POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DEIB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

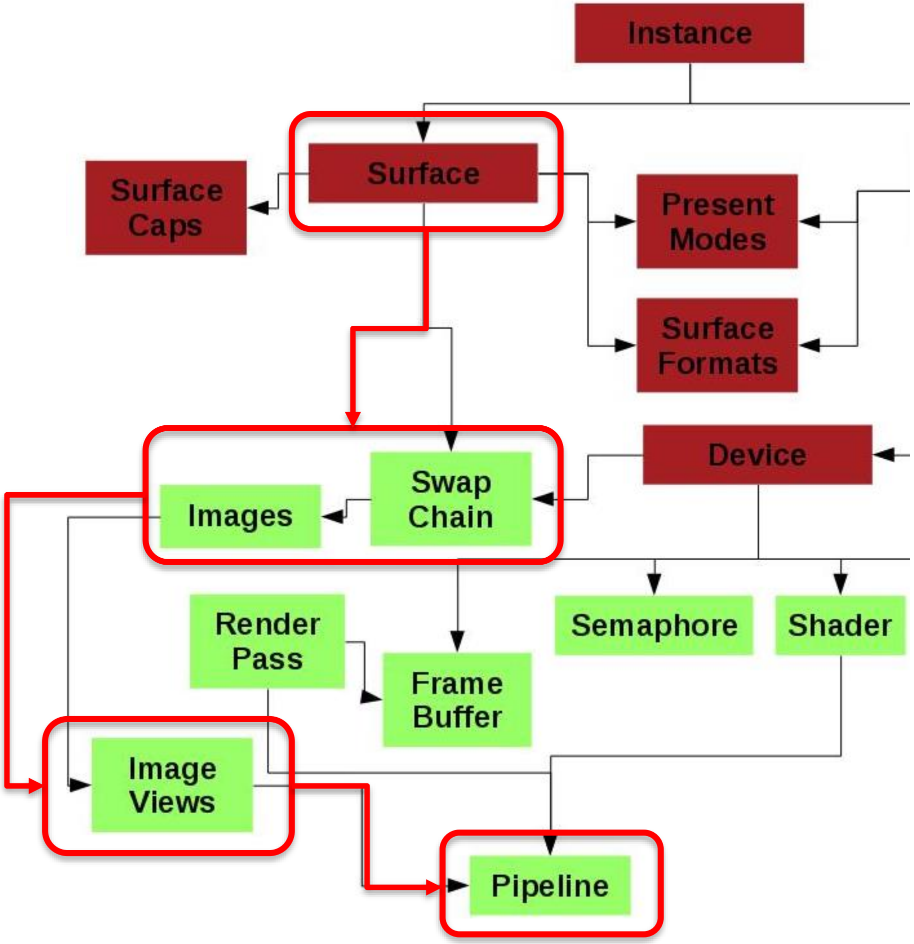
- Navigation Models

Pipeline recreation

The graphics pipeline depends on the Window Surface.

Any change to the window, either for changing resolution, switching to full-screen or resizing, invalidates the pipeline, and all the structure that depends on that.

In such events, the pipeline must be rebuilt.



Pipeline recreation

In `Starter.hpp` the setup and the cleanup of the objects is divided into two stages:

`localInit()` and `localCleanup()` are called once in the lifetime application.

The former configures the Pipeline requirements, but it does not actually creates it. The latter must destroy the pipeline object.

```
void localInit() {
    // Descriptor Layouts [what will be passed to
    P1.init(this, &VD, "shaders/BlinnVert.spv",
                       "shaders/BlinnFrag.spv",
                       {&DSLG, &DSL1});
}

// Here you create your pipelines and Descriptor S
void pipelinesAndDescriptorSetsInit() {
    // This creates a new pipeline (with the curre
    P1.create();

    DS1.init(this, &DSL1, {&T1});
    DSG.init(this, &DSLG, {}); // note that if a D
}

// Here you destroy your pipelines and Descriptor
void pipelinesAndDescriptorSetsCleanup() {
    P1.cleanup();

    DS1.cleanup();
    DSG.cleanup();
}

// Here you destroy all the Models, Texture and De
// You also have destroy the pipelines
void localCleanup() {

    P1.destroy();
}
```

Pipeline recreation

The pipeline is actually created in the method:

`pipelineAndDescriptorSetInit()`, which is called both at the beginning, and every time the pipeline is invalidated.

The cleanup for recreation occurs in:

`pipelineAndDescriptorSetCleanup()`.

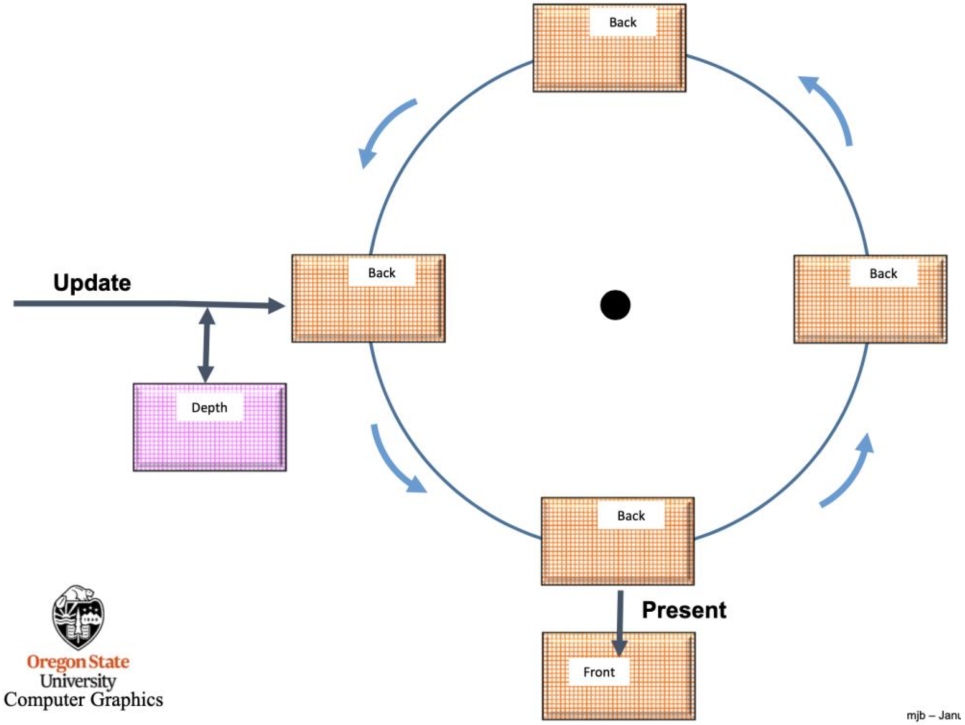
Since *Descriptor Sets* depend on the surface too, they must also be created and destroyed in these functions.

```
void localInit() {
    // Descriptor Layouts [what will be passed to
    P1.init(this, &VD, "shaders/BlinnVert.spv",
                       "shaders/BlinnFrag.spv",
                       {&DSLG, &DSL1});
}

// Here you create your pipelines and Descriptor S
void pipelinesAndDescriptorSetsInit() {
    // This creates a new pipeline (with the curre
    P1.create();

    DS1.init(this, &DSL1, {&T1});
    DSG.init(this, &DSLG, {}); // note that if a 
}

// Here you destroy your pipelines and Descriptor
void pipelinesAndDescriptorSetsCleanup() {
    P1.cleanup();

    DS1.cleanup();
    DSG.cleanup();
}

// Here you destroy all the Models, Texture and De
// You also have destroy the pipelines
void localCleanup() {

    P1.destroy();
}
```

