MuJoCoXR

Linking MuJoCo and a VR environment

2024-07-18

Youenn Le Jeune
INSA Rennes / DIAG Sapienza

I. Introduction and State of the Art

The MuJoCo physics engine is a great tool that contains many useful features that are nowhere to be found in other engines. Such features are, for instance, realistic soft body simulation. Making interactive simulations is possible through the provided API, in C or in Python. It is easy to integrate motion-capture devices and to get feedback from sensors.

However, the rendering pipeline can be difficult to comprehend. There are multiple pre-made visualization solutions: an interactive one for your MJCF files via the "Simulate"[1] code sample, a passive one that you can use in your Python application to show your interactive simulation in real-time[2], a full Renderer Python class[3] that can be really useful for notebooks and, if nothing else is suitable, API functions to render directly in an OpenGL context[4].

As we can see, there is no built-in way to visualize a MuJoCo simulation in a Virtual Reality environment. At the time of writing, some work is being done to add VR support to the "Simulate" application[5]. Nevertheless, it is still not the best choice if you want to fully immerse people in your simulation[6]. One possible solution is to use the Unity Plug-in for MuJoCo[7] and configure the Unity application to display in VR (*not tested*), but again that is not ideal: we do not necessarily need a whole game engine to run a simple simulation. Thus, we will program our own solution to display a MuJoCo simulation in a VR headset.

Regarding the VR part, there are multiple solutions to render to a headset:

- Directly use the vendor-specific API (Meta Quest, HTC Vive…). This require code to be remade for each new device we want to support.
- Use OpenVR, which contains support for VR headsets from multiple vendors but it is tied to Steam.[8]
- Use OpenXR, which is a free standard not tied to any VR company and implemented by all major VR headsets on the market.[9]

We have chosen to use OpenXR because it seems to be the standard that will be mainly used in the future. There are Python bindings available[10]. It supports multiple graphics API including OpenGL that we will use because it is what MuJoCo can render to.

To sum up, our solution will render MuJoCo on a VR headset using OpenGL via the OpenXR standard, all of that in Python.

[1] https://mujoco.readthedocs.io/en/stable/programming/samples.html#sasimulate
[2] https://mujoco.readthedocs.io/en/stable/python.html#passive-viewer
[3] https://github.com/google-deepmind/mujoco/blob/main/python/mujoco/renderer.py
[4] https://mujoco.readthedocs.io/en/stable/programming/visualization.html
[5] https://github.com/google-deepmind/mujoco/pull/1452
[6] The "Simulate" application contains a lot of UI elements to control the simulation and visualization, which breaks immersion.
[7] https://mujoco.readthedocs.io/en/stable/unity.html
[8] https://github.com/ValveSoftware/openvr
[9] https://www.khronos.org/openxr/
[10] https://github.com/cmbruns/pyopenxr

II. Theory

II - 1. Graphics 101

To render something on a screen or a VR headset, computers use graphic cards (GPUs). Those graphic cards receive orders through Graphic APIs such as OpenGL, Direct3D or more recently, Vulkan. Each GPU supports different versions of those APIs, and old GPUs do not even support some APIs.

A Graphic API consists of a large set of instructions related to graphics: draw a line from here to there, clear the screen with this color, draw this texture. OpenGL instructions are all prefixed with `gl` and make heavy use of constants. For instance, a possible instruction is `glClear(GL_COLOR_BUFFER_BIT)` .

Graphic APIs instructions are not only used to *draw*: with the support of *framebuffers*, it is possible for instance to draw on some in-memory texture and then read it to draw on top of another buffer with a different scale. Framebuffers are a "collection" of renderbuffers and textures and can have multiple *attachements*: colors, depth and stencil.

OpenGL is built on the principle of *extensions*.

Graphic APIs work with a *context*: to use their functions, a context must be bound to the thread. It contains references to all GL objects created within it. A context is usually tied to a window. To create those contexts, we usually use dedicated libraries such as GLFW which allows to create a window and attach the associated context to the calling thread.

In Python, there exist a binding for OpenGL: `pyopengl` [11].

II - 2. OpenXR

OpenXR is a standard implemented by pretty much all VR devices. It provides methods for most of the features: displaying images, getting head position in the room, getting controller positions, rendering haptic feedback, enable passthrough for compatible headsets, and so on. For vendor-specific features, extensions are present in OpenXR to use them.

To render images to the eyes, OpenXR uses the concept of *swapchains*. A swapchain is a collection of framebuffers that display sequentially to a screen. It allows to draw on a "back" framebuffer while another "front" one is being displayed on the screen.[12]
In this project, we will make use of only one "stereo" swapchain, which will contain one image for both eyes at the same time. It is also possible to have multiple swapchains, e.g. one per eye.

