Projections Wrap Up

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*).

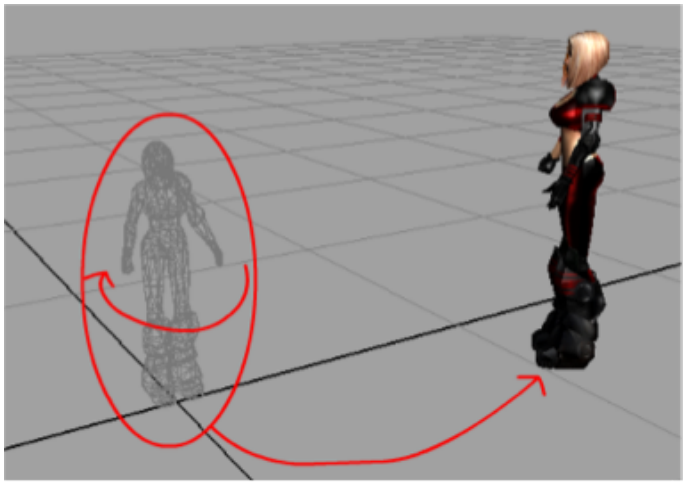
$$p_M = \begin{vmatrix} p_{Mx} & p_{My} & p_{Mz} \end{vmatrix}$$

Step I.a ↓

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

Step I.b ↓

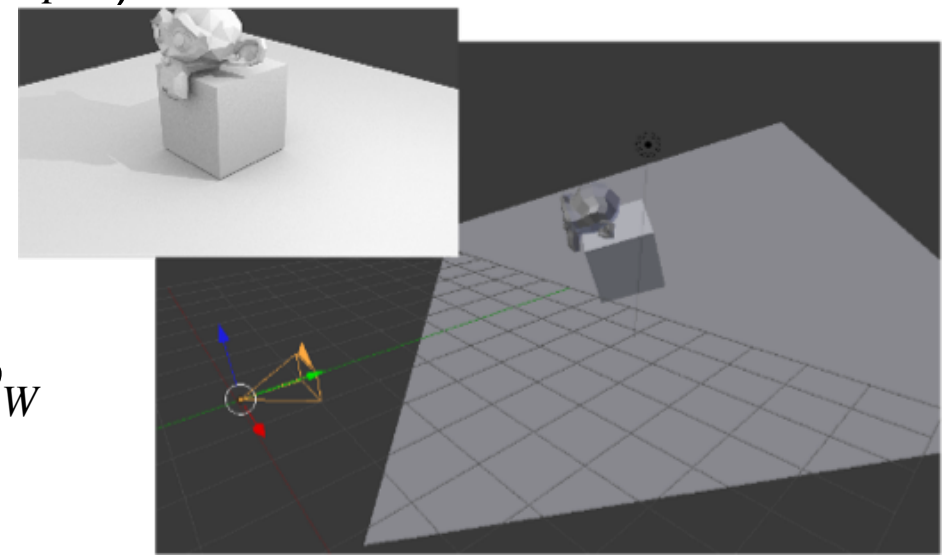
$$p_W = M_W \cdot p_L$$



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

$$p_V = M_V \cdot p_W$$

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

$$p_C = M_P \cdot p_V$$

The World-View-Projection Matrix

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

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

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

Step I-II-III

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

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

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

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.

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

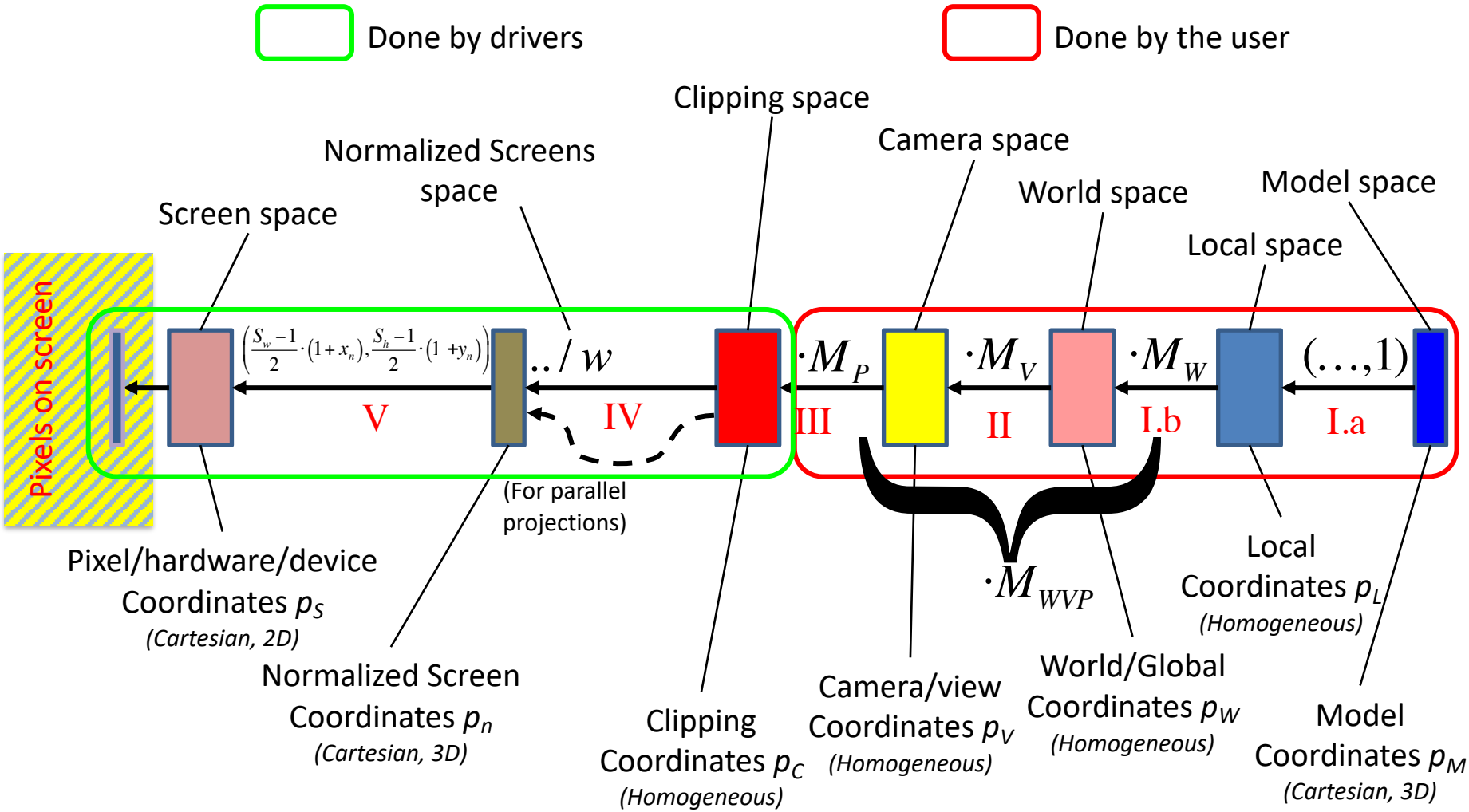
The driver of the video card converts the *clipping coordinates* first to *normalized screen coordinates* (if necessary), and then to *pixel coordinates* to visualize the objects (*Step V*). This is done in a way that is transparent to final user.