Supporting double and triple buffering

As we have seen, Vulkan supports both *Double* and *Triple Buffering*.

This requires that the application must be able to work on several images at the same time: in Vulkan this means that every frame-buffer dependent command must know the image to which is directed.



Supporting double and triple buffering

In `Starter.hpp`, the frame dependency for what concerns the framebuffers, it is handled automatically.

The frame-dependency is however required also for the *Descriptor Sets*, since the GPU could be render one frame with the relevant values, while the application is preparing the new values of global variable for the next frames.

This dependency must be explicitly takent into account by the user.

Supporting double and triple buffering

Both the `populateCommandBuffer()` and the `updateUniformBuffer()` methods receives an `int currentImage` id they must use when dealing with the *Descriptor Sets*.

```
void populateCommandBuffer(VkCommandBuffer commandBuffer,
                           int currentImage) {
    P1.bind(commandBuffer);
    M1.bind(commandBuffer);

    DSG.bind(commandBuffer, P1, 0, currentImage);    // The Global I
    DS1.bind(commandBuffer, P1, 1, currentImage);    // The Material

    vkCmdDrawIndexed(commandBuffer,
            static_cast<uint32_t>(M1.indices.size()), 1, 0, 0, 0);
}

// Here is where you update the uniforms.
// Very likely this will be where you will be writing the logic of
void updateUniformBuffer(uint32_t currentImage) {
    DSG.map(currentImage, &gubo, 0);
    DS1.map(currentImage, &ubo, 0);
    DS1.map(currentImage, &mubo, 2);
}
```

Supporting double and triple buffering

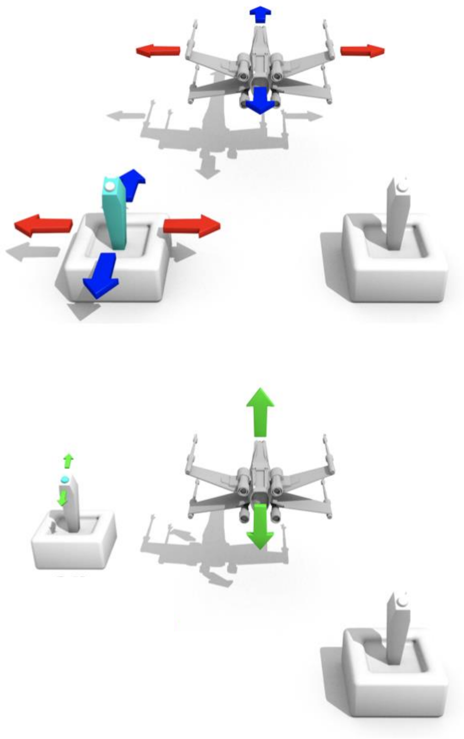
In particular, the `updateUniformBuffer()` method, which deals with defining the values of the *Descriptor Sets* for the current frame, and which in turn specifies the position of the objects and their view, is the main point where the application logic is usually executed.

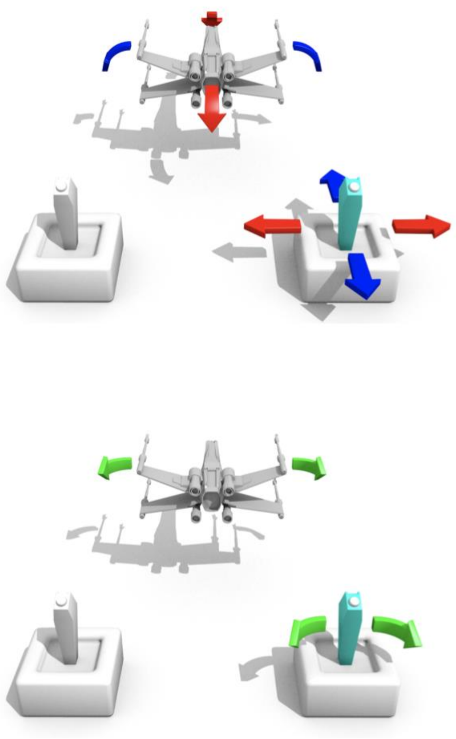
```
void populateCommandBuffer(VkCommandBuffer commandBuffer,
                           int currentImage) {
    P1.bind(commandBuffer);
    M1.bind(commandBuffer);

    DSG.bind(commandBuffer, P1, 0, currentImage);   // The Global I
    DS1.bind(commandBuffer, P1, 1, currentImage);   // The Material

    vkCmdDrawIndexed(commandBuffer,
            static_cast<uint32_t>(M1.indices.size()), 1, 0, 0, 0);
}

// Here is where you update the uniforms.
// Very likely this will be where you will be writing the logic of
void updateUniformBuffer(uint32_t currentImage) {
    DSG.map(currentImage, &gubo, 0);
    DS1.map(currentImage, &ubo, 0);
    DS1.map(currentImage, &mubo, 2);
}
```

Camera navigation models

The motion of an object in 3D 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 considered object flies in an open space without specific reference points (i.e. a starship in orbit).

Controls

All controls – keyboard, joysticks, gamepads and mouse – are usually wrapped to return values in the -1 … +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.