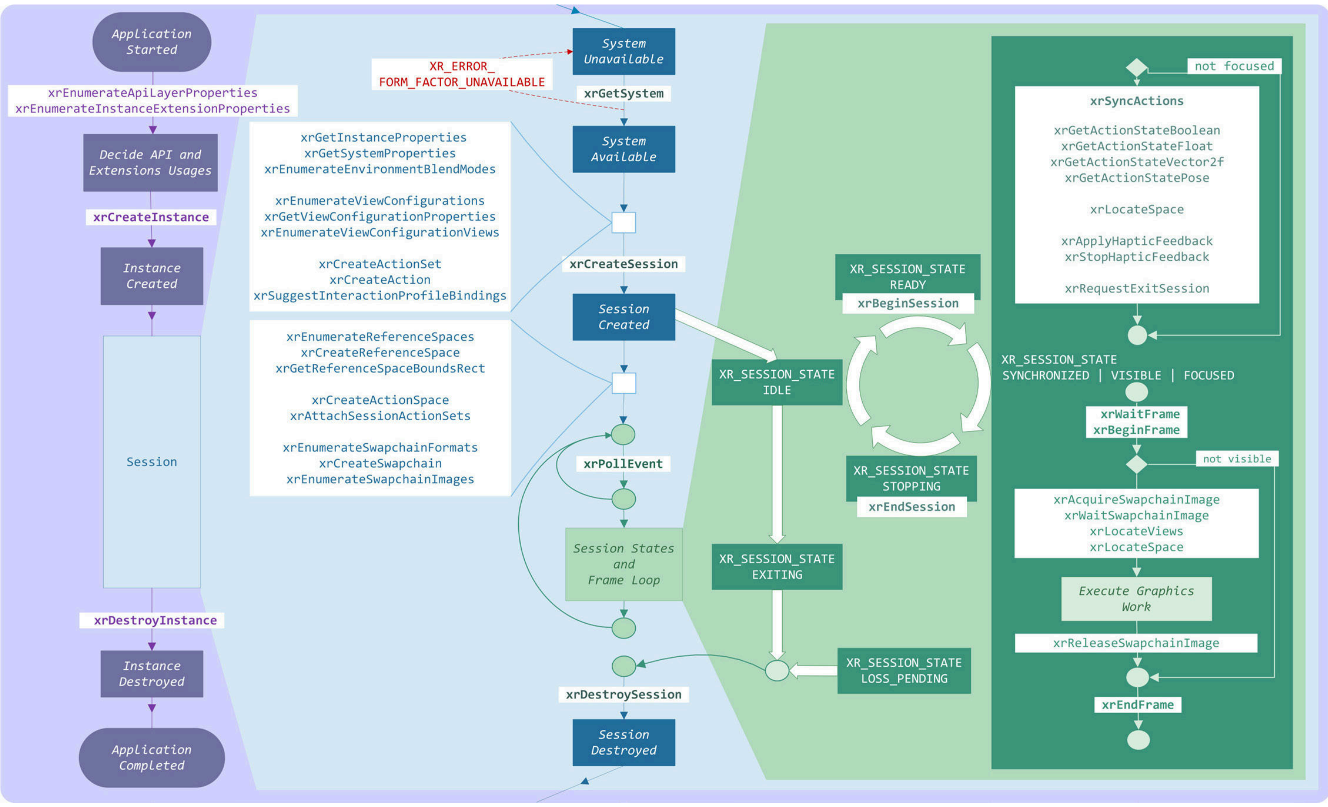
In order to create an OpenXR-compatible application, a precise suite of operations must be followed (see Figure 1 for details):

1. Available extensions are fetched to see if the ones needed are present (for instance, the extension that tells OpenXR to use the OpenGL graphics API, the debug utils extension…)
2. An "instance" is created with the application informations and extensions list. Future methods will use this instance.

[11] https://pyopengl.sourceforge.net/
[12] https://raphlinus.github.io/ui/graphics/gpu/2021/10/22/swapchain-frame-pacing.html

Figure 1 — Lifecycle of an OpenXR application

Not all functions are necessarily used.

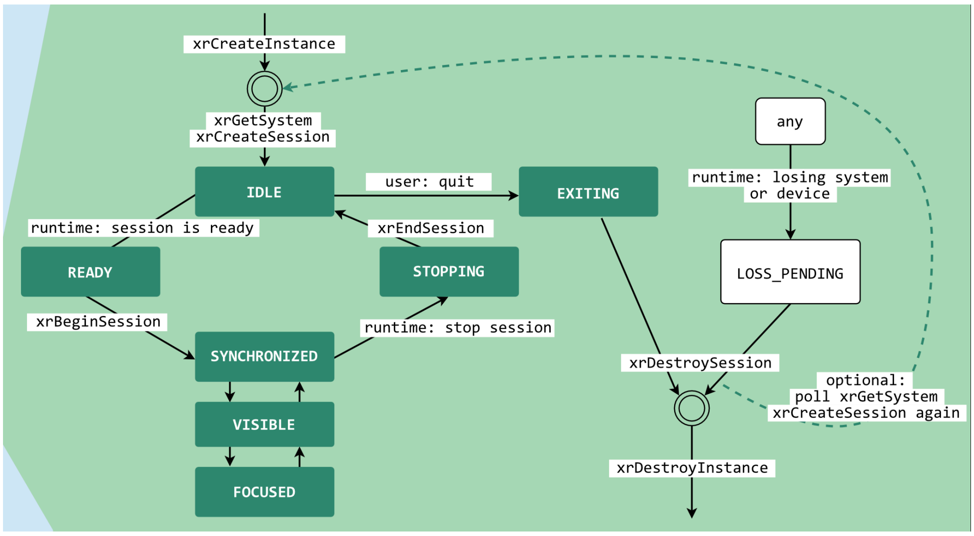


3. System information is fetched. At this point, we can use various methods to get configuration information about the view system (for instance, the "screen" size).
4. At this point, a graphic context must be created, although there is no need to have a window (unless we want to mirror what the user will see).
5. OpenXR is told which graphics API to use.
6. A "session" is created within the instance, with binding to the graphic context. Future methods will use this session.
7. The swapchain is created with specific color format, size, samples count …
8. A reference space is created to be used in head and controllers tracking.
9. A projection layer is created for the swapchain, containing the size and offset of the rectangles associated with each eye.
10. If needed, actions are created to interact with controllers.

At this point, the session is ready. We can enter the main loop:

1. Poll events from OpenXR. Update session according to the new state (see Figure 2 for details).
2. If the session is in state `READY`, `SYNCHRONIZED`, `FOCUSED` or `VISIBLE`:
   1. Wait for the next frame and when ready, begin it.
   2. Locate the views to get the eyes positions and update the projection accordingly.
   3. Acquire the swapchain image and render to it.
   4. Release the image and end the frame.