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

Step V ↓

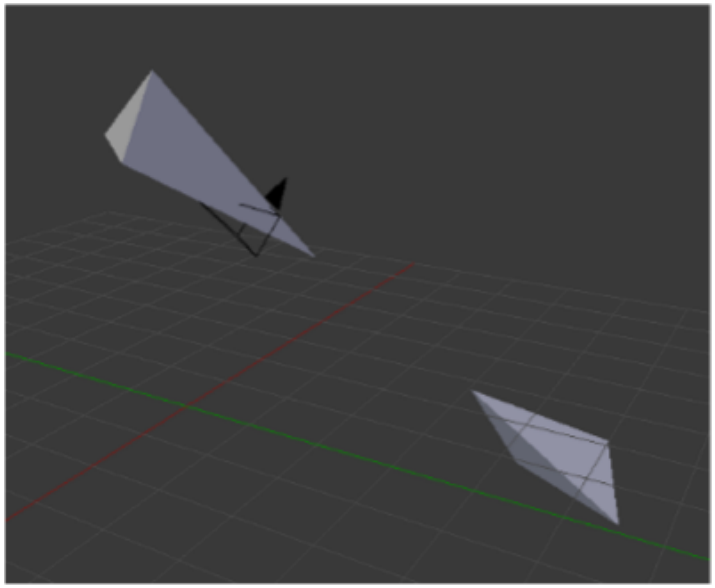
$$(x_S, y_S) = \left(\frac{S_w - 1}{2} \cdot (1 + x_n), \frac{S_h - 1}{2} \cdot (1 + y_n) \right)$$

World-View-Projection Matrices: summary



A complete projection example

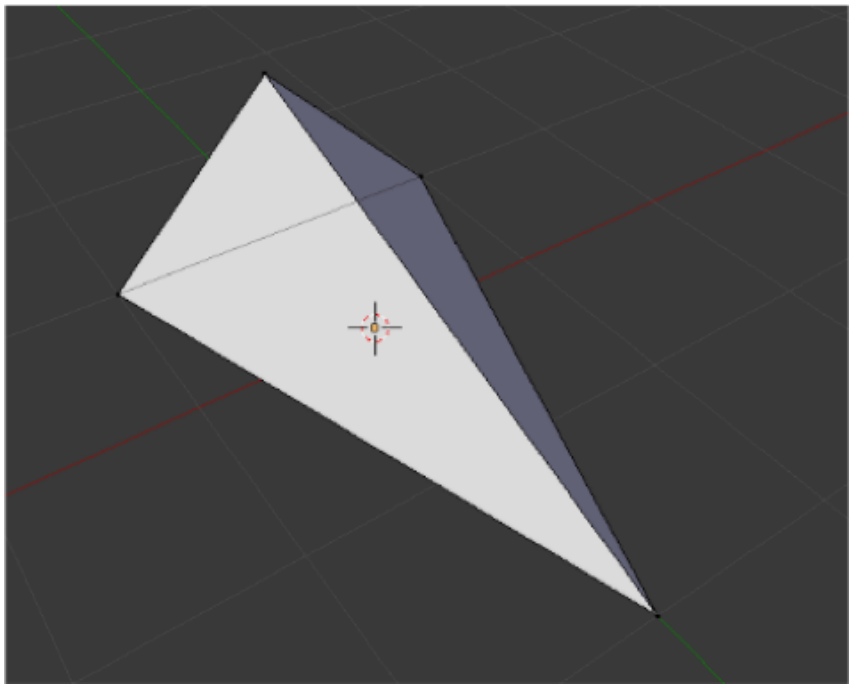
Let us suppose we want to create a simple first-person shooter game, where the user moves a starship.