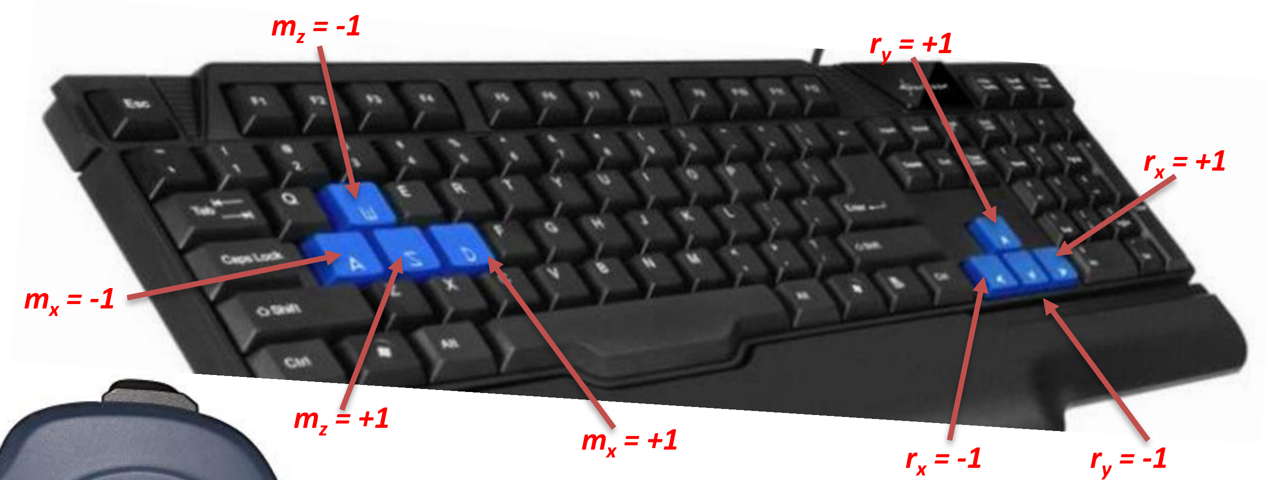
Controls

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

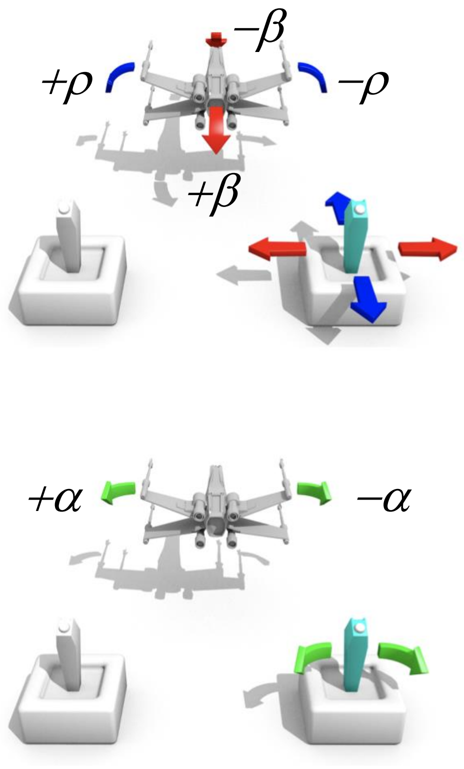
Controls

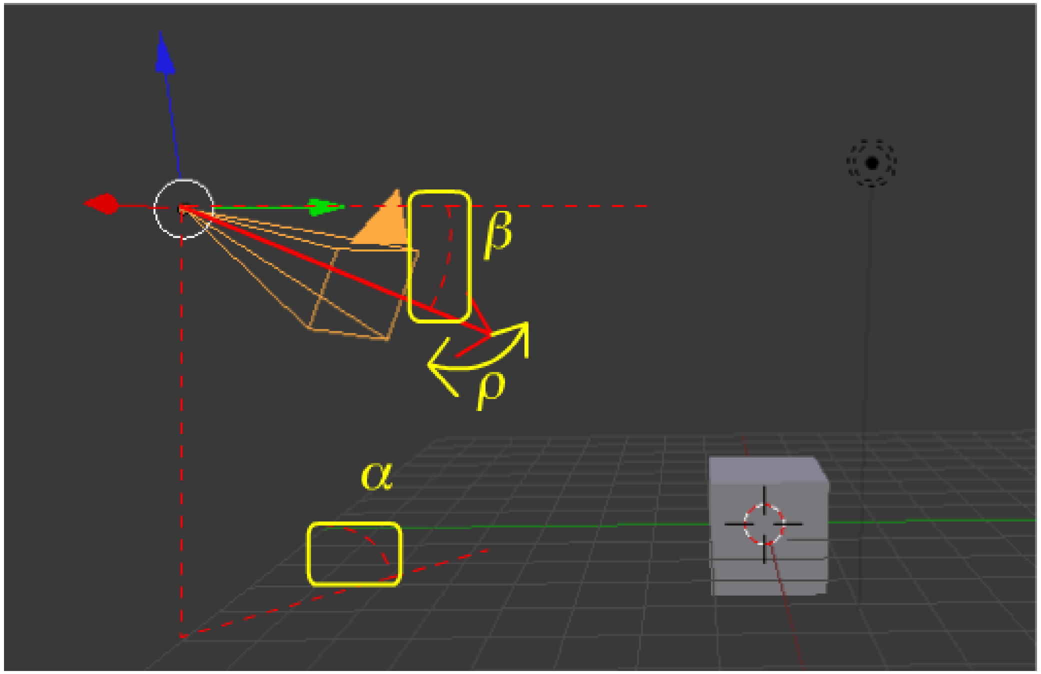
Examples:



The Walk navigation model

In the *Walk* navigation model, rotations around the three axis are used to update the α, β and ρ parameters of the *Look-In-Direction* View Matrix.





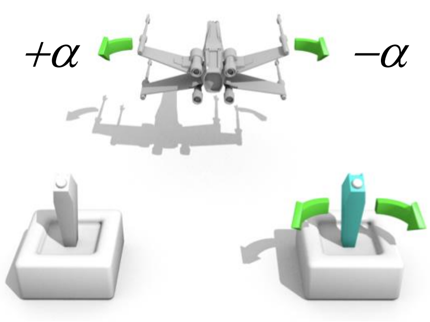
The Walk navigation model

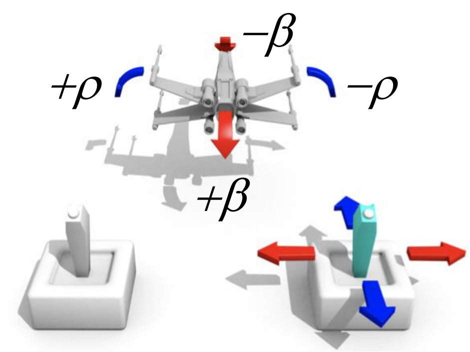
In this case we generally use a vector variable to hold the position, plus three floating points variables to store the rotations.

