

MuJoCoXR

Linking MuJoCo and a VR environment

2024-07-17

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-time², a full Renderer Python class³ that can be really useful for notebooks and, if nothing else is suitable, API functions to render directly in an OpenGL context⁴.

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” application⁵. Nevertheless, it is still not the best choice if you want to fully immerse people in your simulation⁶. One possible solution is to use the Unity Plug-in for MuJoCo⁷ 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 requires 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 available¹⁰. 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>

⁶The “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.

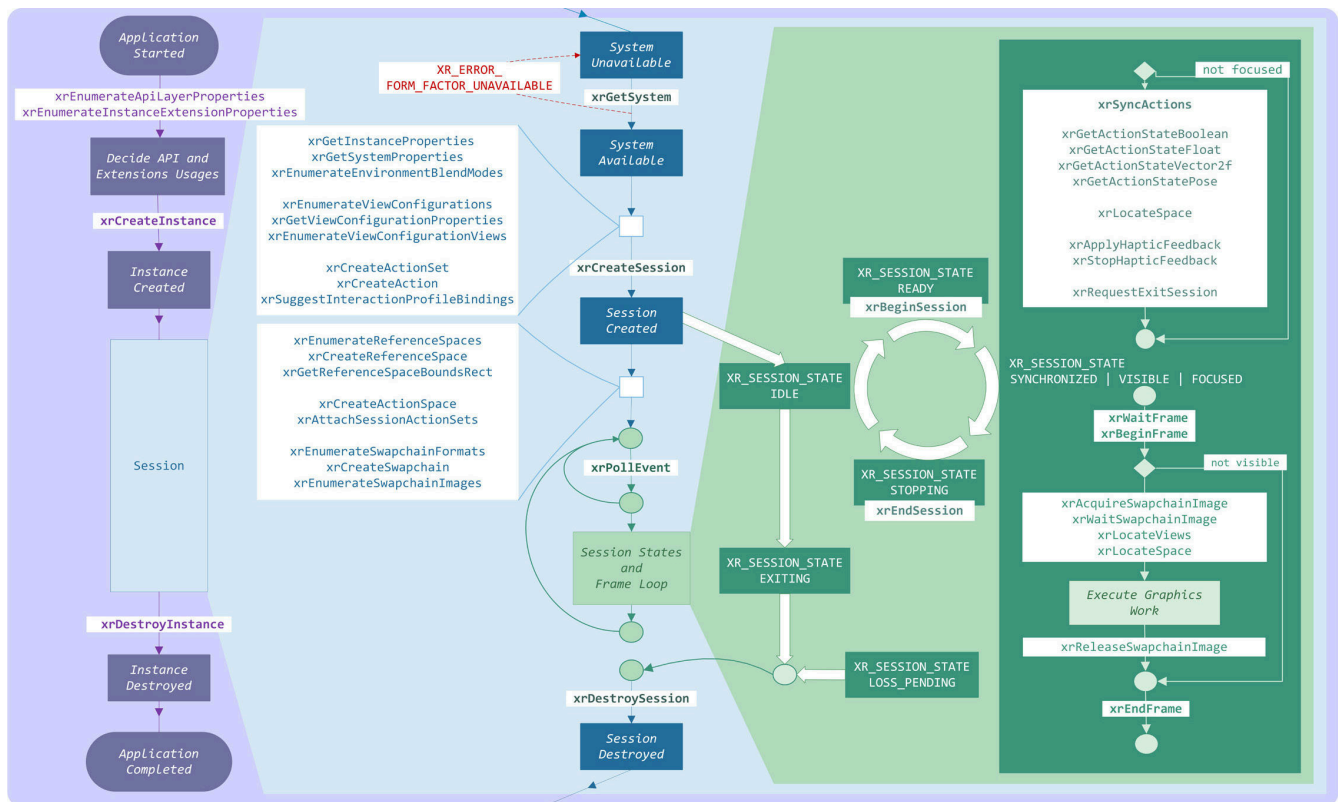
In order to create an OpenXR-compatible application, a precise suite of operations must be followed (see Figure 1 for details):

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.

