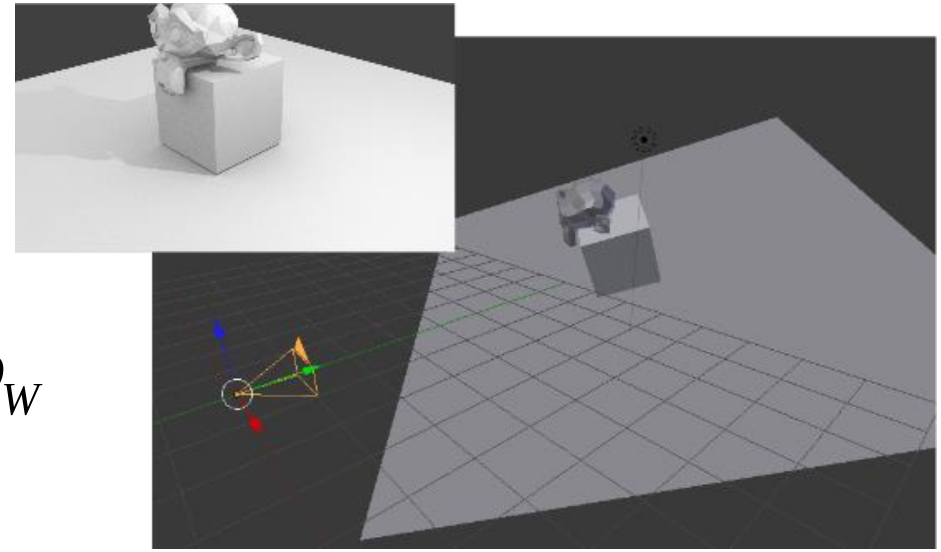
$$p_W = M_W \times p_L$$



World-View-Projection Matrices

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

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



Step II

$$p_V = M_V \times p_W$$

World-View-Projection Matrices

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

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

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

Step III

$$p_C = M_P \times p_V$$

The World-View-Projection Matrix

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

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

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

Step I-II-III

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

World-View-Projection Matrices

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

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

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

Step IV

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

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

World-View-Projection Matrices

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

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

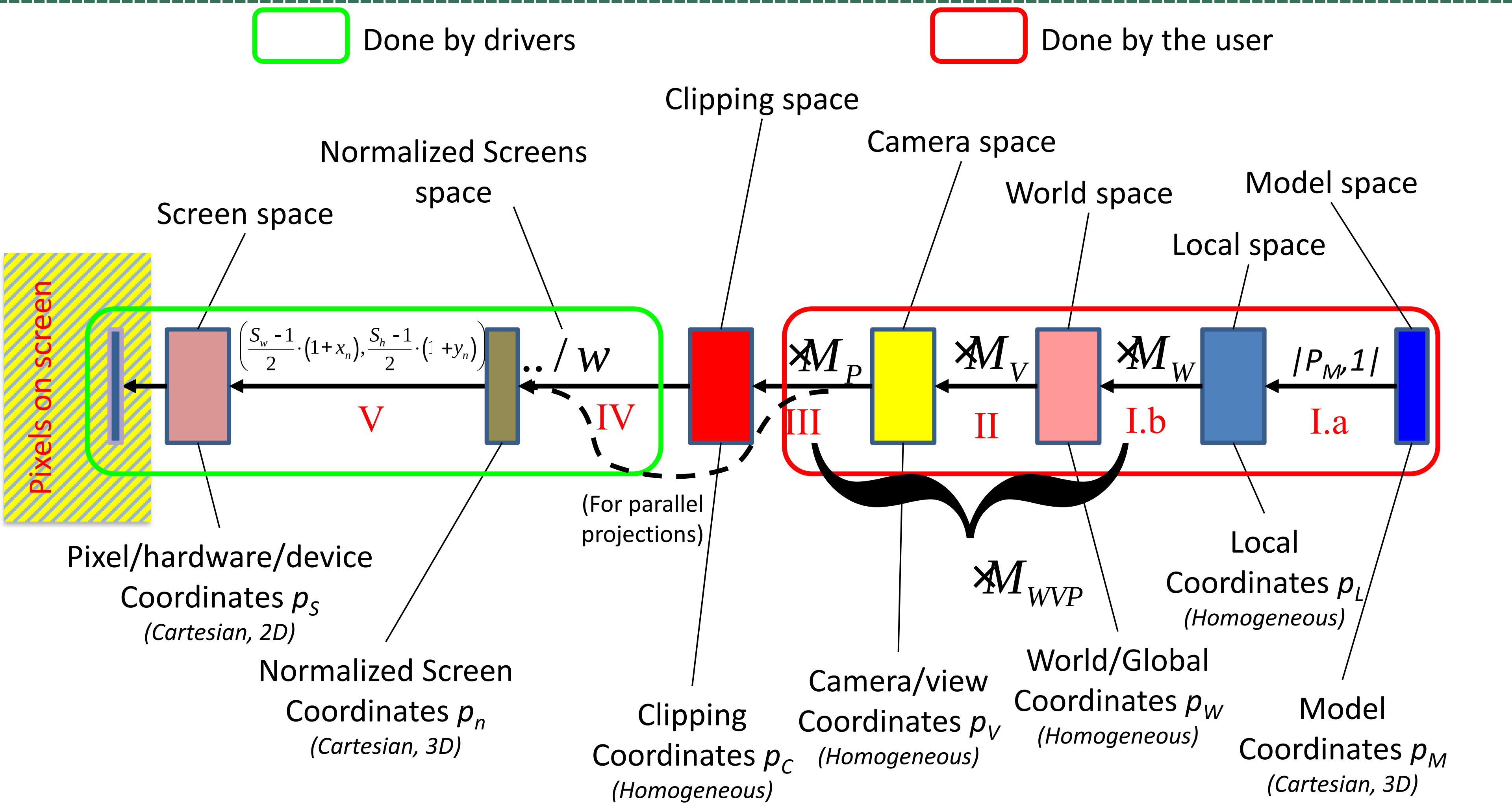
The driver of the video card converts the *clipping coordinates* first to *normalized screen coordinates* (if necessary), and then to *pixel coordinates* to visualize the objects (*Step V*). This is done in a way that is transparent to final user.

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

Step V $\downarrow$

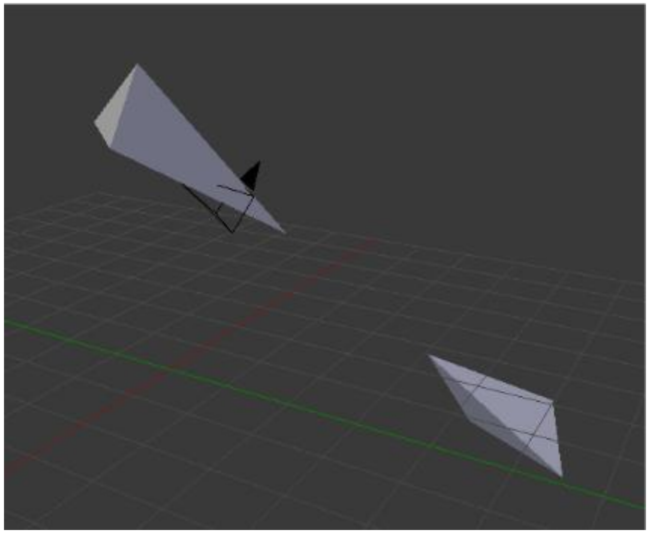
$$(x_S, y_S) = \left(\frac{S_w - 1}{2} \cdot (1 + x_n), \frac{S_h - 1}{2} \cdot (1 + y_n) \right)$$