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

The Walk navigation model

The rotation of the view α influences the direction of motion, while the pitch β and the roll ρ act only on the view.





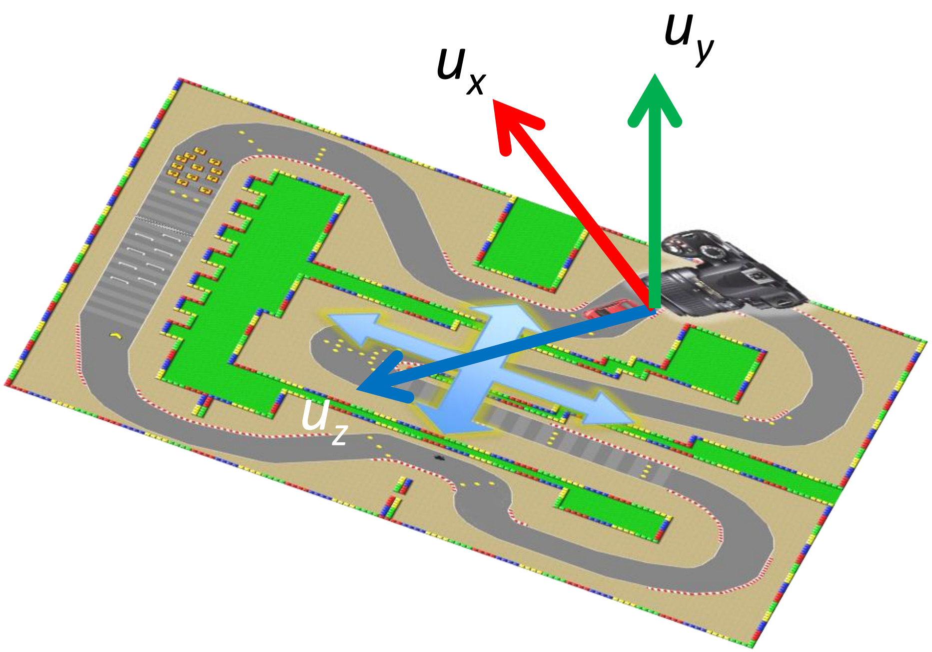
The Walk navigation model

To support this feature, three vectors u_x, u_y and u_z, that represents the unitary movement in each axis, are computed.

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

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

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

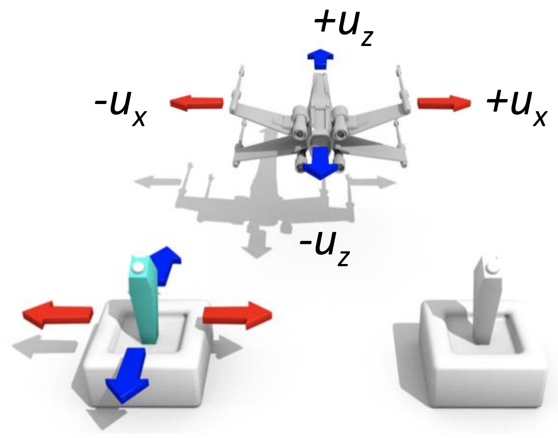


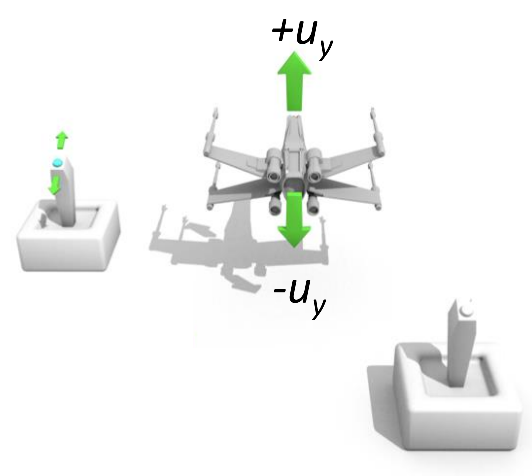
Here the *[…].xyz* notation is used to denote the cartesian coordinate corresponding the the homogeneous one.

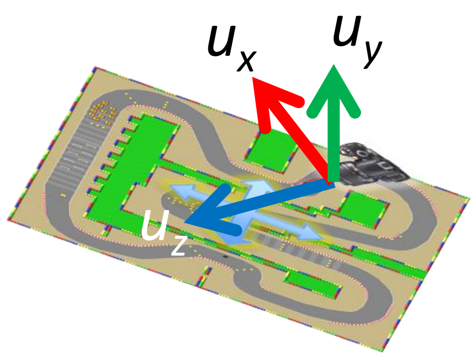
The Walk navigation model

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.

Rotation directly changes the three angles.







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 taken into account. This time difference *dt*, is measured in seconds.

Update

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

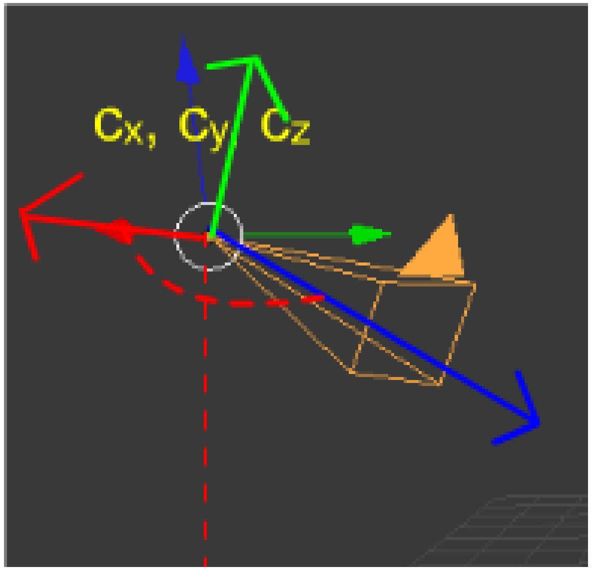
In the Walk model, it is easier to have variables containing the position and direction of the camera, and use them to recreate a new view matrix at each frame update.

```
// 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);
```