Figure 2 — Lifecycle of an OpenXR session



II - 3. MuJoCo

MuJoCo has a whole documentation chapter dedicated on visualization and rendering that is worth reading. It also contains tips for rendering to a VR headset. Other than what is explained in this chapter, there is not much knowledge of MuJoCo required to succeed in this project.

III. Implementation

Most of the OpenXR / OpenGL related code has been inspired by the gl_example pyopenxr example.

A Python file containing the whole source code is available in a Gist on GitHub. You can also find a version in the Section V.

We will now see step by step how everything work.

III - 1. OpenXR initialization

The related method is `_init_xr` *.*

The first part of the method is about creating the OpenXR instance with the required extension. A lot of lines (from line 44 to line 63) are dedicated to make debugging work; this is not interesting. Without it, we can re-write most of the method using one single method call:

```
34 self._xr_instance = xr.create_instance(xr.InstanceCreateInfo(
35     enabled_extension_names: [xr.KHR_OPENGL_ENABLE_EXTENSION_NAME],
36     application_info=xr.ApplicationInfo(
37         application_name=APP_NAME,
38         engine_name="pyopenxr",
39         engine_version=xr.PYOPENXR_CURRENT_API_VERSION,
40         api_version=xr.Version(1, 0, xr.XR_VERSION_PATCH)
41     )
42 ))
```

`xr.KHR_OPENGL_ENABLE_EXTENSION_NAME` is a constant describing the name of the `XR_KHR_opengl_enable` extension.[13] `APP_NAME` is a constant holding the name of our application that should be displayed to the user. It is defined at the beginning of the file. There is nothing really fancy here.

```
70         self._xr_system = xr.get_system(self._xr_instance, xr.SystemGetInfo(xr.FormFactor.HEAD_MOUNTED_DISPLAY))
71         assert xr.enumerate_view_configurations(self._xr_instance, self._xr_system)[0] == xr.ViewConfigurationType.PRIMARY_STEREO
72 
73         views_config = xr.enumerate_view_configuration_views(self._xr_instance, self._xr_system, xr.ViewConfigurationType.PRIMARY_STEREO)
74         assert len(views_config) == 2
75         assert views_config[0].recommended_image_rect_width == views_config[1].recommended_image_rect_width
76         assert views_config[0].recommended_image_rect_height == views_config[1].recommended_image_rect_height
77 
78         self._width, self._height = views_config[0].recommended_image_rect_width, views_config[0].recommended_image_rect_height
79         self._width_render = self._width * 2
```

[13] https://registry.khronos.org/OpenXR/specs/1.1-khr/html/xrspec.html#XR_KHR_opengl_enable

Here we do a bunch of checks to ensure everything is fine and avoid weird errors when rendering afterwards. The `enumerate_view_configuration_views` call at line 73 allows to get the image width and height, that we store. We also store an additional `_width_render` field that is simply the double of the normal width: it is the total width of our stereo render target.

The last part of this method is an ugly mixture of Python and C code to tell OpenXR to use OpenGL. We check that it contains no exception, and then we exit the method.

III - 2. OpenGL context and window

The related method is `_init_window`.

```
103         if not glfw.init():
104             raise RuntimeError("GLFW initialization failed")
105         glfw.window_hint(glfw.DOUBLEBUFFER, False)
106         glfw.window_hint(glfw.RESIZABLE, False)
107         glfw.window_hint(glfw.SAMPLES, 0) # no need for multisampling here, we will resolve ourselves
108         if not self._mirror_window:
109             glfw.window_hint(glfw.VISIBLE, False)
110         self._window_size = [self._width // 2, self._height // 2]
111         self._window = glfw.create_window(*self._window_size, APP_NAME, None, None)
112         if self._window is None:
113             raise RuntimeError("Failed to create GLFW window")
114         glfw.make_context_current(self._window)
```

We use GLFW to create a window. This window is not neccessarily visible (see line 109) but even without it, an OpenGL context will be created.
The `glfw.window_hint` calls are to set the parameters of the window/context to be created. Notably, we disable double-buffering (a swapchain of two images) because we do not really care about the window rendering quality and it can save us the time of image swapping.
The window is created with half the size of one eye, so it can fit on the screen (because most VR headset have huge resolutions).
On line 114, we tell glfw to make the OpenGL context current in the current thread.

It is worth noting that we do not give any information on the OpenGL version and profile we want. This is because MuJoCo specifically requires the *Compatibility profile* (see this MuJoCo page and Section IV - 3.).

III - 3. OpenXR configuration

The related method is `_prepare_xr`.

The first part of the method has nothing really worth noting: it first creates the `graphics_binding` object that must be passed to the OpenXR session and then it creates the actual session using some C/Python code.

```
        self._xr_swapchain = xr.create_swapchain(self._xr_session, xr.SwapchainCreateInfo(
            usage_flags=xr.SWAPCHAIN_USAGE_TRANSFER_DST_BIT | xr.SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT | xr.SWAPCHAIN_USAGE_SAMPLED_BIT,
```

```
147                 format=GL.GL_RGBA8,
148                 sample_count=1 if self._samples is None else self._samples,
149                 array_size=1,
150                 face_count=1,
151                 mip_count=1,
152                 width=self._width_render,
153                 height=self._height
154             ))
155             self._xr_swapchain_images = xr.enumerate_swapchain_images(self._xr_swapchain, xr.SwapchainImageOpenGLKHR)
```

At line 145 we create the swapchain for our stereo image, hence its width being `_width_render` (which is twice one eye's image width). The format has been selected arbitrarily, the best practice would be to enumerate the available formats and select the best one from here. The usage flags[14] contains:

- `TRANSFER_DST` because the swapchain image will be the destination of a pixel transfer operation (seen later)
- `COLOR_ATTACHMENT` because the swapchain image will have colored pixels on it (in most OpenXR applications this is the case)
- `SAMPLED` because the image can be multisampled