World-View-Projection Matrices: summary



A complete projection example

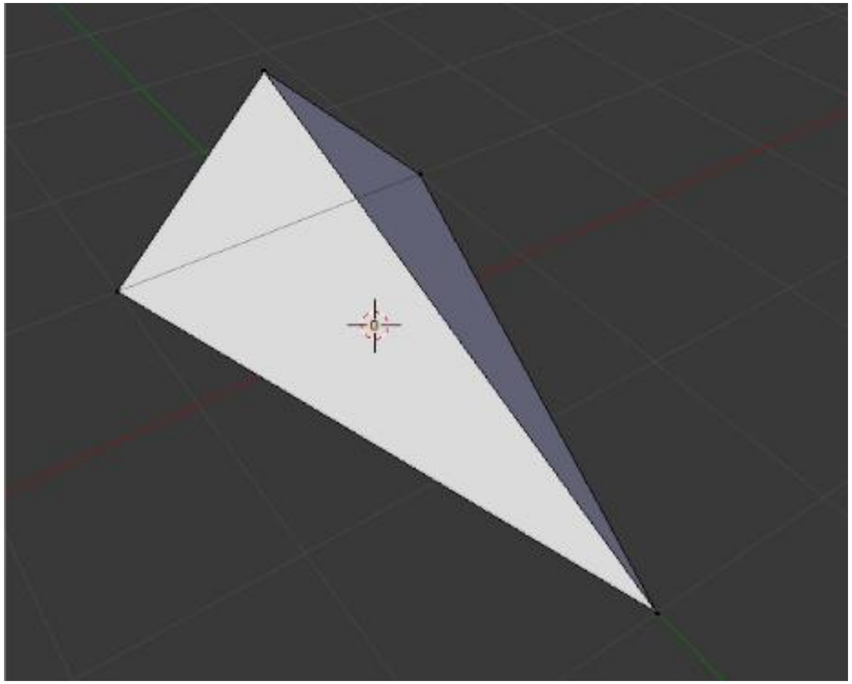
Let us suppose we want to create a simple first-person shooter game, where the user moves a starship.



In the figure, the player starship is the one on the left: however, since it is a first person view, it will never be shown in game, and only the associated camera will be used.

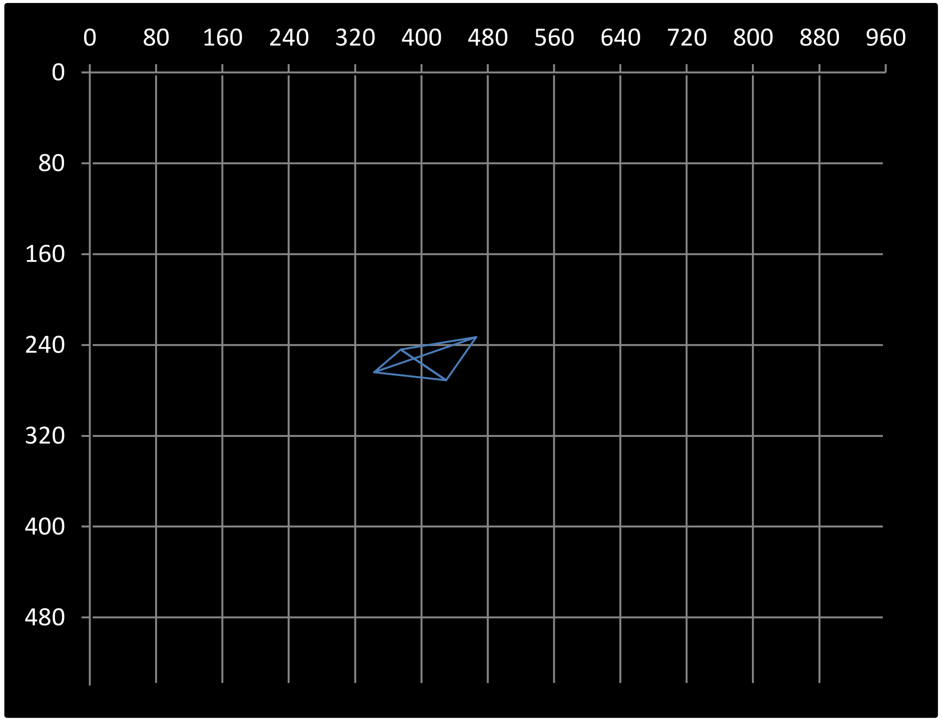
A complete projection example

The game is extremely simple: Starships are modeled with *tetrahedrons*.



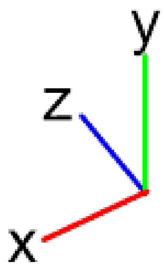
A complete projection example

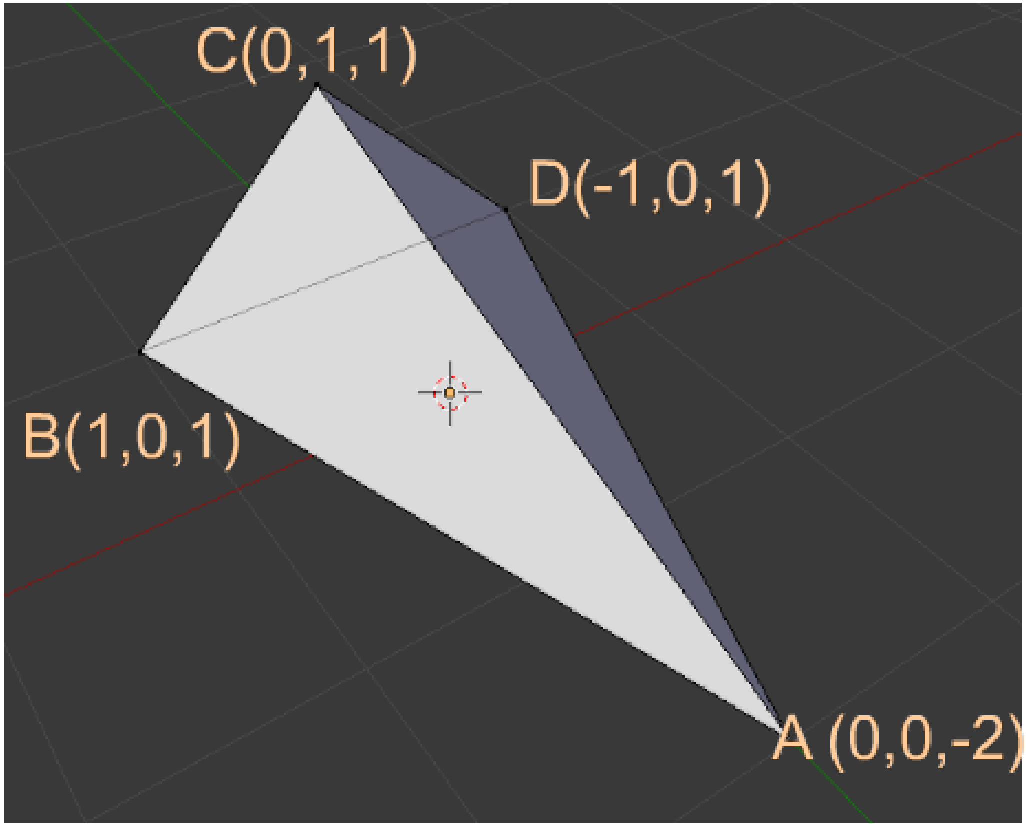
To further simplify the visualization, models are shown in wireframe mode (only their boundary).



A complete projection example

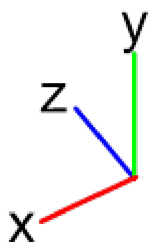
The following local coordinates characterize them.