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

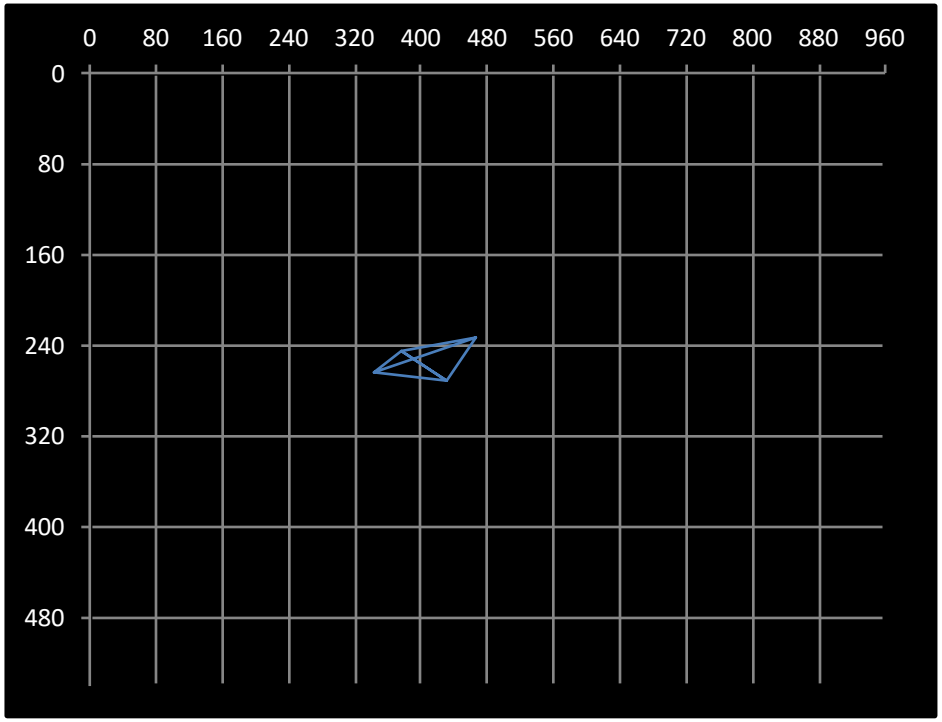
A complete projection example

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



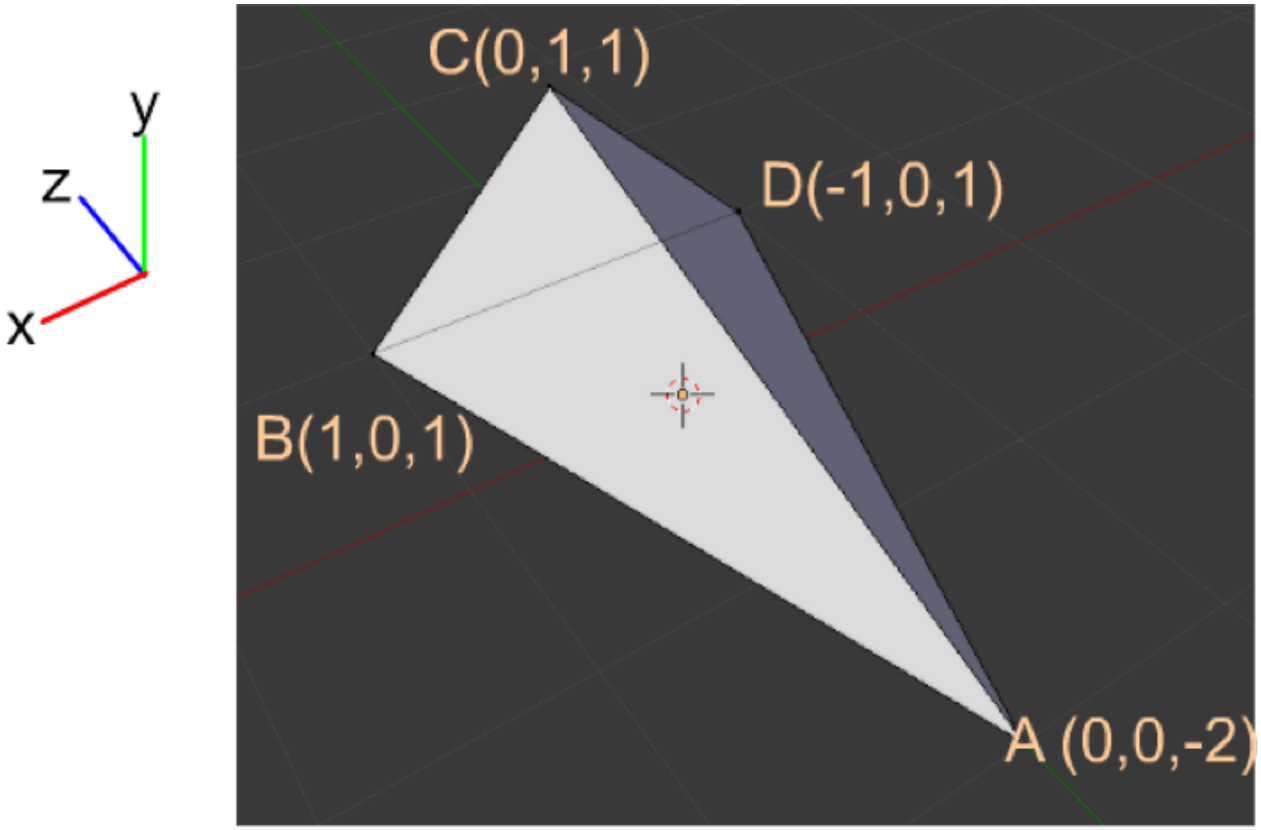
A complete projection example

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



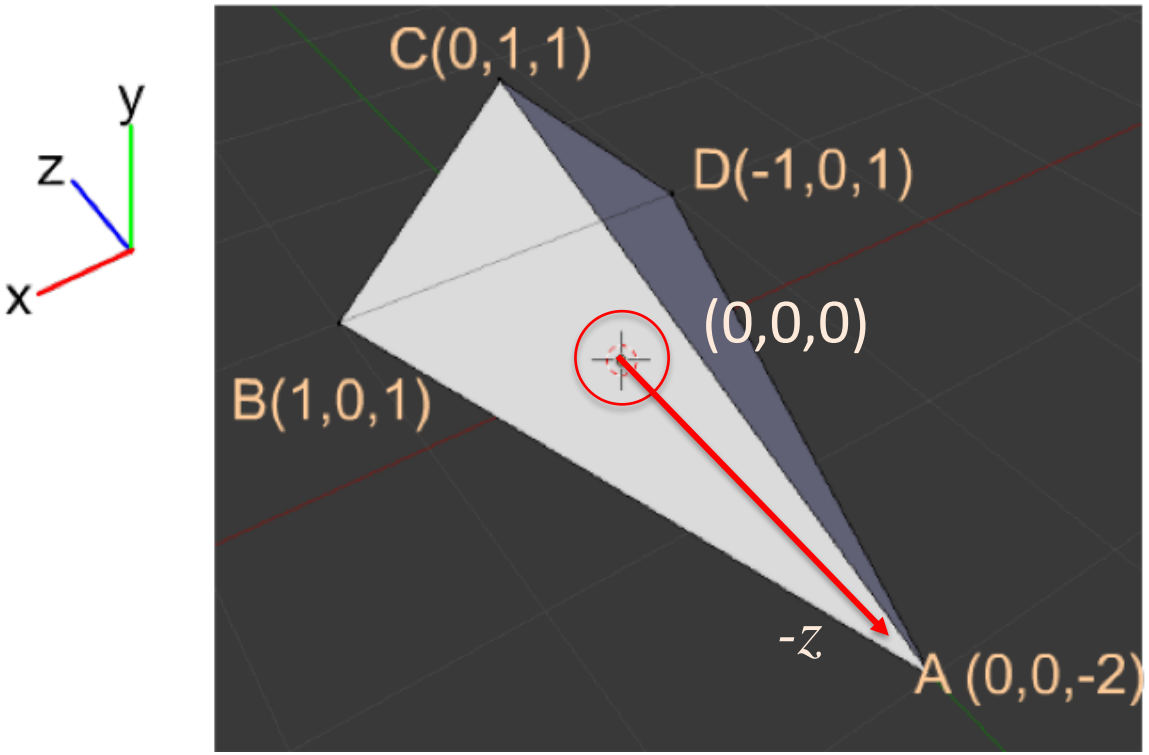
A complete projection example

The following local coordinates characterize them.