The line 155 is there to retrieve the list of images contained in the swapchain: we do it here once instead of doing it for each frame.

```
157             self._xr_projection_layer = xr.CompositionLayerProjection(
158                 # Default space params are okay: identity quaternion and zero vector. Let's use them.
159                 space=xr.create_reference_space(self._xr_session, xr.ReferenceSpaceCreateInfo()),
160                 views = [xr.CompositionLayerProjectionView(
161                     sub_image=xr.SwapchainSubImage(
162                         swapchain=self._xr_swapchain,
163                         image_rect=xr.Rect2Di(
164                             extent=xr.Extent2Di(self._width, self._height),
165                             offset=None if eye_index == 0 else xr.Offset2Di(x = self._width) # right eye offset
166                         )
167                     )
168                 ) for eye_index in range(2)]
169             )
170
171             self._xr_swapchain_fbo = GL.glGenFramebuffers(1)
```

Here we create the projection layer for the swapchain. It is the object that instructs the runtime where to put the rendered image in the virtual user space. It is done in 3 parts:

1. The reference space for the projection is created with the default options (the `STAGE` space type[15] and default orientation and position).

[14] https://registry.khronos.org/OpenXR/specs/1.1/man/html/XrSwapchainUsageFlagBits.html

2. One view per eye is created. Both views are attached to the same swapchain (the one we created earlier) but the `image_rect` of the right eye (which defines which part of the swapchain image is displayed) is offset to the right.
3. Finally, the whole projection layer is created.

Finally at line 171, we create an empty OpenGL framebuffer for the swapchain. We will use it later.

III - 4. MuJoCo preparation

The related method is `_prepare_mujoco`.

```
177         self._mj_model = mujoco.MjModel.from_xml_path("assets/balloons.xml")
178         self._mj_data = mujoco.MjData(self._mj_model)
179         self._mj_scene = mujoco.MjvScene(self._mj_model, 1000)
180         self._mj_scene.stereo = mujoco.mjtStereo.mjSTEREO_SIDEBYSIDE
181 
182         # We want the visualization properties set BEFORE creation of the context,
183         # otherwise we would have to call mjr_resizeOffscreen.
184         self._mj_model.vis.global_.offwidth = self._width_render
185         self._mj_model.vis.global_.offheight = self._height
186         self._mj_model.vis.quality.offsamples = 0 if self._samples is None else self._samples
187 
188         self._mj_context = mujoco.MjrContext(self._mj_model, mujoco.mjtFontScale.mjFONTSCALE_100)
189         self._mj_camera = mujoco._structs.MjvCamera()
190         self._mj_option = mujoco.MjvOption()
191         # We do NOT want to call mjv_defaultFreeCamera
192 
193         mujoco.mjv_defaultOption(self._mj_option)
```

The lines 177 and 178 are basic MuJoCo initialization. This can be done somewhere else in the code, even far sooner.

In this method, we mainly initialize the options of the MuJoCo scene and visualization objects so it can create its render context accordingly. This is done at line 190: when initializing the `MjrContext` object, it internally creates the offscreen framebuffer with the parameters we set at lines 184 to 186.

At this point, everything is ready to start the main loop.

III - 5. Frame loop - first part

The related methods are `loop`, `frame`, `_poll_xr_events` and `_start_xr_frame`.

The main loop structure looks like this:

```
loop:
    poll_events
    if should_quit:
```

[15]https://registry.khronos.org/OpenXR/specs/1.1/man/html/XrReferenceSpaceType.html

```
        stop

    if try_start_frame:
        make_a_frame
```

The `poll_events` part is made of this:

```
363         glfw.poll_events()
364         self._poll_xr_events()
365         if glfw.window_should_close(self._window):
366             self._should_quit = True
```

The `poll_events` method of glfw allows to know if the user wants to close the application on the desktop part (for instance, by closing the mirror window). We update the `_should_quit` field accordingly at line 366.
The `_poll_xr_events` method is fetching all events from the OpenXR instance and, if the event is a `SESSION_STATE_CHANGED` event, it does the following:

```
                    match self._xr_session_state:
                        case xr.SessionState.READY:
                            if not self._should_quit:
                                xr.begin_session(self._xr_session, xr.SessionBeginInfo(xr.ViewConfigurationType.PRIMARY_STEREO))
                        case xr.SessionState.STOPPING:
                            # means the session should end BUT it can start again later,
                            # this happens for instance when the user removes the headset
                            xr.end_session(self._xr_session)
                        case xr.SessionState.EXITING | xr.SessionState.LOSS_PENDING:
                            self._should_quit = True
```

If this is not clear to you, see the Session lifecycle at Figure 2.

If everything is fine and the visualization should not quit, we attempt to start the XR frame:

```
        if self._xr_session_state in [
            xr.SessionState.READY,
            xr.SessionState.FOCUSED,
            xr.SessionState.SYNCHRONIZED,
            xr.SessionState.VISIBLE,
        ]:
            self._xr_frame_state = xr.wait_frame(self._xr_session, xr.FrameWaitInfo())
            xr.begin_frame(self._xr_session, None)
            return True
        return False
```

If the session is in the right state to render a frame, it waits for the frame to be ready (so we do not render faster than the device refresh rate) and *then* it returns `True`.

III - 6. Frame loop - second part

The related methods are `frame`, `_update_mujoco`, `_update_views`, `render`, `_end_xr_frame`.

This only happens if the session is in the state to render a frame. This is how it goes:

```
            self._update_mujoco()
            self._update_views()
            if self._xr_frame_state.should_render:
                self._render()
            self._end_xr_frame()
```

The `_update_mujoco` method is really simple:

```
199         mujoco.mj_step(self._mj_model, self._mj_data)
200         mujoco.mjv_updateScene(self._mj_model, self._mj_data, self._mj_option, None, self._mj_camera, mujoco.mjtCatBit.mjCAT_ALL, self._mj_scene)
```

The line 199 could be done externally, it is not tied to the visualization: it only steps the physics. On the contrary, the call to `mjv_updateScene` at line 200 fetches geometries from the simulation data and stores it in the scene.