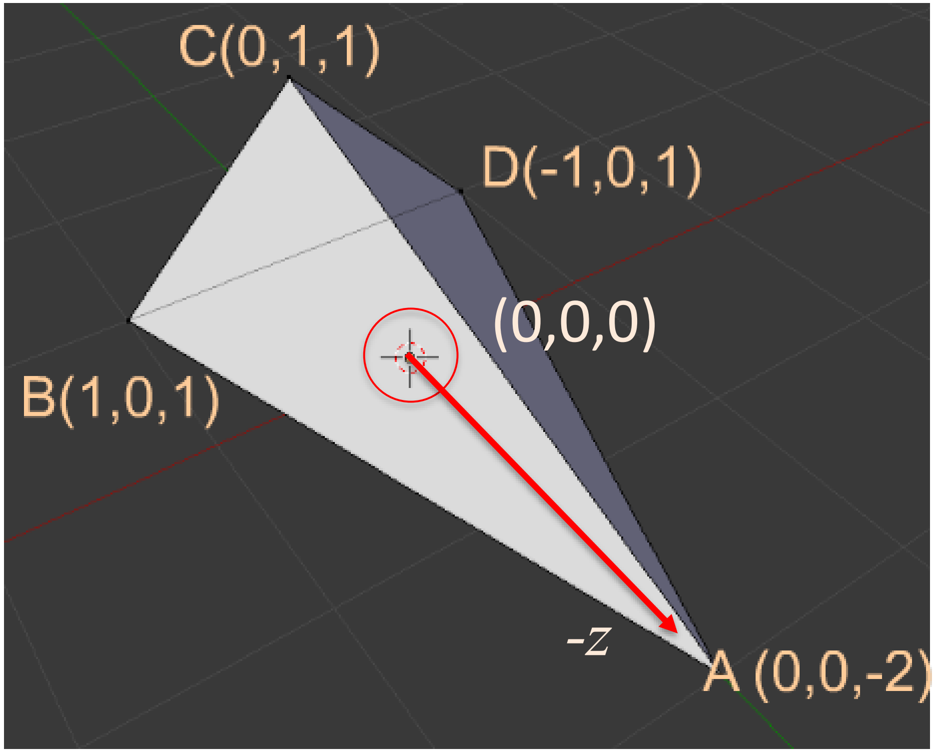




A complete projection example

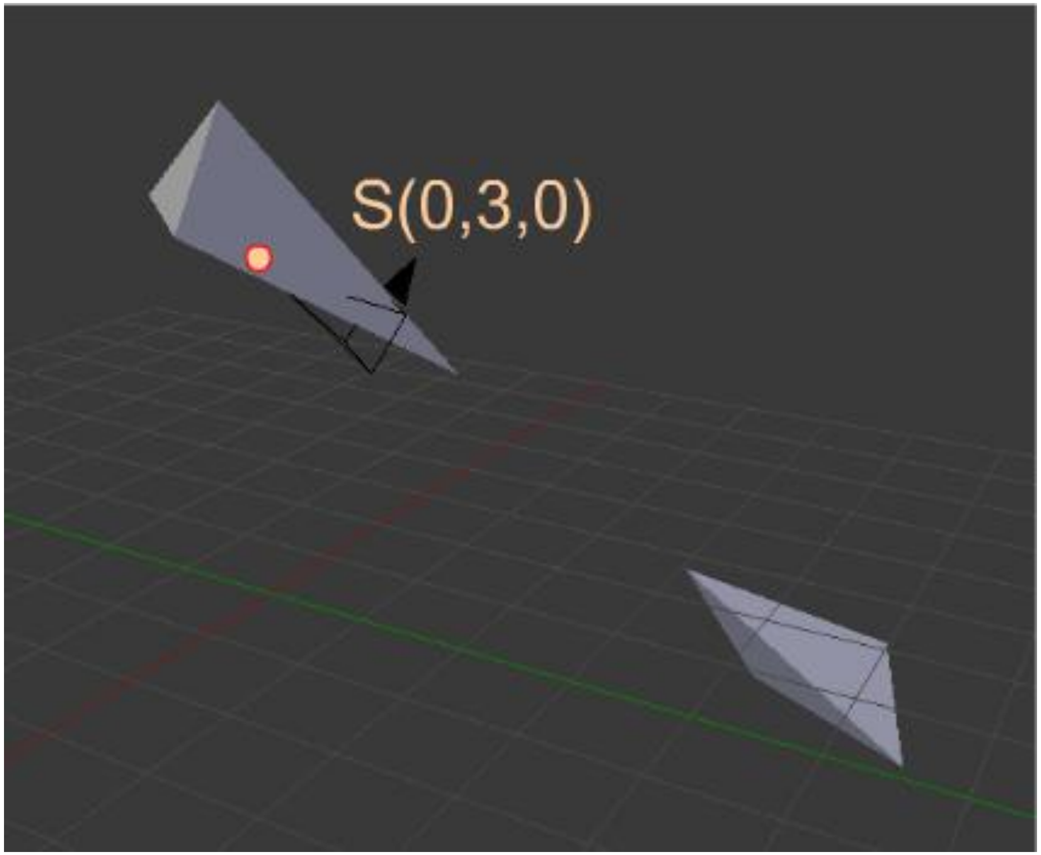
Note that the models have been created oriented along the negative z-axis, with their center in the origin of the local coordinates system.





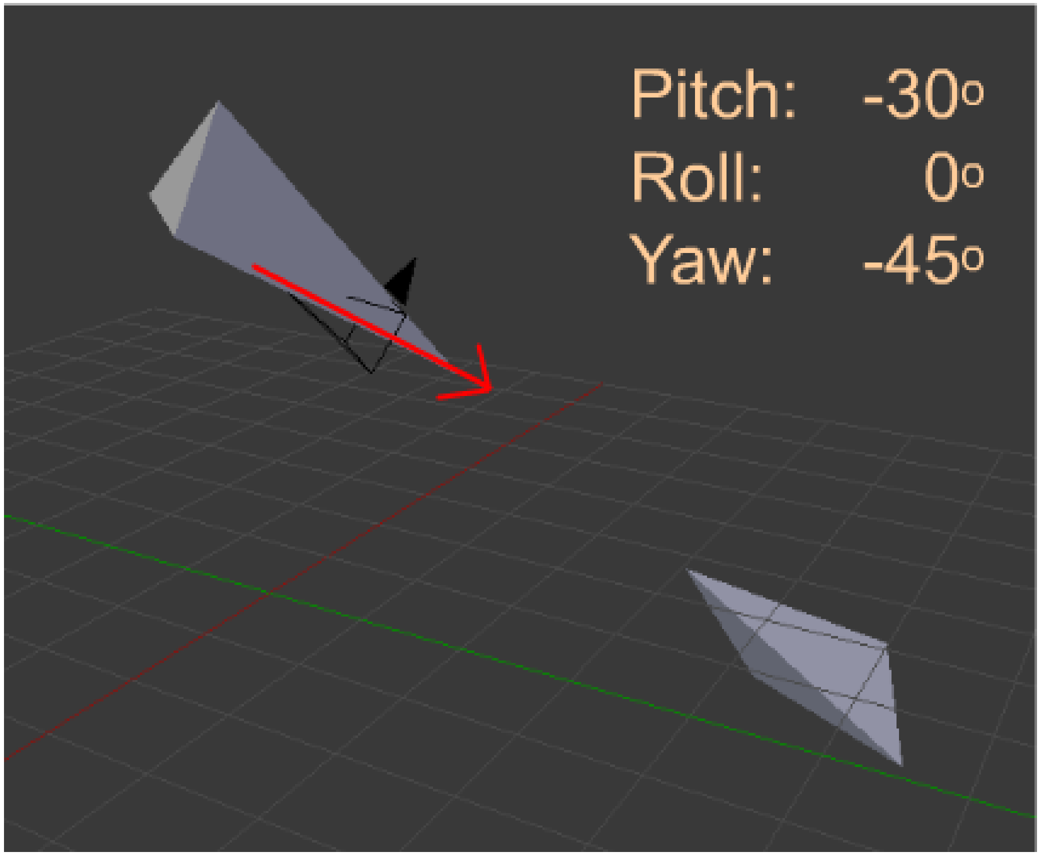
A complete projection example

In a moment of the game, the player is located at:



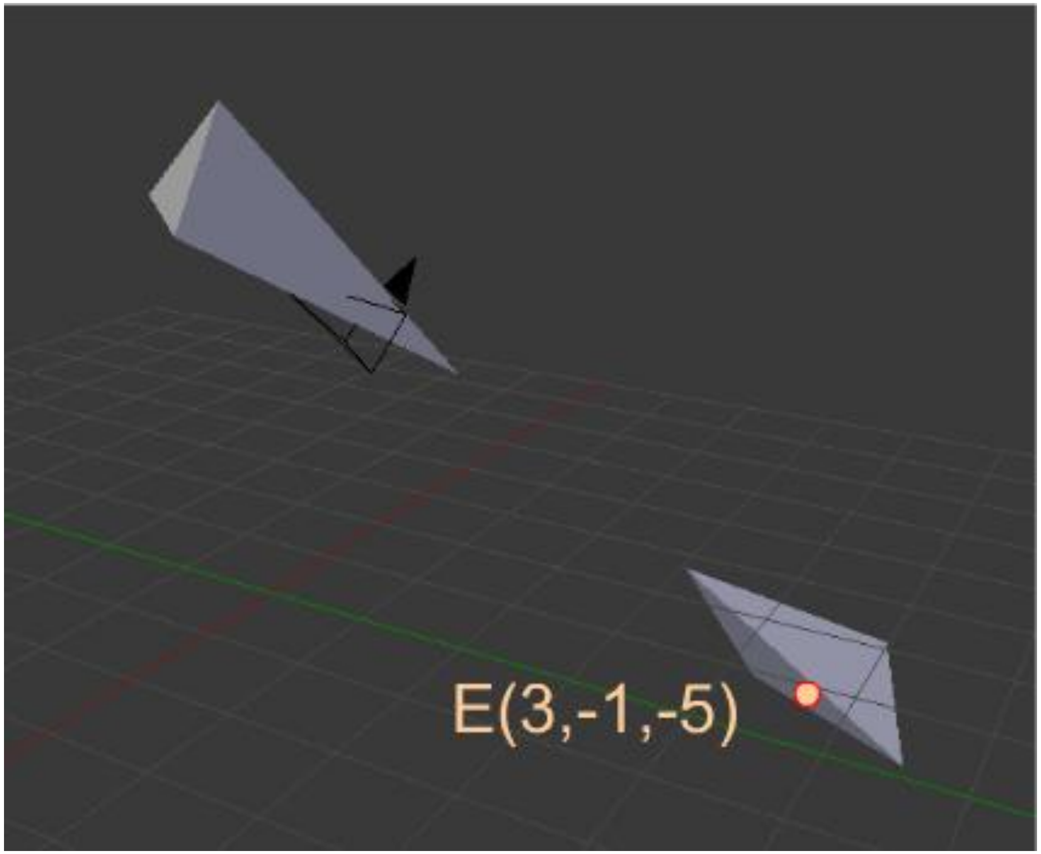
A complete projection example

And it is aiming in the following direction:



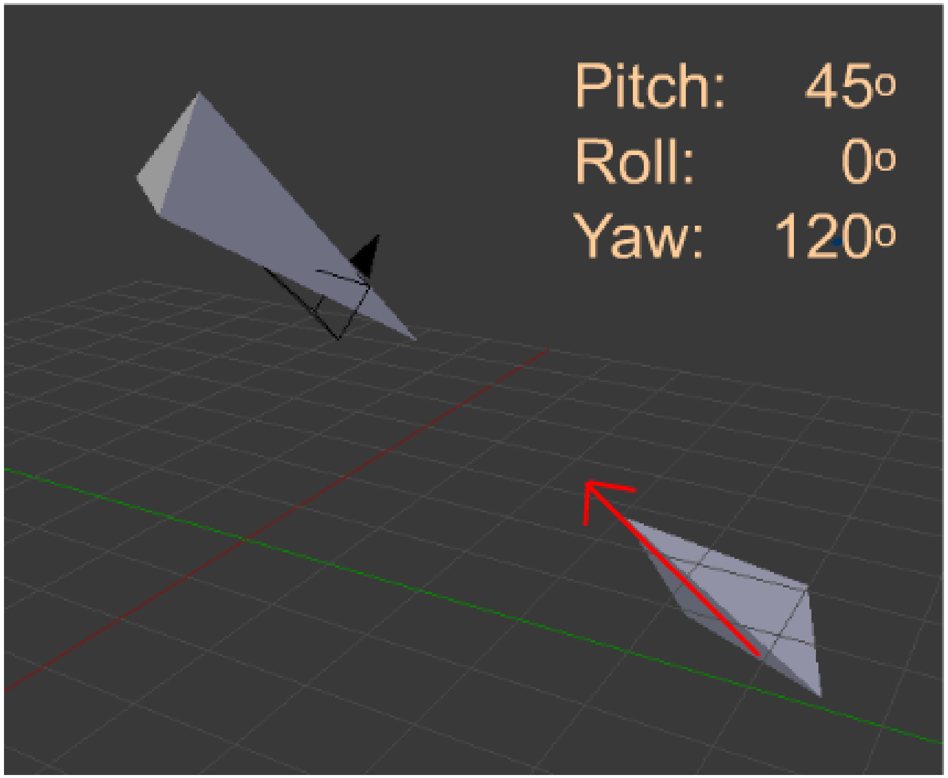
A complete projection example

The enemy fighter is located at:



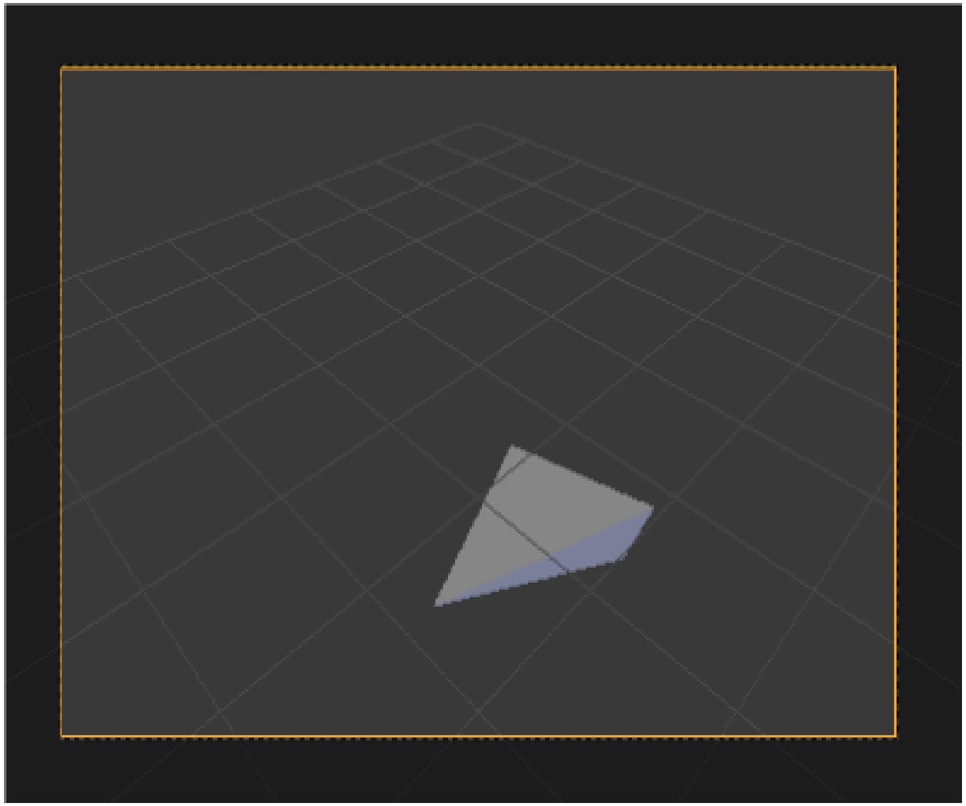
A complete projection example

And it is heading in the following direction



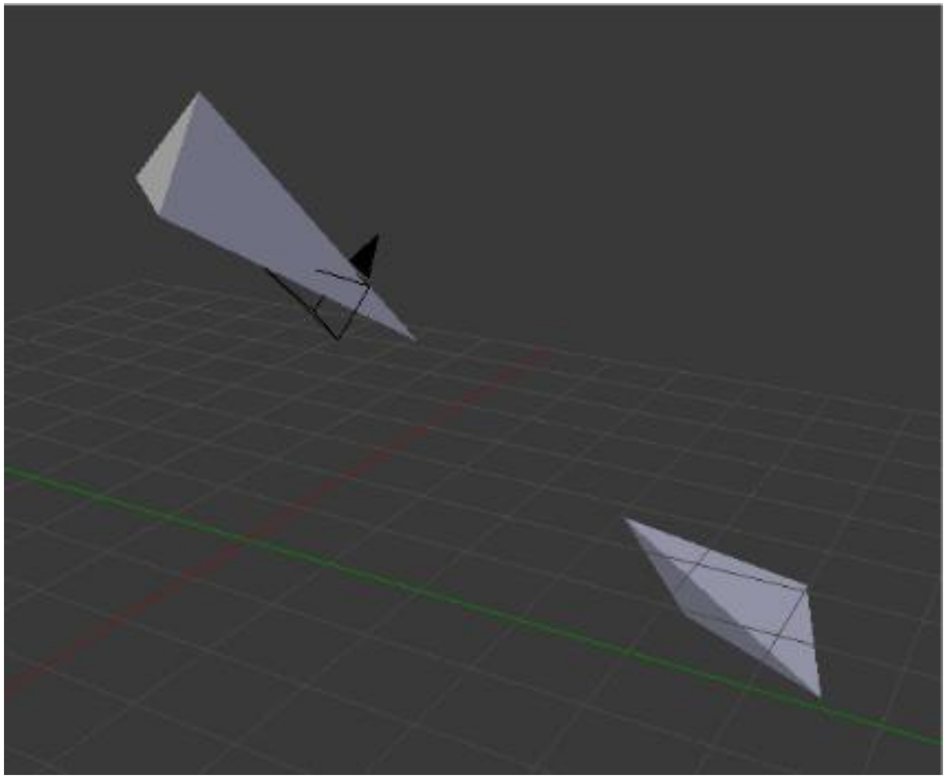
A complete projection example

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



A complete projection example

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



A complete projection example

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

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

T

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

Ry

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

Rx

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

Rz

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

S

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Step I

Mw

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

A complete projection example

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

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

Rz

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Rx

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

Ry

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

T

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

Step II

Mv

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

A complete projection example

We compute the projection matrix associated to the camera:

Perspective

FovY	a
90	1,25

n	f
0,5	9,5

Pp

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

Step III

A complete projection example

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

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

*

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

*

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

=

World-View-Projection

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

Step I-II-III

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

A complete projection example

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

Points

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

*

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

=

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

Step IV

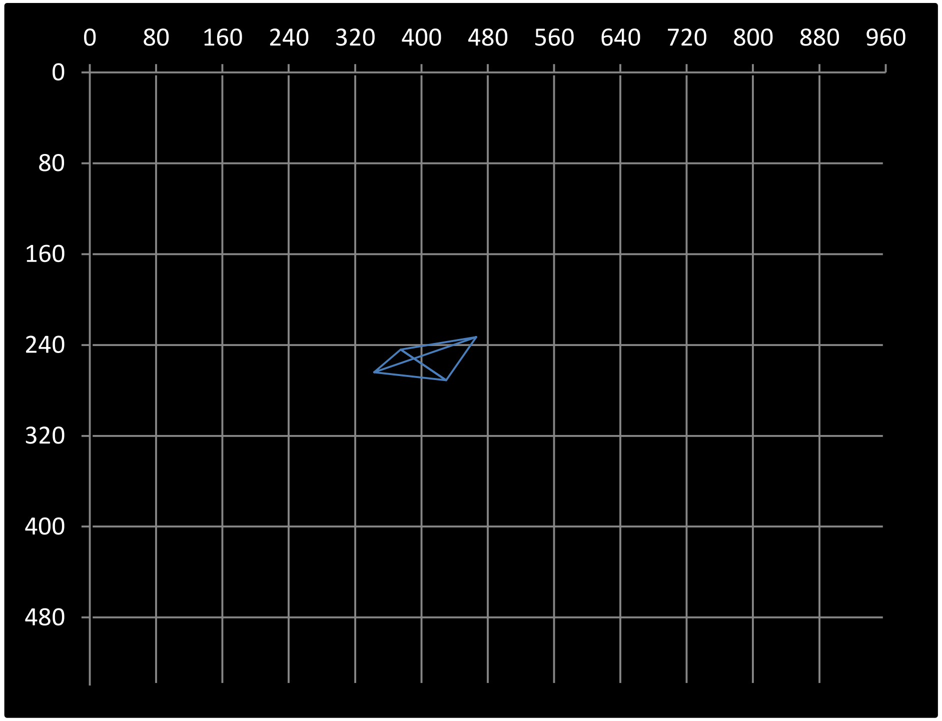
Normalized Screen Coordinates				
	-0,28	-0,22	-0,10	-0,03
	-0,02	-0,10	0,01	-0,14
	0,90	0,98	0,98	0,97

Pixel Coordinates				
	343	375	430	466
	264	244	271	233

Step V

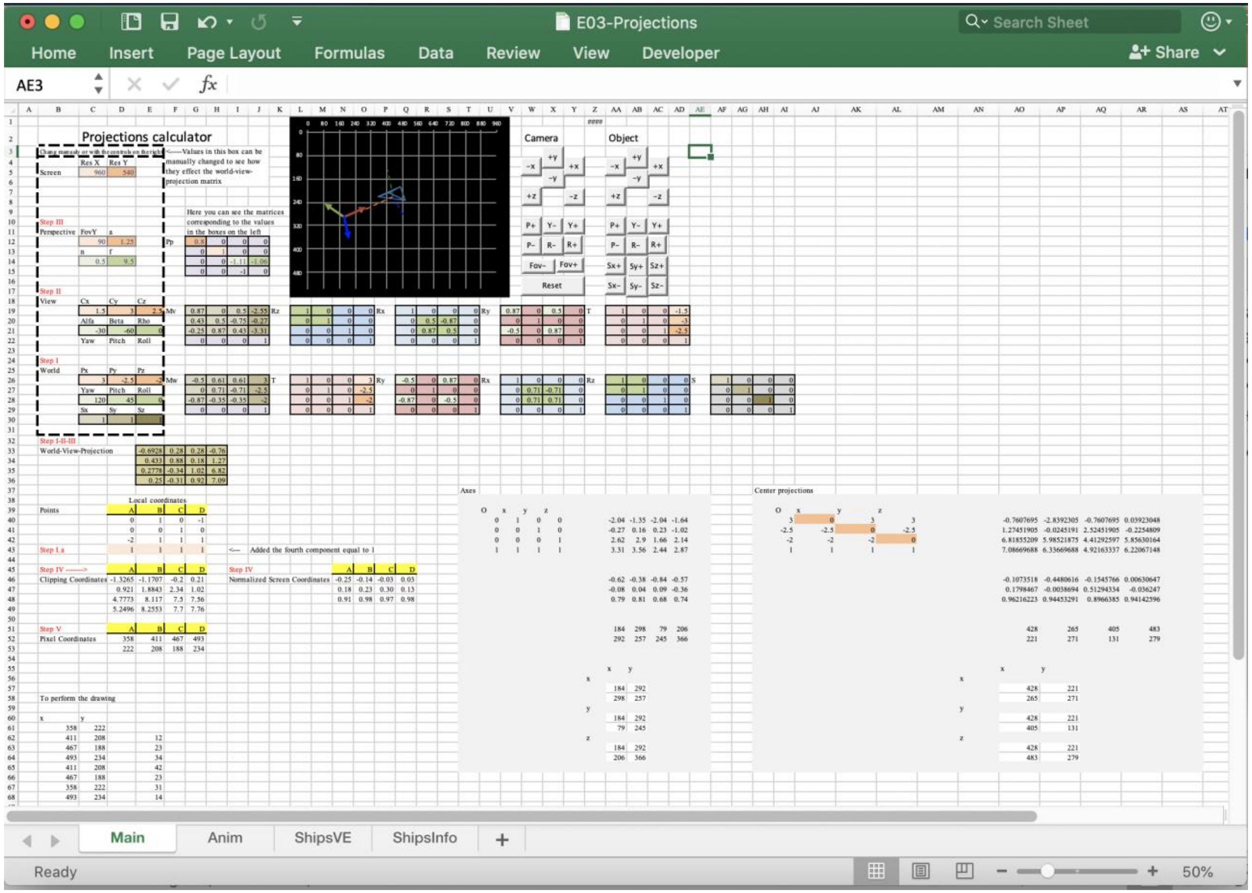
A complete projection example

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



A complete projection example

This set of computations can be performed in any numerical computation tool with graphing capabilities, even in something as standard as *Microsoft Excel*.



Note: the example shown here uses the OpenGL convention for the normalized screen coordinates. Using the Vulkan convention would produce slightly different but very similar results.

Moving assets

Moving an object in 3D space, is very similar to moving a camera.