The Fly navigation model

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

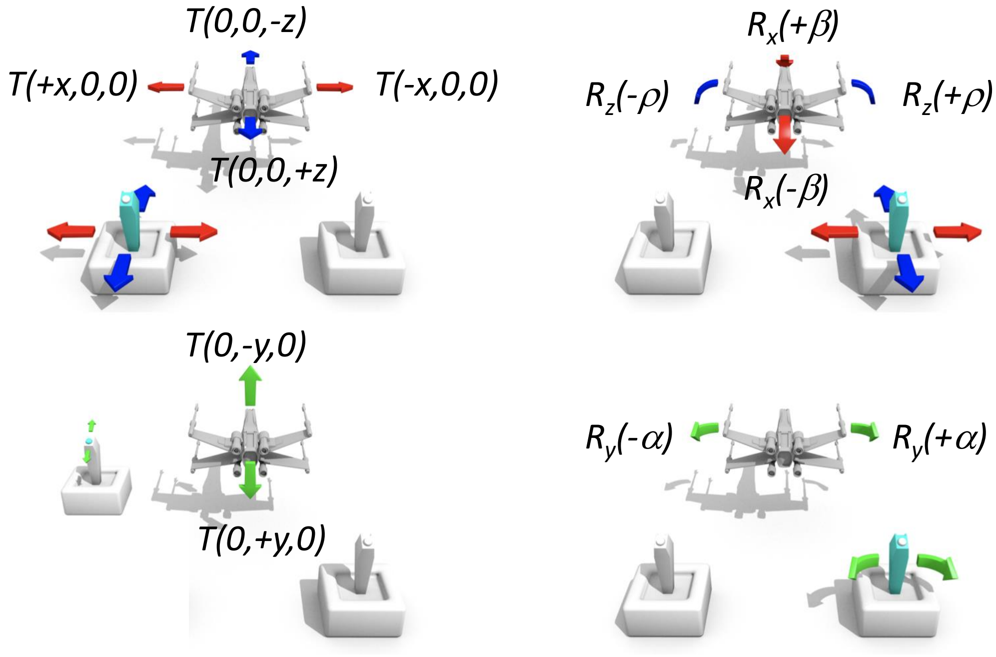
In this case, we store only the view matrix and we update it directly.




```
// 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.

Position of the viewer

As we have seen, the rendering equation requires the position of the canera to compute the viewer direction.

In the *Walk* model, this is already available, since it is already one of the parameters of the technique.

In the Fly model, it can be obtained from the camera matrix (which is the inverse of the view matrix), remembering that the last row of a transform matrix holds the translation.

```
glm::mat3 CamPos = glm::vec3(glm::inverse(ViewMatrix) * glm::vec4(0, 0, 0, 1));
```

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.

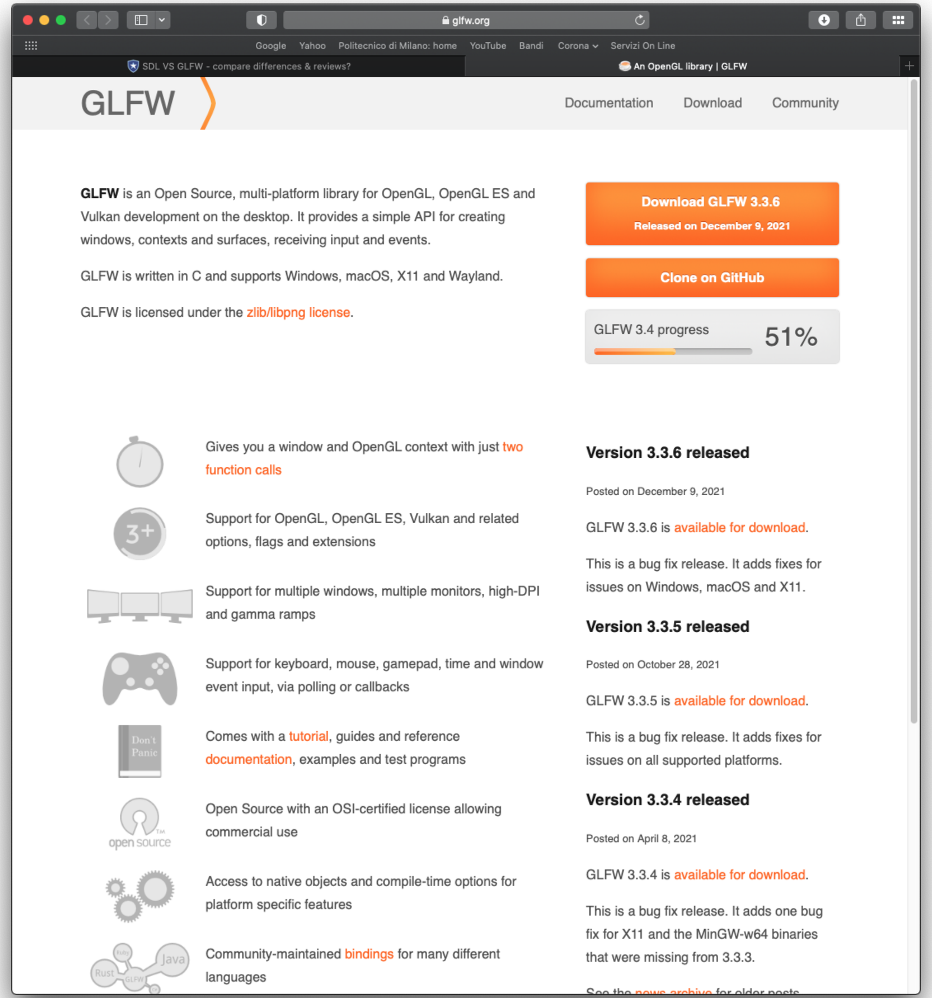
Several libraries have been developed for this task: GLFW and SDL are the two most popular for desktop applications.

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

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
```

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.

```
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;
}
```

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 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;
}
```

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.

```
#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;
```

Accessing controls in `Starter.hpp`

The function getSixAxis() of Starter.hpp, receives four variables, and fills them with the following values:

- A floating point containing the time in seconds since the previous frame
- A vec3 containing the motion axes
- A vec3 containing the rotation axes
- A boolean containing the pressure of a fire button.