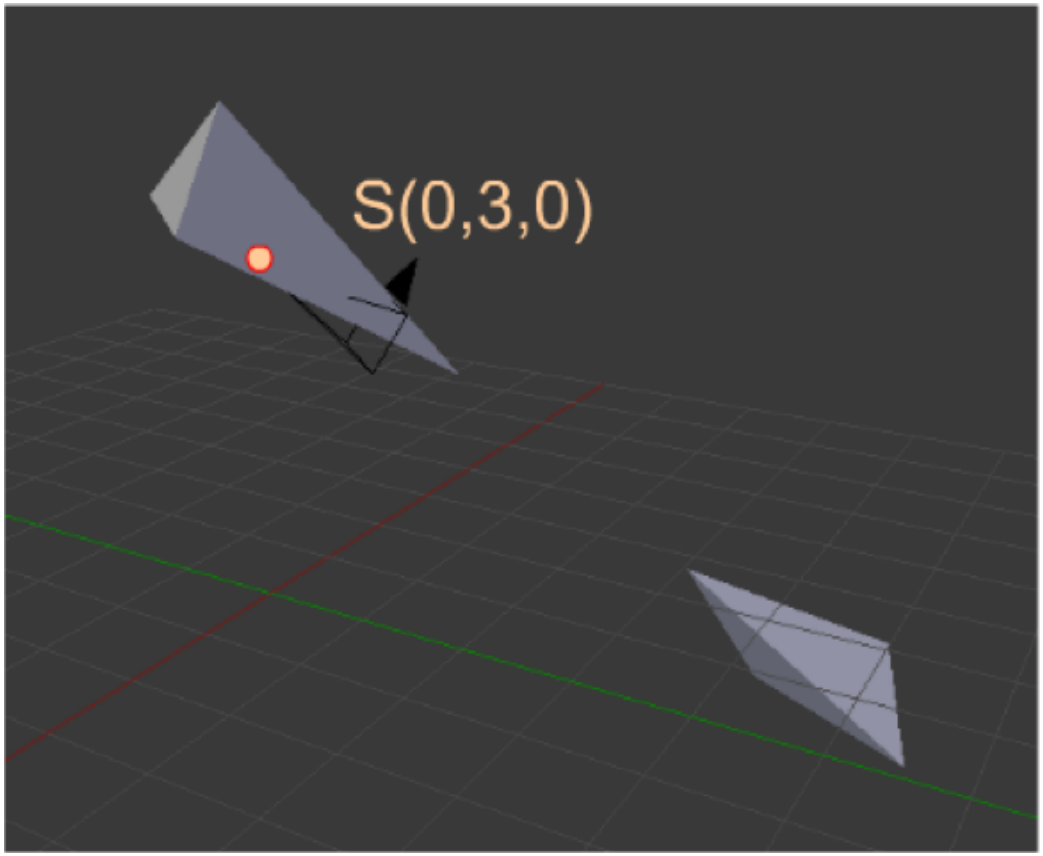
A complete projection example

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



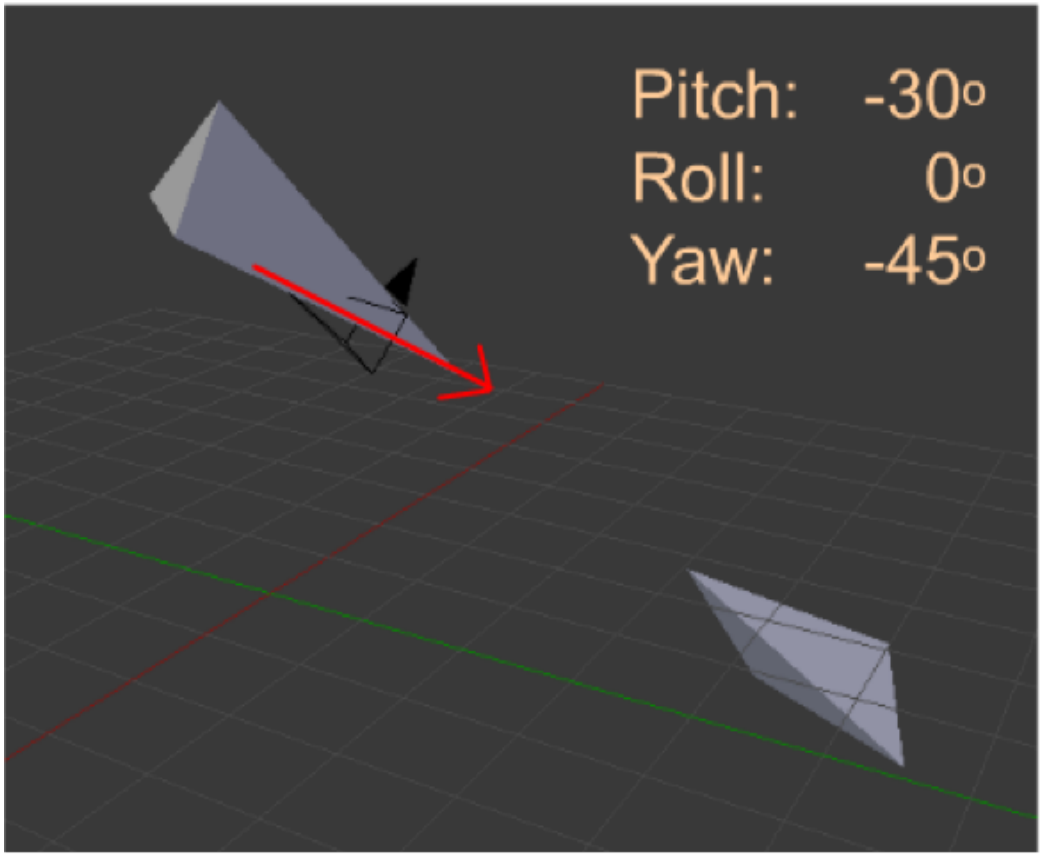
A complete projection example

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