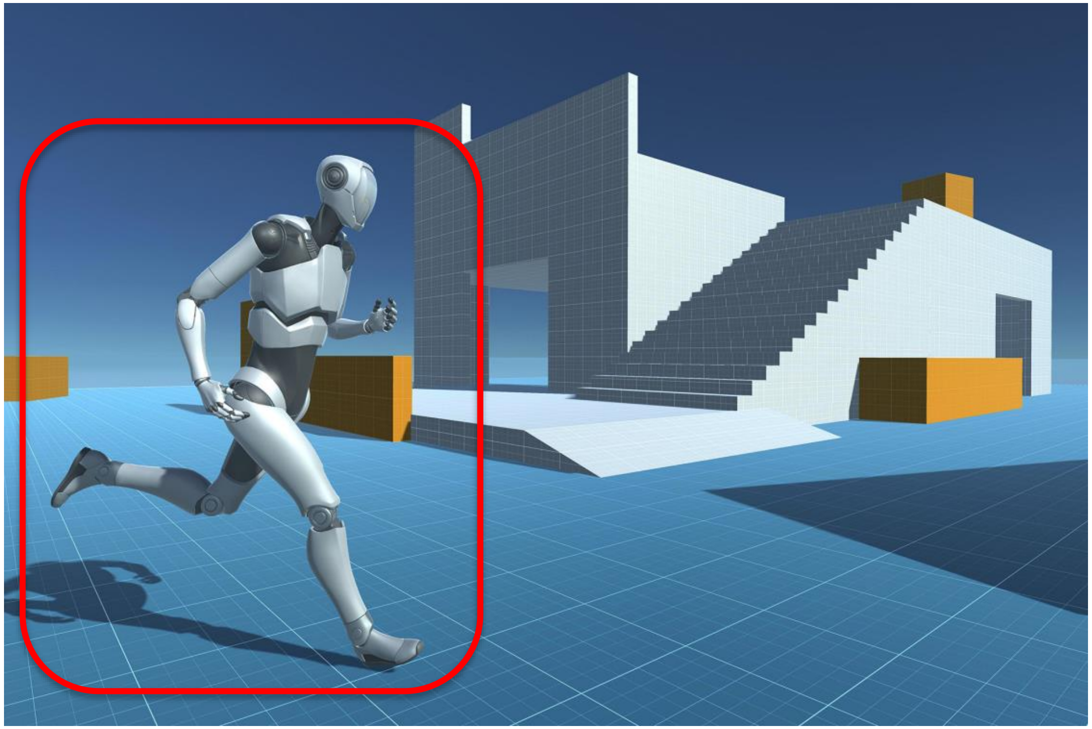
However, depending on the application, there are a much larger range of possibilities, which cannot be covered in a general way.

Here we will briefly outline three common motion techniques:

- Ground motion (similar to the Walk camera navigation model).
- World coordinates motion.
- Local coordinates motion (similar to the Fly camera navigation model).

Moving objects on a ground based scene

The *Ground* motion is usually used in third-person applications, to move the object that corresponds to the target of the camera.



Moving objects on a ground based scene

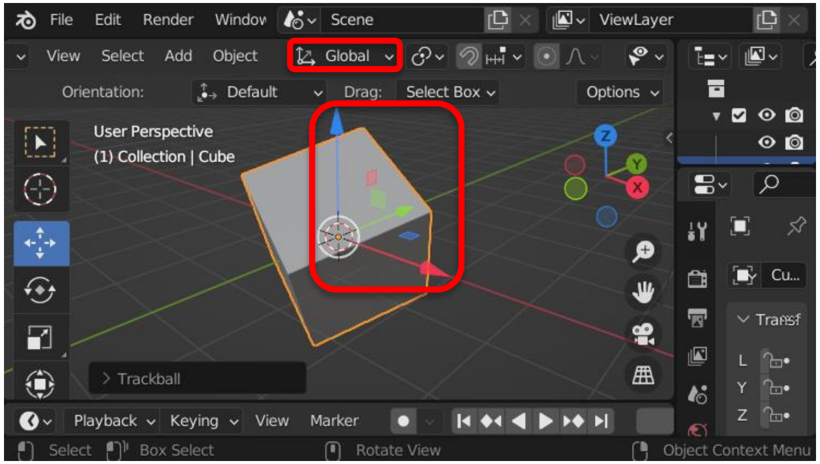
The *Ground* motion cycle is basically identical to the Walk procedure for a camera object: the position and the angles of the object are stored into four variables.

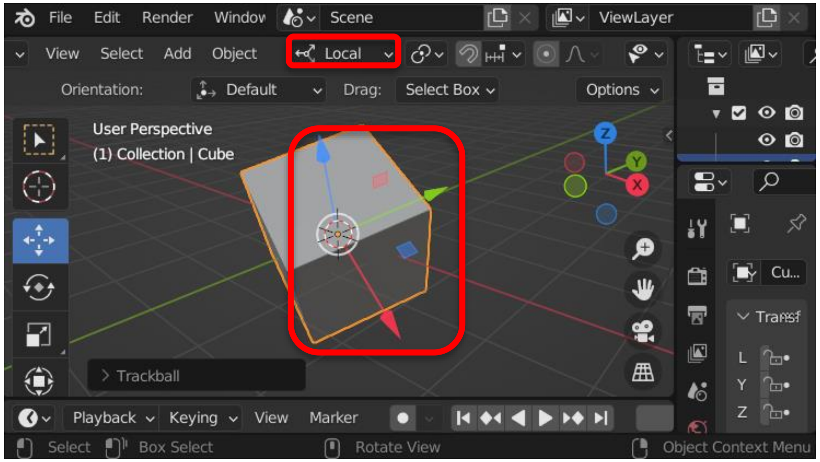
Moreover, in most cases, only the *yaw* angle is required, greatly simplifying the procedure.

```
// external variables to hold
// the object position
float yaw, pitch, roll;
glm:vec3 pos;

...

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

WorldMatrix = MakeWorldEuler(pos,
                          alpha, beta, rho);
```

Local and Global coordinates motion model

The *Local* and *Global* coordinates models are used to mimic the classical tools available to position objects in applications like *Blender*.





Global coordinates motion model

The Global Coordinates model differs from the Ground one because here motion directions are not affected by the orientation of the object, and are always aligned with the scene main axes.

```
// external variables to hold
// the object position
float yaw, pitch, roll;
glm:vec3 pos;

...

// The Walk model update procedure
glm::mat4 WorldMatrix;
glm::vec3 ux = glm::vec3(1,0,0);
glm::vec3 uy = glm::vec3(0,1,0);
glm::vec3 uz = glm::vec3(0,0,1);

pitch += omega * rx * dt;
yaw   += omega * ry * dt;
roll  += omega * rz * dt;
pos += ux * mu * mx * dt;
pos += uy * mu * my * dt;
pos += uz * mu * mz * dt;

WorldMatrix = MakeWorldEuler(pos,
                             alpha, beta, rho);
```

The local coordinates model

The update cycle for a motion in local coordinates is instead similar to the Fly camera model. It is also used for controlling for example space ships in third person applications:

```
// external variable to hold
// the world matrix
glm::mat4 WorldMatrix;

...

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

The local and global coordinates model – quaternion form

The orientation can generally be stored more efficiently using a quaternion. Here it is an example for the local coordinates case.

```
// external variable to hold
// the world matrix
glm:vec3 pos;
glm:quat rot;
...