```
void getSixAxis(float &deltaT,
                glm::vec3 &m,
                glm::vec3 &r,
                bool &fire) {
```

Controls are mapped in the following way:

```
void getSixAxis(float &deltaT,
                glm::vec3 &m,
                glm::vec3 &r,
                bool &fire) {
```

Camera position in third person applications

In a Third Person application, the users moves a main actor object, and the camera follows it.

In particular, we have to determine three positions: the one of the object, the on of the camera and the one of its target. The target is usually connected to the position of the object.

Lets focus on two popular cases:

- Flying a space ship
- Controlling a character moving on the ground

Flying a space ship in third person

The update cycle for the main object motion is 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 * m.x * dt, mu * m.y * dt, mu * m.z * dt));
WorldMatrix = WorldMatrix * glm::rotate(glm::mat4(1), omega * r.x * dt,
                            glm::vec3(1, 0, 0));
WorldMatrix = WorldMatrix * glm::rotate(glm::mat4(1), omega * r.y * dt,
                            glm::vec3(0, 1, 0));
WorldMatrix = WorldMatrix * glm::rotate(glm::mat4(1), omega * r.z * dt,
                            glm::vec3(0, 0, 1));
```

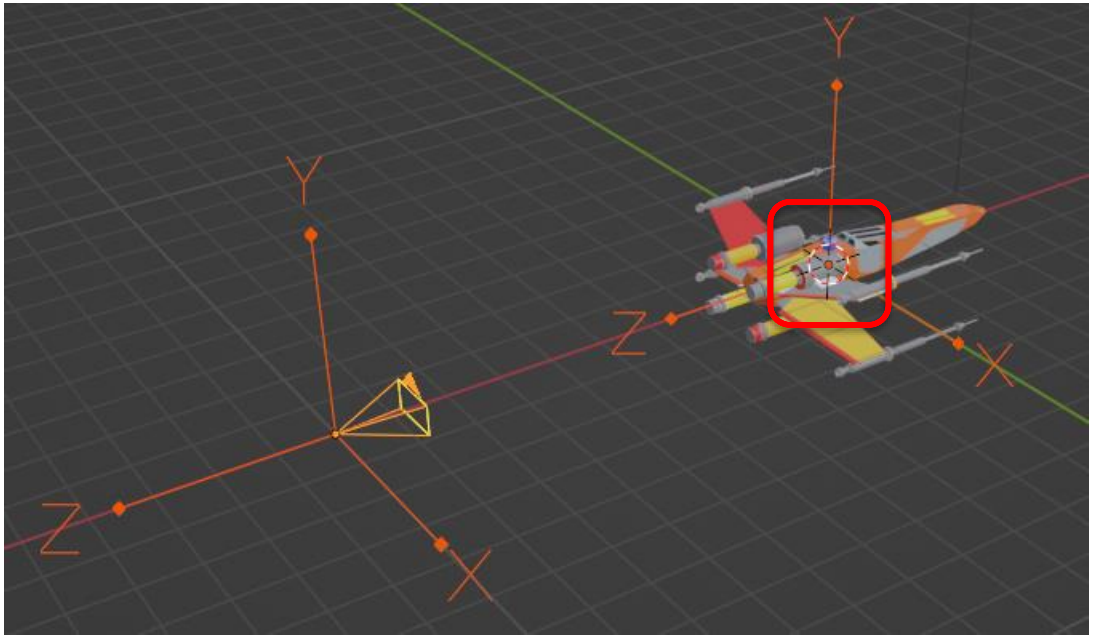
The local and global coordinates model – quaternion form

The orientation can generally be stored more efficiently using a quaternion. In this case, however, axis direction should be retrieved.

```
// 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 * r.x * dt, glm::vec3(1, 0, 0));
rot = rot * glm::rotate(glm::quat(1,0,0,0), omega * r.y * dt, glm::vec3(0, 1, 0));
rot = rot * glm::rotate(glm::quat(1,0,0,0), omega * r.z * 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 * m.x * dt;
pos += uy * mu * m.y * dt;
pos += uz * mu * m.z * dt;

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

Flying a space ship in third person

When flying a space ship, the target generally corresponds to the position of the object.



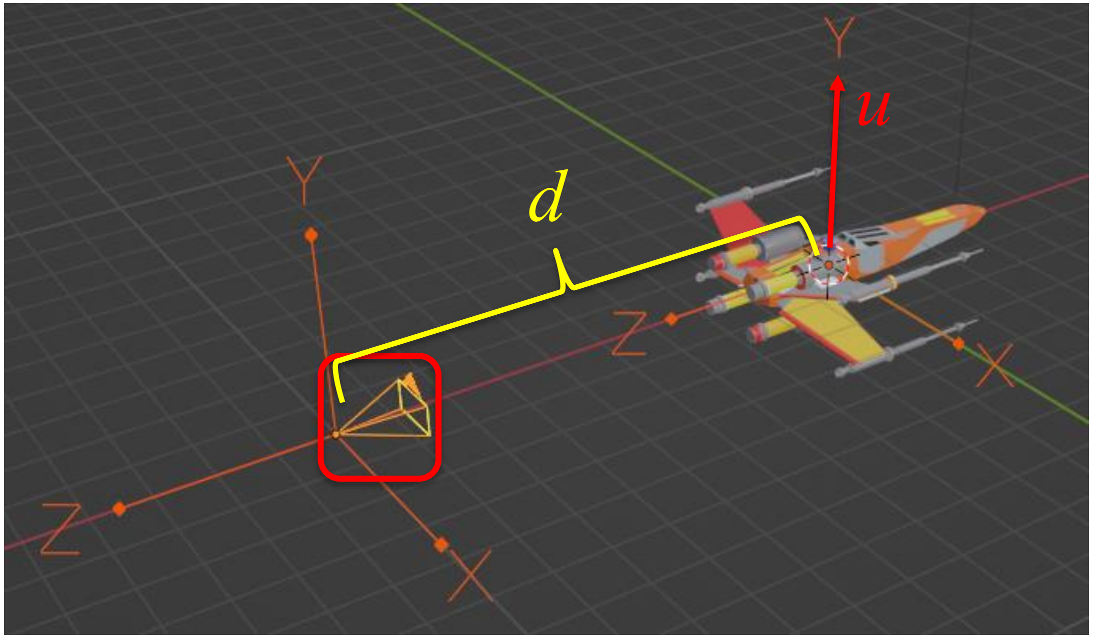
Flying a space ship in third person

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)*.

The up vector, which generally corresponds to the *y-axis* of the ship, can be computed from the World Matrix as well.

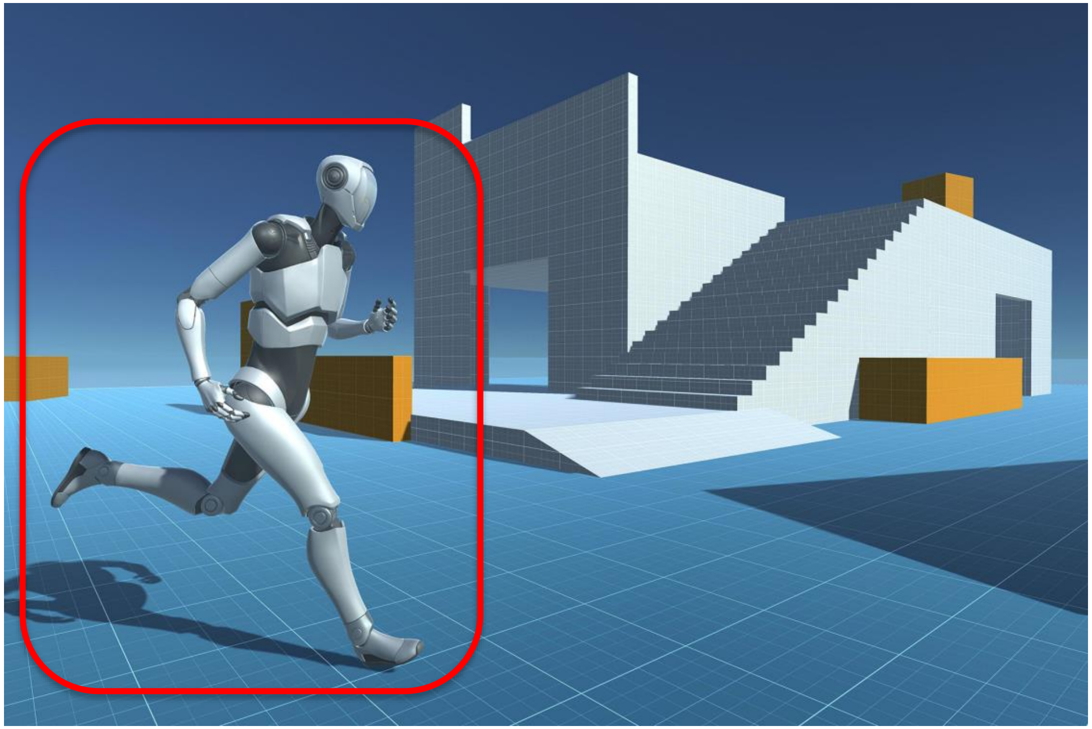
$$c = [M_W \cdot (0,0,d,1)^T]_{.xyz}$$

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



Moving objects on a ground based scene

A *Ground* motion technique is 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,
                             yaw, pitch, roll);
```