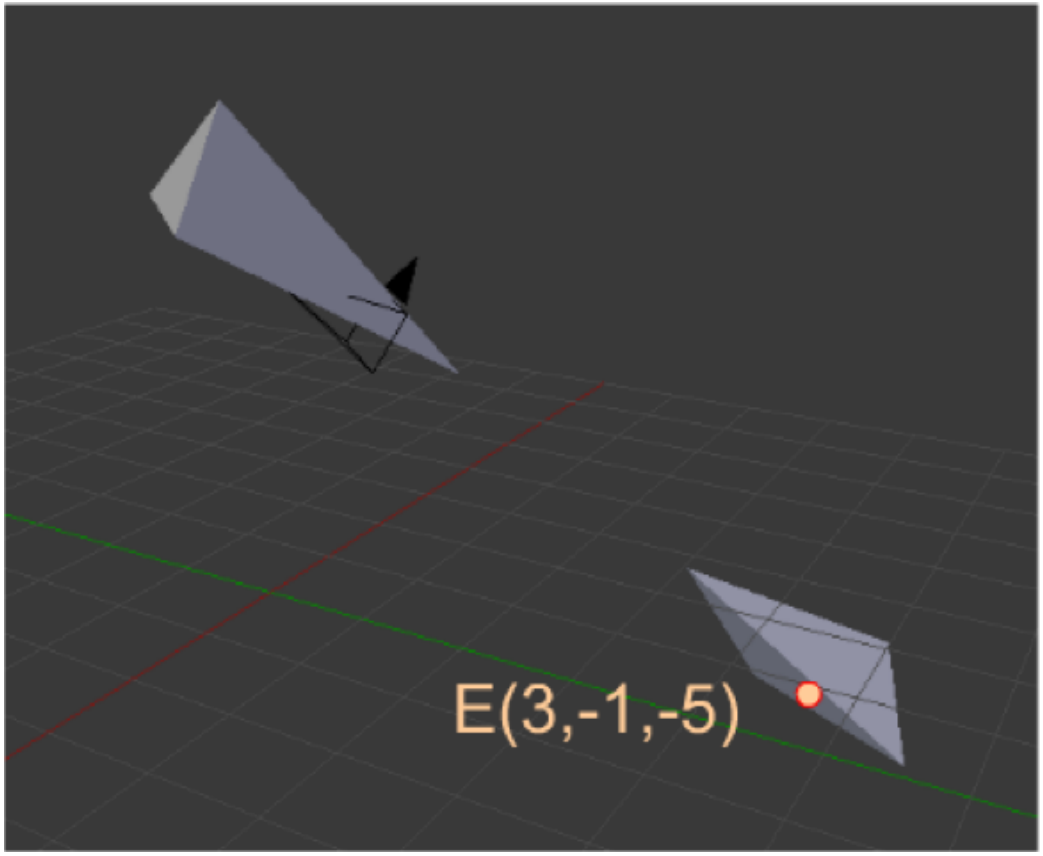
A complete projection example

And it is aiming in the following direction:



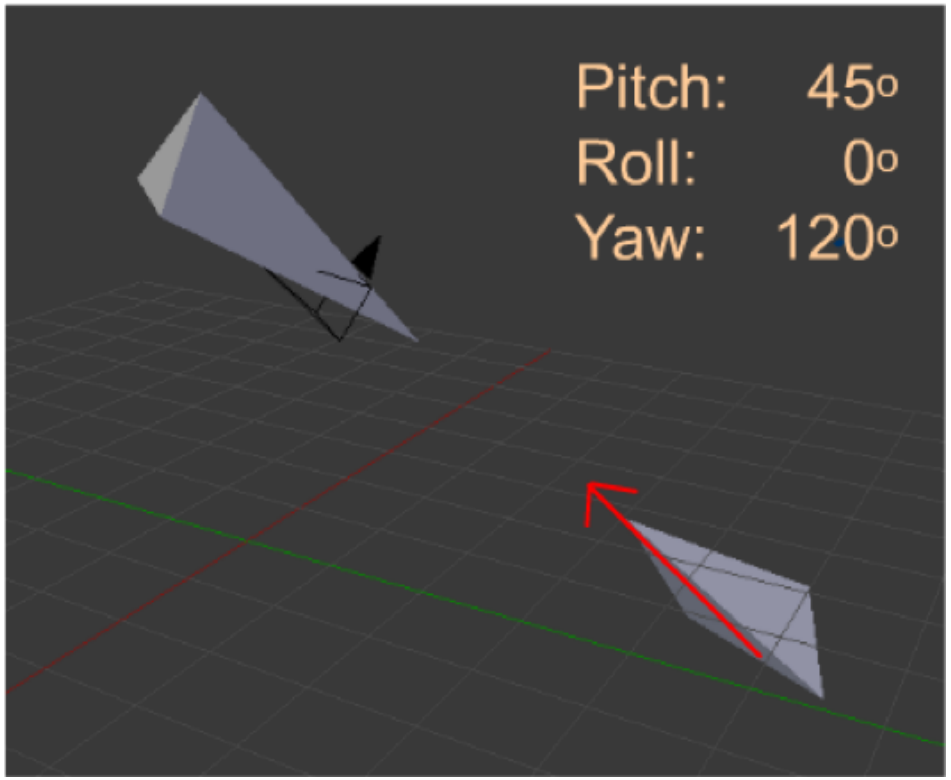
A complete projection example

The enemy fighter is located at:



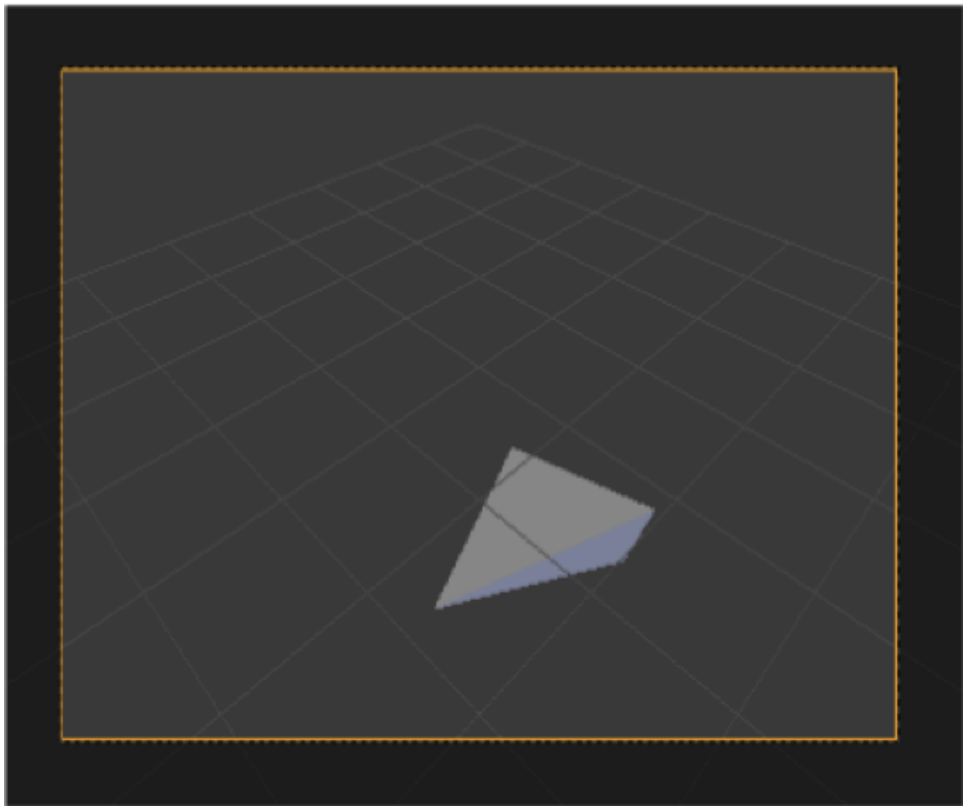
A complete projection example

And it is heading in the following direction



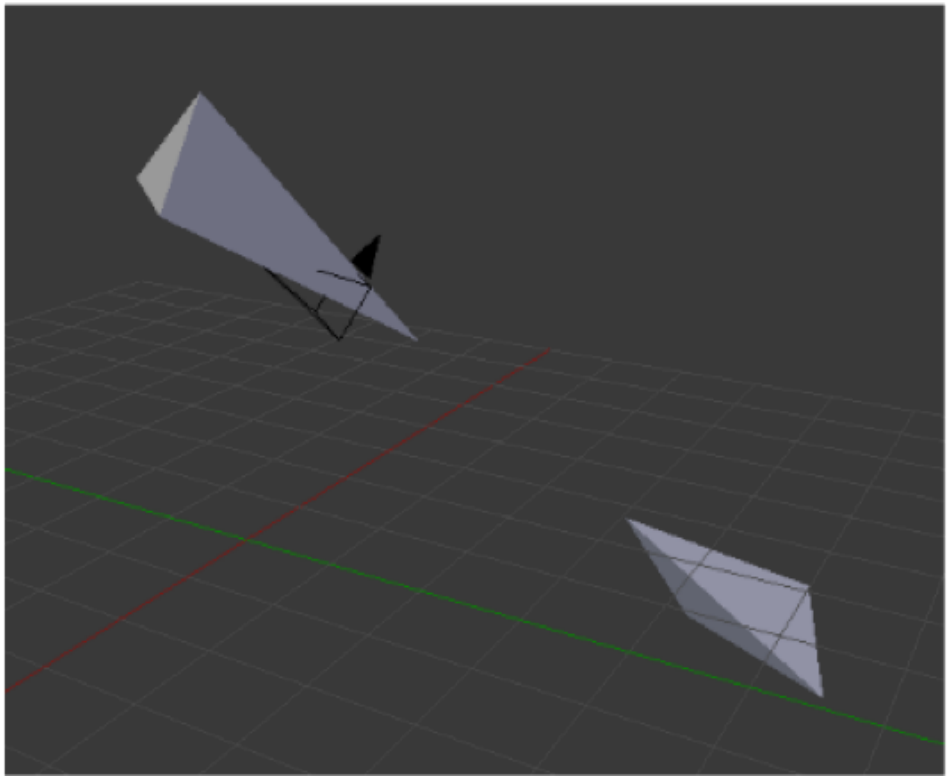
A complete projection example

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^o, 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