The `_update_views` method is the one that takes care of the head tracking. It goes in 3 parts: first, it fetches the `view_states` which contains, for each eye, its position, orientation and field of view. Then, it updates the projection layer accordingly and the two cameras in the MuJoCo scene to follow the eyes. Finally, it tells MuJoCo that all coordinates should be transformed in a certain way (otherwise, the world is tilted to the right).

The `_render` function is important and complex:

```
        # We first ask to acquire a swapchain image to render onto
        image_index = xr.acquire_swapchain_image(self._xr_swapchain, xr.SwapchainImageAcquireInfo())
        xr.wait_swapchain_image(self._xr_swapchain, xr.SwapchainImageWaitInfo(timeout=xr.INFINITE_DURATION))

        # Once we acquired it, we bind the image to our framebuffer object
        glfw.make_context_current(self._window)
        GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, self._xr_swapchain_fbo)
        GL.glFramebufferTexture2D(
            GL.GL_FRAMEBUFFER,
            GL.GL_COLOR_ATTACHMENT0,
            GL.GL_TEXTURE_2D if self._samples == None else GL.GL_TEXTURE_2D_MULTISAMPLE,
            self._xr_swapchain_images[image_index].image,
            0
        )
```

This first part prepares the framebuffer we created at the end of Section III - 3. by attaching the current swapchain image.

`glBindFramebuffer(GL_FRAMEBUFFER, fbo)` sets the framebuffer object as the one which will receive the read and draw operations.

`glFramebufferTexture2D` attaches the image as the first color attachement of the framebuffer object.

```
303         mujoco.mjr_setBuffer(mujoco.mjtFramebuffer.mjFB_OFFSCREEN, self._mj_context)
304         mujoco.mjr_render(mujoco.MjrRect(0, 0, self._width_render, self._height), self._mj_scene, self._mj_context)
```

The *real* rendering is done in the line 304. Afterwards, all is left is to copy the final image from MuJoCo's offscreen framebuffer to our own framebuffer, which has the swapchain image attached.

```
        # We copy what MuJoCo rendered on our framebuffer object
        GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, self._mj_context.offFBO)
        GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, self._xr_swapchain_fbo)
        GL.glBlitFramebuffer(
            0, 0,
            self._width_render, self._height,
            0, 0,
            self._width_render, self._height,
            GL.GL_COLOR_BUFFER_BIT,
            GL.GL_NEAREST
```

The first two instructions are to set which framebuffer will be read from and which one will be drawn on.

`glBlitFramebuffer` is an instruction to "copy" the pixels (the color ones in our case) from the read framebuffer to the draw one. Both framebuffers color attachements have the same size, so we put the same rectangle twice.

The rest of the method is made to downsample the rendered image and then copy it to our mirror window (if needed).

IV. Enhancements

IV - 1. Real-time simulation

For now, the code does 1 simulation step per render frame. However, due to synchronization made by OpenXR, one frame cannot be *shorter* than what it is supposed to be, so the framerate does not exceed the refresh rate of the device (for instance, 80Hz for the Oculus Rift S). This means that the simulation will update 80 times per second, no more. If the timestep set in the MuJoCo is not set to 1/80 of a second, the simulation will not be in "real-time".

To fix that, there are two options:

- The easy one is to change the timestep of your MuJoCo model to match the refresh rate. For the Oculus Rift S, you would set the timestep to $1/80 = 0.0125s$. This however is not ideal because some simulations will not be stable at such a large timestep.
- The harder one is to change the code to do, for each frame, the amount of simulation steps needed to advance the same amount of time the frame should durate. For a frame duration of $\Delta t_{\text{frame}} = 1/f$ and a timestep of Δt_{sim}, you would advance for

  $$n_{\text{steps}} = \left\lfloor \frac{\Delta t_{\text{frame}}}{\Delta t_{\text{sim}}} \right\rfloor$$

  You can get the frame duration using `_xr_frame_state.predicted_display_period` (in nanoseconds).

IV - 2. Hand tracking

Hand tracking is not included in the demo file, because it is tied in how the hand is represented in the MJCF. However, here are the basic steps to implement it:

- In the MuJoCo model, add a mocap body that will receive the hand position.
- For OpenXR, we have to create an *action* that will receive data from the controller. To do that, in the Python program after the `_prepare_xr` step, add another step which follow the same steps as in this example.
- To get the position and orientation in each frame, use thie code:

```
xr.sync_actions(self._xr_session, xr.ActionsSyncInfo(active_action_sets = ctypes.pointer(xr.ActiveActionSet(
    action_set=self._action_set,
    subaction_path=xr.NULL_PATH # wildcard to get all actions
))))
space_location = xr.locate_space(
    space=self._action_space,
    base_space=self._xr_projection_layer.space,
    time=self._xr_frame_state.predicted_display_time
)
if (space_location.location_flags & xr.SPACE_LOCATION_POSITION_VALID_BIT
    and space_location.location_flags & xr.SPACE_LOCATION_ORIENTATION_VALID_BIT):

    hand_pos = numpy.zeros(3)
    hand_rot = numpy.zeros(4)
```

```
    orientation = [space_location.pose.orientation[3], *space_location.pose.orientation[3]]
    mujoco.mjv_room2model(hand_pos, hand_rot, list(space_location.pose.position), orientation, self._mj_scene)
    return hand_pos, hand_rot
```

The important part here is to remember to change the quaternion format to the MuJoCo one (w, x, y, z) and to call `mjv_room2model` to automatically apply the transformations defined in `_update_views` to the pose.