// The local coordinates model update proc. With quaternions
glm::mat4 WorldMatrix;
rot = rot * glm::rotate(glm::quat(1,0,0,0), omega * rx * dt, glm::vec3(1, 0, 0));
rot = rot * glm::rotate(glm::quat(1,0,0,0), omega * ry * dt, glm::vec3(0, 1, 0));
rot = rot * glm::rotate(glm::quat(1,0,0,0), omega * rz * dt, glm::vec3(0, 0, 1));

glm::vec3 ux = glm::vec3(glm::mat4(rot) * glm::vec4(1,0,0,1));
glm::vec3 uy = glm::vec3(glm::mat4(rot) * glm::vec4(0,1,0,1));
glm::vec3 uz = glm::vec3(glm::mat4(rot) * glm::vec4(0,0,1,1));
pos += ux * mu * mx * dt;
pos += uy * mu * my * dt;
pos += uz * mu * mz * dt;

WorldMatrix = MakeWorldQuat(pos, rot);
```

In this case, since we do not have the entire matrix, the axes direction should be retrieved for updating the position of the object.

Camera position in third person applications

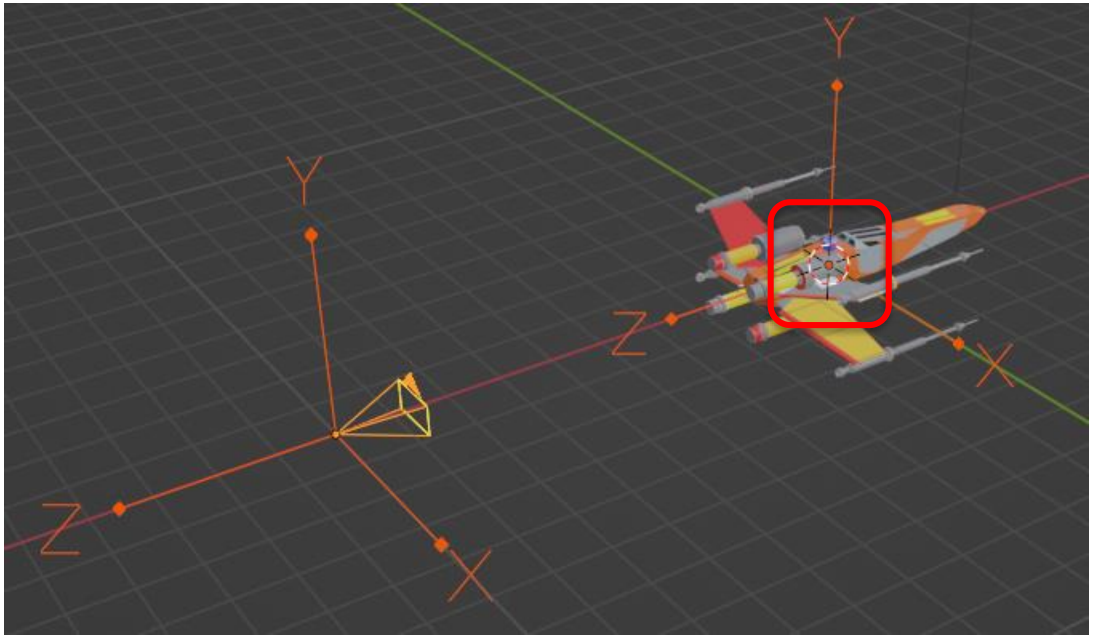
When creating a third person application, we have to determine both the position of the camera and its target. The latter is usually an object moved one of the techniques just introduced.

Lets focus on two popular cases:

- Flying a space ship using a “local” coordinates motion model
- Controlling a character using the “ground” based model

Flying a space ship in third person

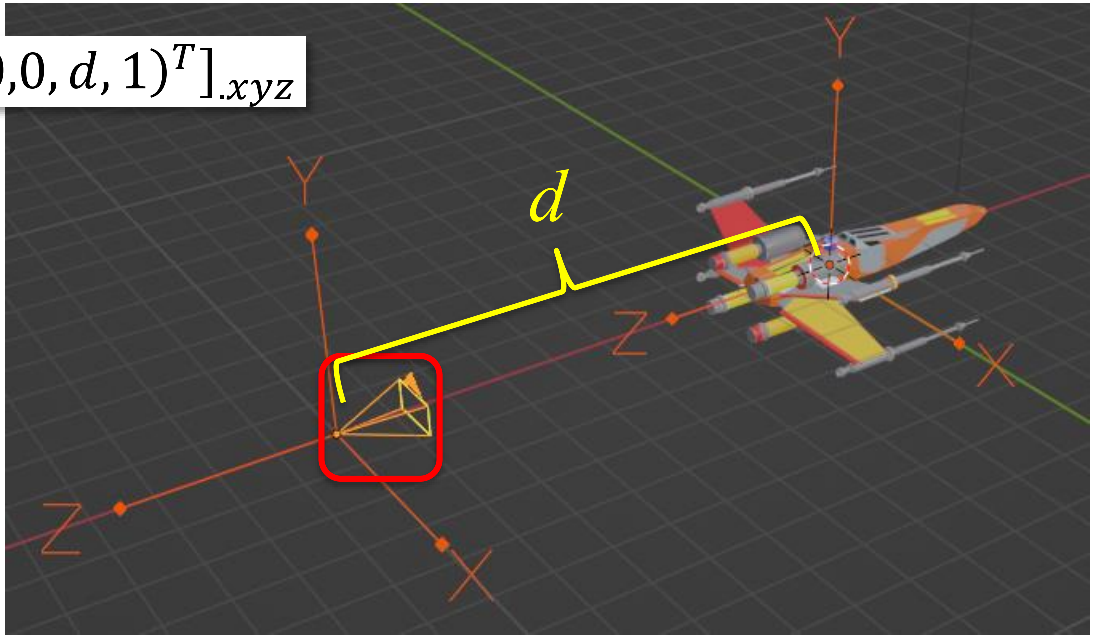
When flying a space ship in local coordinates model, the target generally corresponds to the position of the object.



Flying a space ship in third person

The camera is positioned at a constant (application dependent) distance *d* from the target. The camera center can then be computed applying the World Matrix of the ship to point *(0, 0, d, 1)*.