Rarely needed

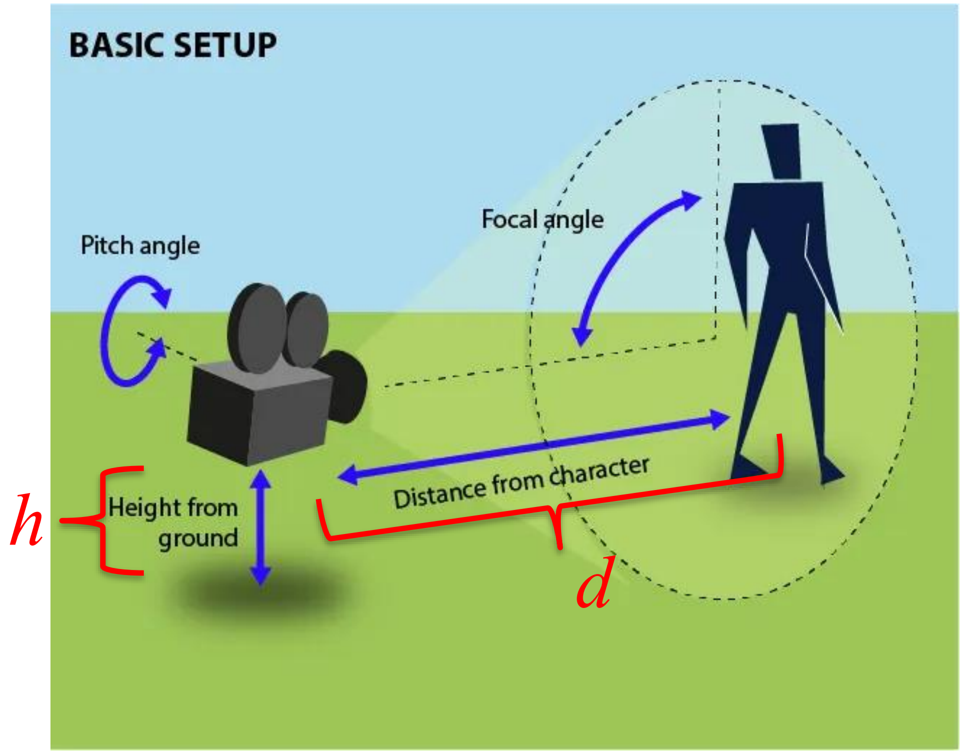
Rarely needed

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 of the character.

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.

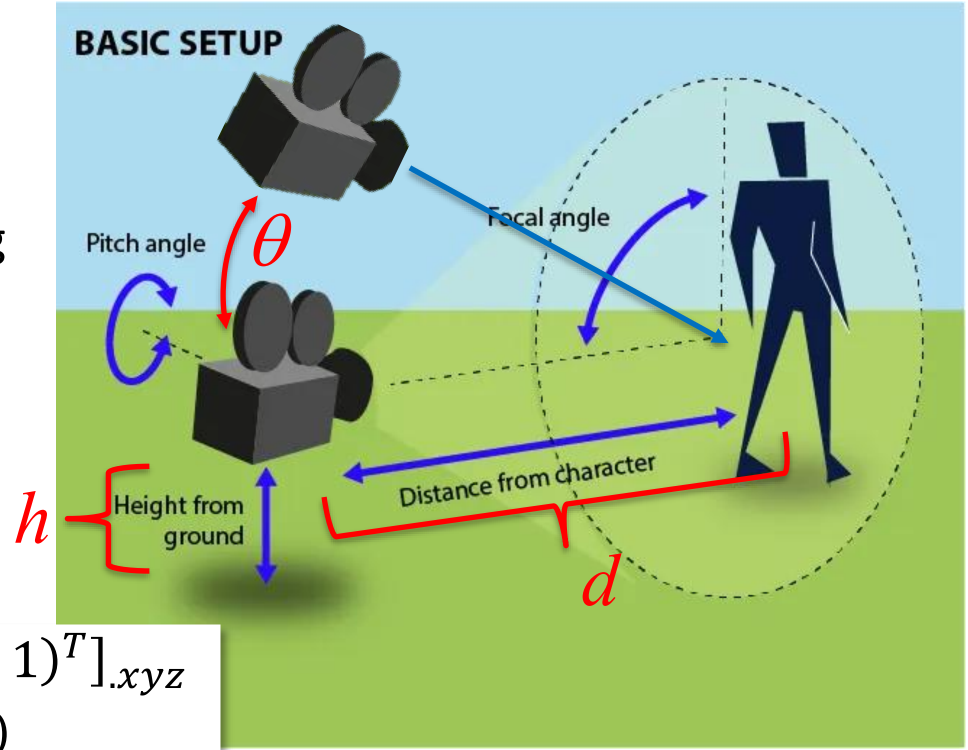


Moving a character in third person

As introduced, the character usually uses only the yaw to control the direction.