IV - 3. Note on the `ContextObject` provided by *pyopenxr*

The *pyopenxr* bindings provide a pre-made class to handle most of the instance, session and swapchain work and let us focus on the interesting parts. This class is named `ContextObject` . However, it is not suitable for use in our case for two reasons:

- `ContextObject` creates one swapchain per eye, whereas we want one big swapchain containing both eyes (because this is how we want MuJoCo to render).
  - This issue could have been bypassed by creating a new swapchain ourselves and not using the premade ones. However, we loose a big part of the advantage of using this class in the first hand: simplicity.
- `ContextObject` uses an internal `OpenGLGraphics` class that handles a lot of rendering-related code. However, this class initializes the OpenGL context with the version 4.5 and *Core profile*. As we saw earlier, MuJoCo requires the *Compatibility profile*.
  - This is not bypassable without recompiling all of *pyopenxr*.

For those reasons, we made every OpenXR-related code from the ground up.

V. Annex - Source Code

```python
import xr
import mujoco
import glfw
import platform
import ctypes
import numpy
from OpenGL import GL
from typing import Optional

APP_NAME = "MuJoCo XR Viewer"
FRUSTUM_NEAR = 0.05
FRUSTUM_FAR = 50

class MujocoXRViewer:
    def __init__(self, mirror_window = False, debug = False, samples: Optional[int] = None):
        self._mirror_window = mirror_window
        self._debug = debug
        self._samples = samples
        self._should_quit = False

    def __enter__(self):
        self._init_xr()
        self._init_window()
        self._prepare_xr()
        self._prepare_mujoco()
        return self

    def _init_xr(self):
        """
        Initializes the OpenXR environment prior to session creation.

        Also fetches informations about the setup, most importantly the render size.
        """
        extensions = [xr.KHR_OPENGL_ENABLE_EXTENSION_NAME]
        instance_create_info = xr.InstanceCreateInfo(
            application_info=xr.ApplicationInfo(
                application_name=APP_NAME,
                engine_name="pyopenxr",
                engine_version=xr.PYOPENXR_CURRENT_API_VERSION,
                api_version=xr.Version(1, 0, xr.XR_VERSION_PATCH)
            )
        )

        if self._debug:
            def debug_callback_py(severity, _type, data, _user_data):
                print(severity, f"{data.contents.function_name.decode()}: {data.contents.message.decode()}")
                return True

            debug_messenger = xr.DebugUtilsMessengerCreateInfoEXT(
                message_severities=
                    xr.DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT
                    | xr.DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT
                    | xr.DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT
                    | xr.DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT,
                message_types=
                    xr.DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT
                    | xr.DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT
                    | xr.DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT
                    | xr.DEBUG_UTILS_MESSAGE_TYPE_CONFORMANCE_BIT_EXT,
                user_callback=xr.PFN_xrDebugUtilsMessengerCallbackEXT(debug_callback_py)
            )
            instance_create_info.next = ctypes.cast(ctypes.pointer(debug_messenger), ctypes.c_void_p)
            extensions.append(xr.EXT_DEBUG_UTILS_EXTENSION_NAME)

        instance_create_info.enabled_extension_names = extensions
        self._xr_instance = xr.create_instance(instance_create_info)

        # The following fetches important informations about the setup
        # (mainly rendering size)
        self._xr_system = xr.get_system(self._xr_instance, xr.SystemGetInfo(xr.FormFactor.HEAD_MOUNTED_DISPLAY))
        assert xr.enumerate_view_configurations(self._xr_instance, self._xr_system)[0] == xr.ViewConfigurationType.PRIMARY_STEREO

        views_config = xr.enumerate_view_configuration_views(self._xr_instance, self._xr_system, xr.ViewConfigurationType.PRIMARY_STEREO)
```

```python
        assert len(views_config) == 2
        assert views_config[0].recommended_image_rect_width == views_config[1].recommended_image_rect_width
        assert views_config[0].recommended_image_rect_height == views_config[1].recommended_image_rect_height

        self._width, self._height = views_config[0].recommended_image_rect_width, views_config[0].recommended_image_rect_height
        self._width_render = self._width * 2

        pxrGetOpenGLGraphicsRequirementsKHR = ctypes.cast(
            xr.get_instance_proc_addr(
                self._xr_instance,
                "xrGetOpenGLGraphicsRequirementsKHR",
            ),
            xr.PFN_xrGetOpenGLGraphicsRequirementsKHR
        )
        graphics_result = pxrGetOpenGLGraphicsRequirementsKHR(
            self._xr_instance,
            self._xr_system,
            ctypes.byref(xr.GraphicsRequirementsOpenGLKHR())
        )
        graphics_result = xr.exception.check_result(xr.Result(graphics_result))
        if graphics_result.is_exception():
            raise graphics_result

    def _init_window(self):
        """
        Initializes the GLFW window (and make it hidden if mirrored mode is disabled).

        Creates the OpenGL context that will be used.
        """
        if not glfw.init():
            raise RuntimeError("GLFW initialization failed")
        glfw.window_hint(glfw.DOUBLEBUFFER, False)
        glfw.window_hint(glfw.RESIZABLE, False)
        glfw.window_hint(glfw.SAMPLES, 0) # no need for multisampling here, we will resolve ourselves
        if not self._mirror_window:
            glfw.window_hint(glfw.VISIBLE, False)
        self._window_size = [self._width // 2, self._height // 2]
        self._window = glfw.create_window(*self._window_size, APP_NAME, None, None)
        if self._window is None:
            raise RuntimeError("Failed to create GLFW window")
        glfw.make_context_current(self._window)