$$c = [M_W \cdot (0,0,d,1)^T]_{.xyz}$$

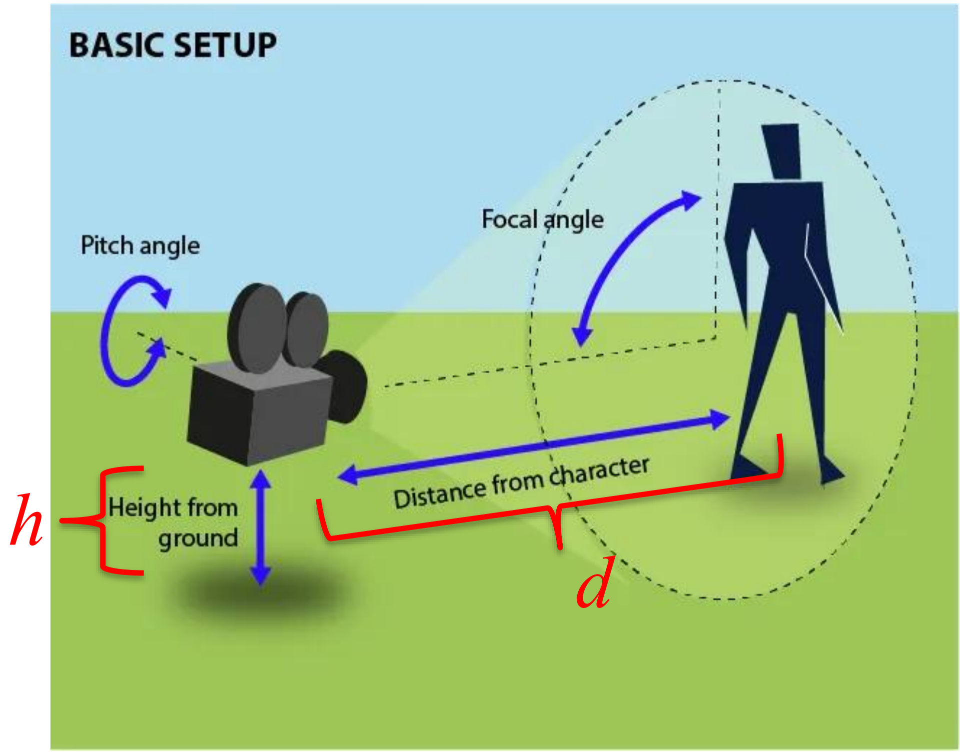


Moving a character in third person

When moving a character, we have to take into account that the target is generally different from the center of the object: for example, the origin is at the center of the feet, but the target is the head.

In the simplest scenario, we just store an “height” *h* for the target.

We also need a distance *d* of the target, as for the “flight” model.

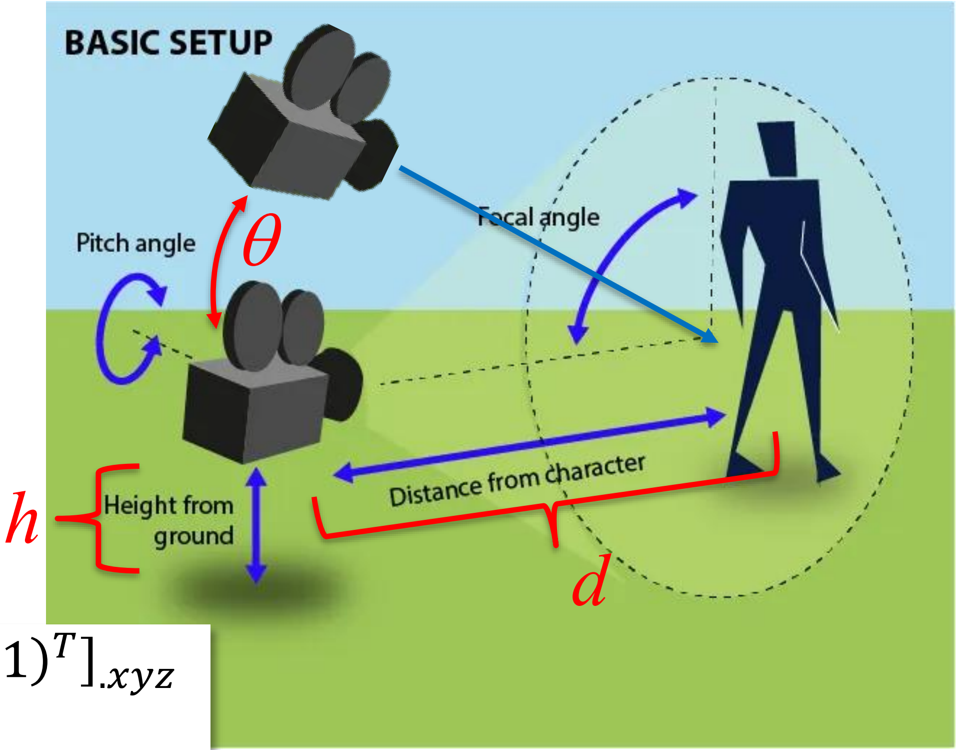


Moving a character in third person

In this scenario, the character usually moves only with the yaw.

Pitch can be implemented by rotating the target point of an angle θ.

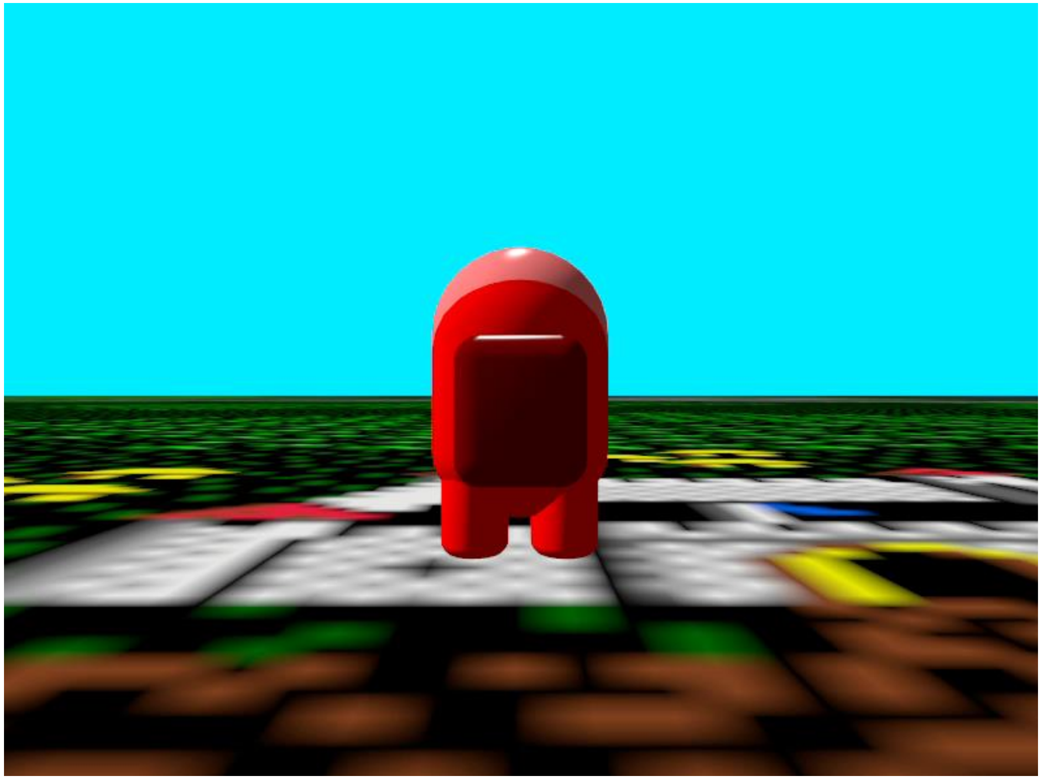
The camera *c* and target *a*, can then be defined in the following way:



$$c = [M_W \cdot (0, h + d \cdot \sin\theta, d \cdot \cos\theta, 1)^T]_{.xyz}$$
$$a = [M_W \cdot (0,0,0,1)^T]_{.xyz} + (0,0,h)$$

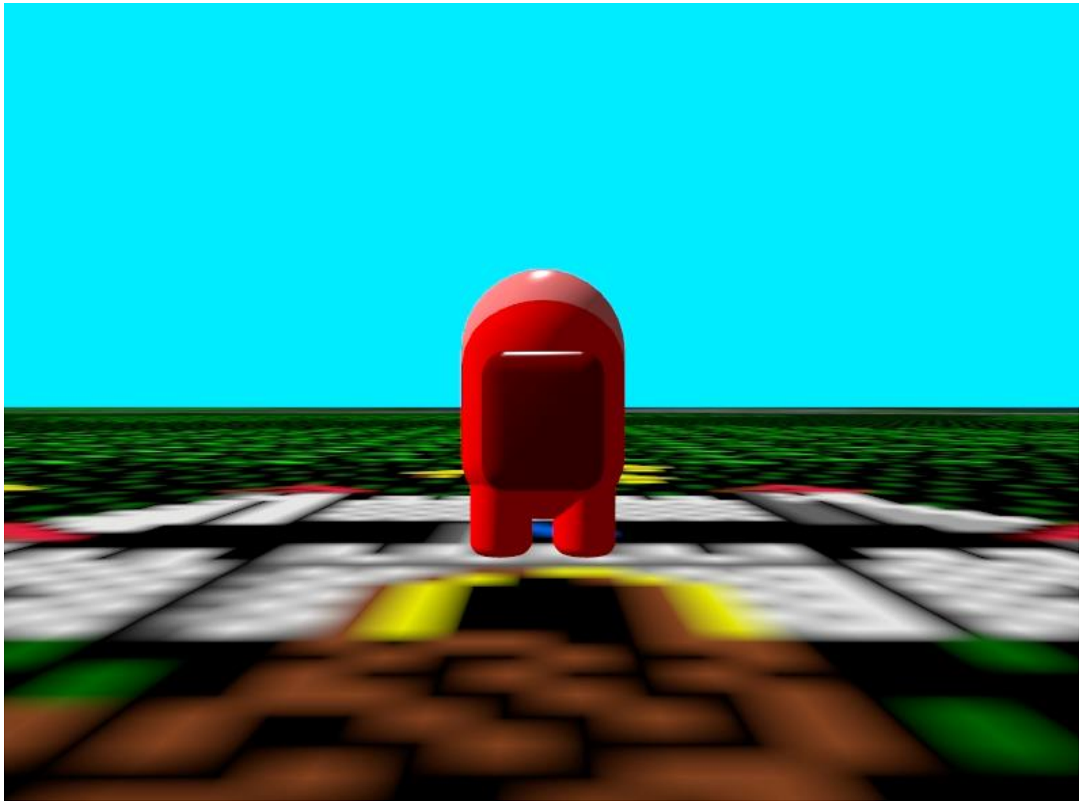
Damping

In both scenario, a direct motion of the camera center can produce unnatural motions that are unpleasant to view.



Damping

A solution is applying a small damping factor, that filters motion with time.



Damping

This can be implemented in the following way:

$$p = p_{OLD} \cdot e^{-\lambda \cdot dt} + p_{NEW} \cdot \left(1 - e^{-\lambda \cdot dt}\right)$$
$$p_{OLD} = p$$

Where λ is the damping speed. Usually a factor of $\lambda = 10$, produces reasonable results.

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)