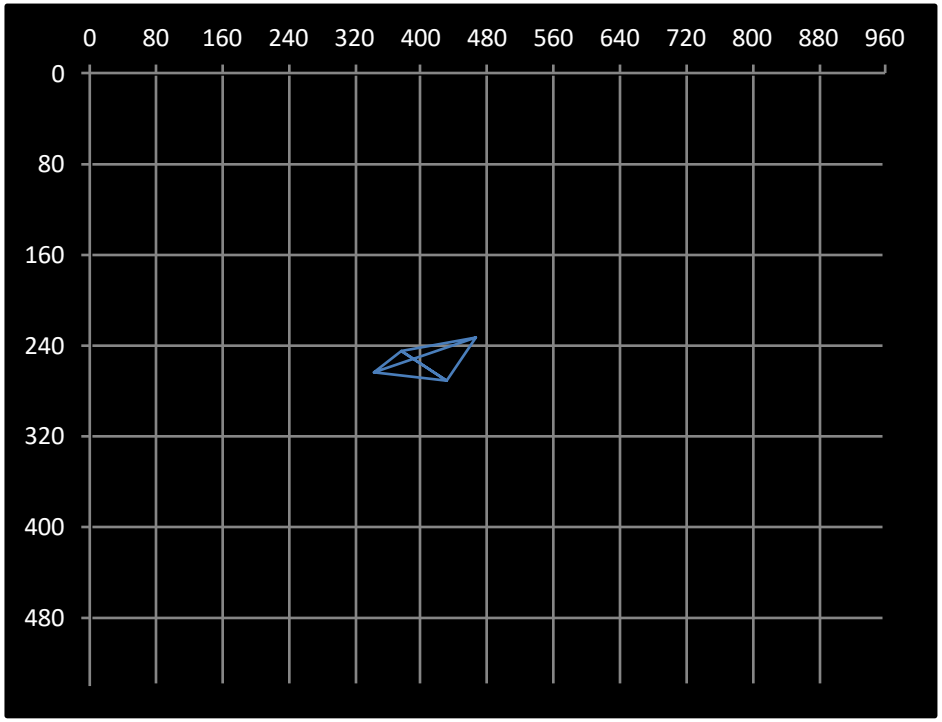
Pixel Coordinates

343	375	430	466
264	244	271	233

Step V

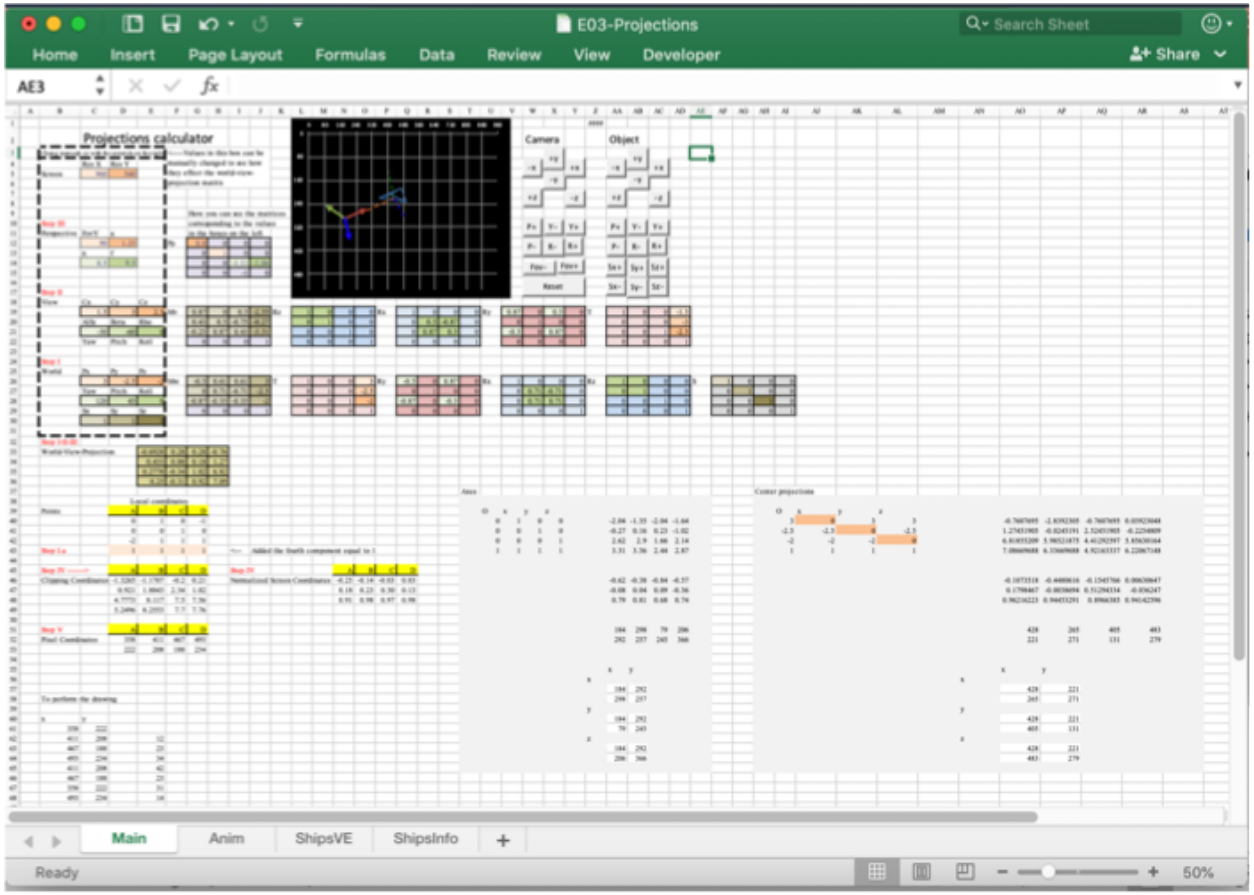
A complete projection example

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 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 cycle is basically identical to the Walk procedure for a camera object.

```
// 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);
```