        # Attempt to disable vsync on the desktop window or
        # it will interfere with the OpenXR frame loop timing
        glfw.swap_interval(0)

    def _prepare_xr(self):
        """
        Creates the OpenXR session and prepares everything to launch the frames loop.
        """
        if platform.system() == 'Windows':
            from OpenGL import WGL
            graphics_binding = xr.GraphicsBindingOpenGLWin32KHR()
            graphics_binding.h_dc = WGL.wglGetCurrentDC()
            graphics_binding.h_glrc = WGL.wglGetCurrentContext()
        else:
            from OpenGL import GLX
            graphics_binding = xr.GraphicsBindingOpenGLXlibKHR()
            graphics_binding.x_display = GLX.glXGetCurrentDisplay()
            graphics_binding.glx_context = GLX.glXGetCurrentContext()
            graphics_binding.glx_drawable = GLX.glXGetCurrentDrawable()

        self._xr_session = xr.create_session(
            self._xr_instance,
            xr.SessionCreateInfo(
                0,
                self._xr_system,
                next=ctypes.cast(ctypes.pointer(graphics_binding), ctypes.c_void_p)
            )
        )
        self._xr_session_state = xr.SessionState.IDLE

        self._xr_swapchain = xr.create_swapchain(self._xr_session, xr.SwapchainCreateInfo(
            usage_flags=xr.SWAPCHAIN_USAGE_TRANSFER_DST_BIT | xr.SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT | xr.SWAPCHAIN_USAGE_SAMPLED_BIT,
            format=GL.GL_RGBA8,
            sample_count=1 if self._samples is None else self._samples,
            array_size=1,
```

```
                face_count=1,
                mip_count=1,
                width=self._width_render,
                height=self._height
            ))
            self._xr_swapchain_images = xr.enumerate_swapchain_images(self._xr_swapchain, xr.SwapchainImageOpenGLKHR)

            self._xr_projection_layer = xr.CompositionLayerProjection(
                # Default space params are okay: identity quaternion and zero vector. Let's use them.
                space=xr.create_reference_space(self._xr_session, xr.ReferenceSpaceCreateInfo()),
                views = [xr.CompositionLayerProjectionView(
                    sub_image=xr.SwapchainSubImage(
                        swapchain=self._xr_swapchain,
                        image_rect=xr.Rect2Di(
                            extent=xr.Extent2Di(self._width, self._height),
                            offset=None if eye_index == 0 else xr.Offset2Di(x = self._width) # right eye offset
                        )
                    )
                ) for eye_index in range(2)]
            )

            self._xr_swapchain_fbo = GL.glGenFramebuffers(1)

        def _prepare_mujoco(self):
            """
            Prepares the MuJoCo environment.
            """
            self._mj_model = mujoco.MjModel.from_xml_path("assets/balloons.xml")
            self._mj_data = mujoco.MjData(self._mj_model)
            self._mj_scene = mujoco.MjvScene(self._mj_model, 1000)
            self._mj_scene.stereo = mujoco.mjtStereo.mjSTEREO_SIDEBYSIDE

            # We want the visualization properties set BEFORE creation of the context,
            # otherwise we would have to call mjr_resizeOffscreen.
            self._mj_model.vis.global_.offwidth = self._width_render
            self._mj_model.vis.global_.offheight = self._height
            self._mj_model.vis.quality.offsamples = 0 if self._samples is None else self._samples

            self._mj_context = mujoco.MjrContext(self._mj_model, mujoco.mjtFontScale.mjFONTSCALE_100)
            self._mj_camera = mujoco._structs.MjvCamera()
            self._mj_option = mujoco.MjvOption()
            # We do NOT want to call mjv_defaultFreeCamera

            mujoco.mjv_defaultOption(self._mj_option)

        def _update_mujoco(self):
            """
            Updates MuJoCo for one frame.
            """
            mujoco.mj_step(self._mj_model, self._mj_data)
            mujoco.mjv_updateScene(self._mj_model, self._mj_data, self._mj_option, None, self._mj_camera, mujoco.mjtCatBit.mjCAT_ALL, self._mj_scene)

        def _start_xr_frame(self):
            """
            Starts a frame in the OpenXR environment.

            Returns:
                bool: whether or not we should update the scene and maybe render it.
            """
            if self._xr_session_state in [
                xr.SessionState.READY,
                xr.SessionState.FOCUSED,
                xr.SessionState.SYNCHRONIZED,
                xr.SessionState.VISIBLE,
            ]:
                self._xr_frame_state = xr.wait_frame(self._xr_session, xr.FrameWaitInfo())
                xr.begin_frame(self._xr_session, None)
                return True
            return False

        def _end_xr_frame(self):
            xr.end_frame(self._xr_session, xr.FrameEndInfo(
                self._xr_frame_state.predicted_display_time,
                xr.EnvironmentBlendMode.OPAQUE,
                layers=[ctypes.byref(self._xr_projection_layer)] if self._xr_frame_state.should_render else []
            ))

        def _poll_xr_events(self):
```

```python
        while True:
            try:
                event_buffer = xr.poll_event(self._xr_instance)
                event_type = xr.StructureType(event_buffer.type)
                if event_type == xr.StructureType.EVENT_DATA_SESSION_STATE_CHANGED:
                    event = ctypes.cast(
                        ctypes.byref(event_buffer),
                        ctypes.POINTER(xr.EventDataSessionStateChanged)).contents
                    self._xr_session_state = xr.SessionState(event.state)
                    match self._xr_session_state:
                        case xr.SessionState.READY:
                            if not self._should_quit:
                                xr.begin_session(self._xr_session, xr.SessionBeginInfo(xr.ViewConfigurationType.PRIMARY_STEREO))
                        case xr.SessionState.STOPPING:
                            # means the session should end BUT it can start again later,
                            # this happens for instance when the user removes the headset