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

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



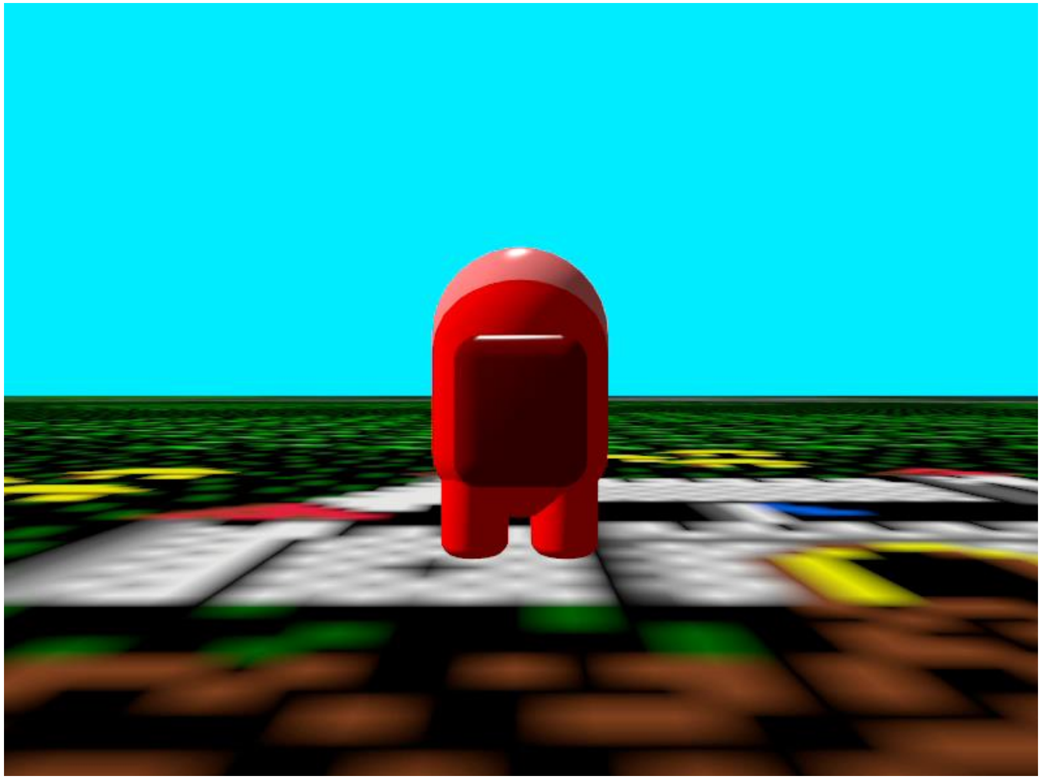
$$c = [M_W \cdot (0, h + d \cdot \sin\theta\,, d \cdot \cos\theta\,, 1)^T]_{.xyz}$$
$$a = [M_W \cdot (0,0,0,1)^T]_{.xyz} + (0,0,h)$$

$$u = (0,1,0)$$

The up vector is almost always the *y-axis*.

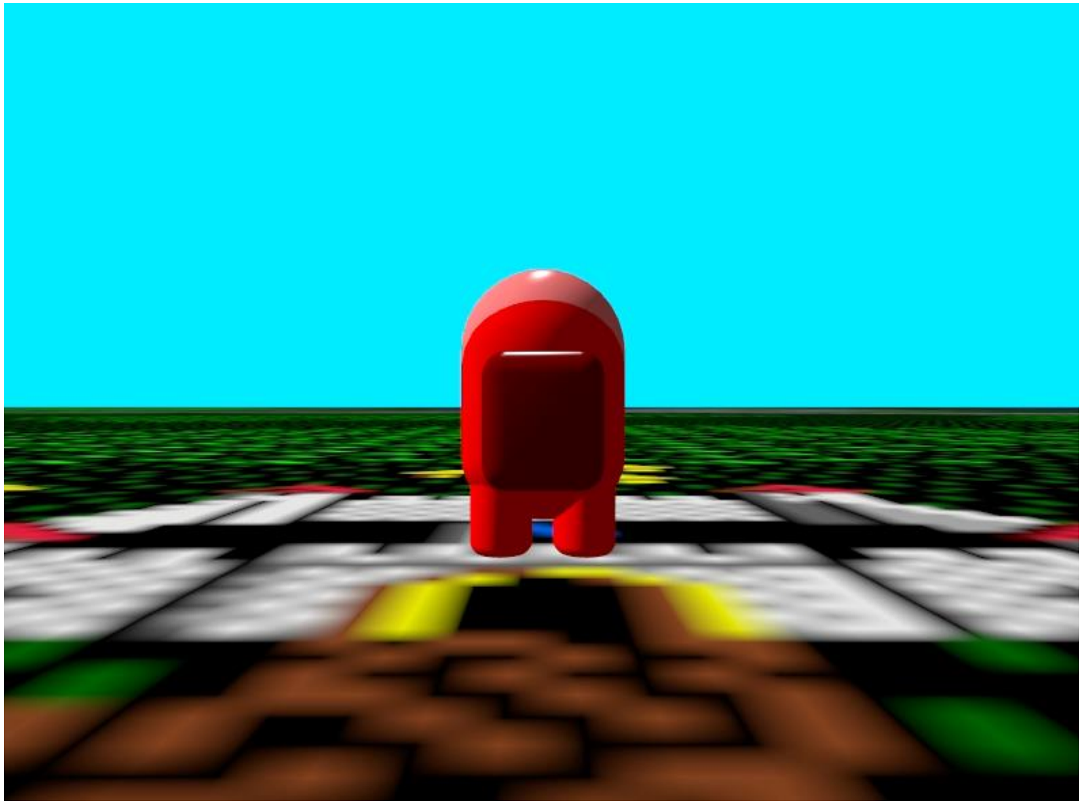
Damping

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 good results.

In the previous equation, p_{NEW} is determined using one of the techniques previously described to compute position of the center of the camera.

Interesting results can be obtained using the same technique also to filter also the main position of the object.

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)