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:

```
// 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.

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.

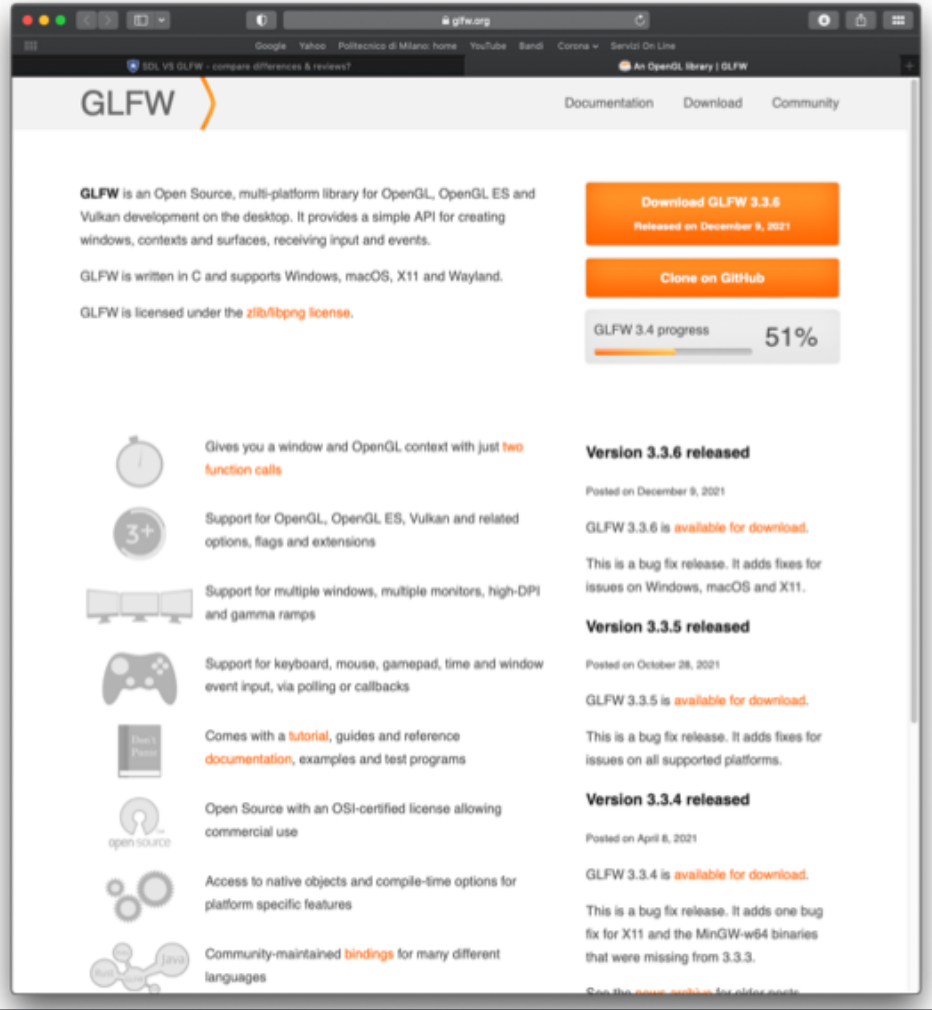
Several libraries have been developed for this task: GLFW and SDL are the two most popular for desktop applications.

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.

```
391
392     void initWindow() {
393         glfwInit();
394
395         glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
396
397         window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
398         glfwSetWindowUserPointer(window, this);
399         glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
400     }
401
402     static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {
403         auto app = reinterpret_cast<Assignment0*>
404                         (glfwGetWindowUserPointer(window));
405         app->framebufferResized = true;
406     }
407
```

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;

391
392     void initWindow() {
393         glfwInit();
394
395         glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
396
397         window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
398         glfwSetWindowUserPointer(window, this);
399         glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
400     }
401
402     static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {
403         auto app = reinterpret_cast<Assignment0*>
404                         (glfwGetWindowUserPointer(window));
405         app->framebufferResized = true;
406     }
407
```

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.

```
2941        if(glfwGetKey(window, GLFW_KEY_A)) {
2942            CamPos -= MOVE_SPEED * glm::vec3(CamDir[0]) * deltaT;
2943        }
2944        if(glfwGetKey(window, GLFW_KEY_D)) {
2945            CamPos += MOVE_SPEED * glm::vec3(CamDir[0]) * deltaT;
2946        }
2947        if(glfwGetKey(window, GLFW_KEY_S)) {
2948            CamPos += MOVE_SPEED * glm::vec3(CamDir[2]) * deltaT;
2949        }
2950        if(glfwGetKey(window, GLFW_KEY_W)) {
2951            CamPos -= MOVE_SPEED * glm::vec3(CamDir[2]) * deltaT;
2952        }
2953        if(glfwGetKey(window, GLFW_KEY_F)) {
2954            CamPos -= MOVE_SPEED * glm::vec3(CamDir[1]) * deltaT;
2955        }
2956        if(glfwGetKey(window, GLFW_KEY_R)) {
2957            CamPos += MOVE_SPEED * glm::vec3(CamDir[1]) * deltaT;
2958        }
```

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
```

GLFW

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.

```
2655        static double old_xpos = 0, old_ypos = 0;
2656        double xpos, ypos;
2657        glfwGetCursorPos(window, &xpos, &ypos);
2658        double m_dx = xpos - old_xpos;
2659        double m_dy = ypos - old_ypos;
2660        old_xpos = xpos; old_ypos = ypos;
2661
2662        glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, GLFW_TRUE);
2663        if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) {
2664            CamAng.y += m_dx * ROT_SPEED / MOUSE_RES;
2665            CamAng.x += m_dy * ROT_SPEED / MOUSE_RES;
2666        }
```

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).

```
2655        static double old_xpos = 0, old_ypos = 0;
2656        double xpos, ypos;
2657        glfwGetCursorPos(window, &xpos, &ypos);
2658        double m_dx = xpos - old_xpos;
2659        double m_dy = ypos - old_ypos;
2660        old_xpos = xpos; old_ypos = ypos;
2661
2662        glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, GLFW_TRUE);
2663        if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) {
2664            CamAng.y += m_dx * ROT_SPEED / MOUSE_RES;
2665            CamAng.x += m_dy * ROT_SPEED / MOUSE_RES;
2666        }
```

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:

```
2655  static double old_xpos = 0, old_ypos = 0;
2656  double xpos, ypos;
2657  glfwGetCursorPos(window, &xpos, &ypos);
2658  double m_dx = xpos - old_xpos;
2659  double m_dy = ypos - old_ypos;
2660  old_xpos = xpos; old_ypos = ypos;
2661
2662  glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, GLFW_TRUE);
2663  if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) {
2664      CamAng.y += m_dx * ROT_SPEED / MOUSE_RES;
2665      CamAng.x += m_dy * ROT_SPEED / MOUSE_RES;
2666  }
```

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

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.

```
41
42  #include <chrono>
43
```

```
2642        static auto startTime = std::chrono::high_resolution_clock::now();
2643        static float lastTime = 0.0f;
2644
2645        auto currentTime = std::chrono::high_resolution_clock::now();
2646        float time = std::chrono::duration<float, std::chrono::seconds::period>
2647                     (currentTime - startTime).count();
2648        float deltaT = time - lastTime;
2649        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.

```
41
42 #include <chrono>
43
```

```
2642        static auto startTime = std::chrono::high_resolution_clock::now();
2643        static float lastTime = 0.0f;
2644
2645        auto currentTime = std::chrono::high_resolution_clock::now();
2646        float time = std::chrono::duration<float, std::chrono::seconds::period>
2647                     (currentTime - startTime).count();
2648        float deltaT = time - lastTime;
2649        lastTime = time;
```