                            xr.end_session(self._xr_session)
                        case xr.SessionState.EXITING | xr.SessionState.LOSS_PENDING:
                            self._should_quit = True
            except xr.EventUnavailable:
                break # We got all events

    def _update_views(self):
        _, view_states = xr.locate_views(self._xr_session, xr.ViewLocateInfo(
            xr.ViewConfigurationType.PRIMARY_STEREO,
            self._xr_frame_state.predicted_display_time,
            self._xr_projection_layer.space,
        ))
        for eye_index, view_state in enumerate(view_states):
            self._xr_projection_layer.views[eye_index].fov = view_state.fov
            self._xr_projection_layer.views[eye_index].pose = view_state.pose

            cam = self._mj_scene.camera[eye_index]
            cam.pos = list(view_state.pose.position)
            cam.frustum_near = FRUSTUM_NEAR
            cam.frustum_far = FRUSTUM_FAR
            cam.frustum_bottom = numpy.tan(view_state.fov.angle_down) * FRUSTUM_NEAR
            cam.frustum_top = numpy.tan(view_state.fov.angle_up) * FRUSTUM_NEAR
            cam.frustum_center = 0.5 * (numpy.tan(view_state.fov.angle_left) + numpy.tan(view_state.fov.angle_right)) * FRUSTUM_NEAR
            # no need to set left/right as it will be computed using center

            rot_quat = list(view_state.pose.orientation)
            # Guess what? OpenXR quaternions are in form (x, y, z, w)
            # while MuJoCo quaternions are in form (w, x, y, z)...
            rot_quat = [rot_quat[3], *rot_quat[0:3]]

            forward, up = numpy.zeros(3), numpy.zeros(3)
            mujoco.mju_rotVecQuat(forward, [0, 0, -1], rot_quat)
            mujoco.mju_rotVecQuat(up, [0, 1, 0], rot_quat)
            cam.forward, cam.up = forward.tolist(), up.tolist()

        self._mj_scene.enabletransform = True
        self._mj_scene.rotate[0] = numpy.cos(0.25 * numpy.pi)
        self._mj_scene.rotate[1] = numpy.sin(-0.25 * numpy.pi)

    def _render(self):
        """
        Renders the scene in the swapchain and eventually mirrors it on the window if needed.
        """
        # We first ask to acquire a swapchain image to render onto
        image_index = xr.acquire_swapchain_image(self._xr_swapchain, xr.SwapchainImageAcquireInfo())
        xr.wait_swapchain_image(self._xr_swapchain, xr.SwapchainImageWaitInfo(timeout=xr.INFINITE_DURATION))

        # Once we acquired it, we bind the image to our framebuffer object
        glfw.make_context_current(self._window)
        GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, self._xr_swapchain_fbo)
        GL.glFramebufferTexture2D(
            GL.GL_FRAMEBUFFER,
            GL.GL_COLOR_ATTACHMENT0,
            GL.GL_TEXTURE_2D if self._samples == None else GL.GL_TEXTURE_2D_MULTISAMPLE,
            self._xr_swapchain_images[image_index].image,
            0
        )

        # We ask MuJoCo to render on its own offscreen framebuffer
        mujoco.mjr_setBuffer(mujoco.mjtFramebuffer.mjFB_OFFSCREEN, self._mj_context)
```

```
        mujoco.mjr_render(mujoco.MjrRect(0, 0, self._width_render, self._height), self._mj_scene, self._mj_context)

        # We copy what MuJoCo rendered on our framebuffer object
        GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, self._mj_context.offFBO)
        GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, self._xr_swapchain_fbo)
        GL.glBlitFramebuffer(
            0, 0,
            self._width_render, self._height,
            0, 0,
            self._width_render, self._height,
            GL.GL_COLOR_BUFFER_BIT,
            GL.GL_NEAREST
        )

        if self._mirror_window:
            # We copy the data from the MuJoCo buffer to the window one (0 is the default window fbo)
            if self._samples is not None:
                # We first resolve multi-sample if needed
                GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, self._mj_context.offFBO_r)
                GL.glBlitFramebuffer(
                    0, 0,
                    self._width_render, self._height,
                    0, 0,
                    self._width_render, self._height,
                    GL.GL_COLOR_BUFFER_BIT,
                    GL.GL_NEAREST
                )
                GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, self._mj_context.offFBO_r)

            # We then copy the data to the window
            GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, 0)
            GL.glBlitFramebuffer(
                0, 0,
                self._width, self._height, # one eye only (left)
                0, 0,
                *self._window_size,
                GL.GL_COLOR_BUFFER_BIT,
                0x90BA # EXT_framebuffer_multisample_blit_scaled, SCALED_RESOLVE_FASTEST_EXT
            )
        xr.release_swapchain_image(self._xr_swapchain, xr.SwapchainImageReleaseInfo())

    def __exit__(self, exc_type, exc_value, traceback):
        if self._window is not None:
            glfw.make_context_current(self._window)
            if self._xr_swapchain_fbo is not None:
                GL.glDeleteFramebuffers(1, [self._xr_swapchain_fbo])
                self._xr_swapchain_fbo = None
            glfw.terminate()
        if self._xr_swapchain is not None:
            xr.destroy_swapchain(self._xr_swapchain)
        if self._xr_session is not None:
            xr.destroy_session(self._xr_session)
        if self._xr_instance is not None:
            # # may break on Linux SteamVR
            # xr.destroy_instance(self._xr_instance)
            pass # does not seem to work
        glfw.terminate()

    def frame(self):
        glfw.poll_events()
        self._poll_xr_events()
        if glfw.window_should_close(self._window):
            self._should_quit = True

        if self._should_quit:
            return

        if self._start_xr_frame():
            self._update_mujoco()
            self._update_views()
            if self._xr_frame_state.should_render:
                self._render()
            self._end_xr_frame()

    def loop(self):
        while not self._should_quit:
            self.frame()
```