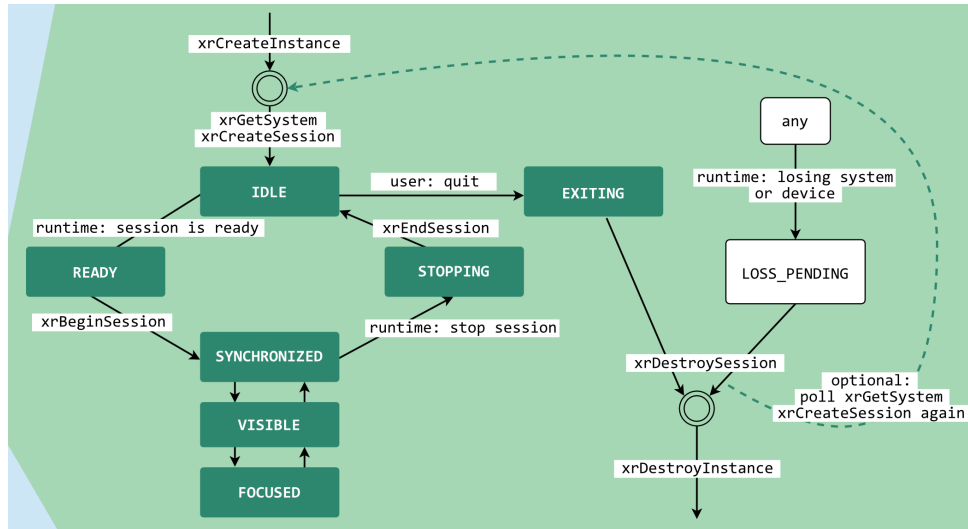


3. System information is fetched. At this point, we can use various methods to get configuration information about the view system (for instance, the “screen” size).
4. At this point, a graphic context must be created, although there is no need to have a window (unless we want to mirror what the user 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 IV.

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,
71     xr.SystemGetInfo(xr.FormFactor.HEAD_MOUNTED_DISPLAY))
72     assert xr.enumerate_view_configurations(self._xr_instance,
73 self._xr_system)[0] == xr.ViewConfigurationType.PRIMARY_STEREO
74
75     views_config =
76     xr.enumerate_view_configuration_views(self._xr_instance, self._xr_system,
77     xr.ViewConfigurationType.PRIMARY_STEREO)
78     assert len(views_config) == 2
79     assert views_config[0].recommended_image_rect_width ==
views_config[1].recommended_image_rect_width
80     assert views_config[0].recommended_image_rect_height ==
views_config[1].recommended_image_rect_height
81
82     self._width, self._height =
83     views_config[0].recommended_image_rect_width,
84     views_config[0].recommended_image_rect_height
85     self._width_render = self._width * 2

```

¹³https://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
108     here, we will resolve ourselves
109     if not self._mirror_window:
110         glfw.window_hint(glfw.VISIBLE, False)
111         self._window_size = [self._width // 2, self._height // 2]
112         self._window = glfw.create_window(*self._window_size, APP_NAME,
113         None, None)
114         if self._window is None:
115             raise RuntimeError("Failed to create GLFW window")
116         glfw.make_context_current(self._window)

```

We use GLFW to create a window. This window is not necessarily 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 III - 7.).

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.

```

145         self._xr_swapchain = xr.create_swapchain(self._xr_session,
146         xr.SwapchainCreateInfo(
147             usage_flags=xr.SWAPCHAIN_USAGE_TRANSFER_DST_BIT |
148             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 flags¹⁴ 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 run-time 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 type¹⁵ 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_SIDE_BY_SIDE
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:

```

237             match self._xr_session_state:
238                 case xr.SessionState.READY:
239                     if not self._should_quit:
240                         xr.begin_session(self._xr_session,
xr.SessionBeginInfo(xr.ViewConfigurationType.PRIMARY_STEREO))
241                 case xr.SessionState.STOPPING:
242                     # means the session should end BUT it can
start again later,
243                     # this happens for instance when the user
removes the headset
244                     xr.end_session(self._xr_session)
245                 case xr.SessionState.EXITING |
xr.SessionState.LOSS_PENDING:
246                     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:

```

209         if self._xr_session_state in [
210             xr.SessionState.READY,
211             xr.SessionState.FOCUSED,
212             xr.SessionState.SYNCHRONIZED,
213             xr.SessionState.VISIBLE,
214         ]:
215             self._xr_frame_state = xr.wait_frame(self._xr_session,
xr.FrameWaitInfo())
216             xr.begin_frame(self._xr_session, None)
217             return True
218         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:

```

372         self._update_mujoco()
373         self._update_views()
374         if self._xr_frame_state.should_render:
375             self._render()
376             self._end_xr_frame()

```

The `_update_mujoco` method is really simple:

```

199         mujoco.mj_step(self._mj_model, self._mj_data)
        mujoco.mjv_updateScene(self._mj_model, self._mj_data,
200 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:

```

287         # We first ask to acquire a swapchain image to render onto
288         image_index = xr.acquire_swapchain_image(self._xr_swapchain,
289 xr.SwapchainImageAcquireInfo())
290         xr.wait_swapchain_image(self._xr_swapchain,
291 xr.SwapchainImageWaitInfo(timeout=xr.INFINITE_DURATION))
292
293         # Once we acquired it, we bind the image to our framebuffer
294         object
295         glfw.make_context_current(self._window)
296         GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, self._xr_swapchain_fbo)
297         GL.glFramebufferTexture2D(
298             GL.GL_FRAMEBUFFER,
299             GL.GL_COLOR_ATTACHMENT0,
300             GL.GL_TEXTURE_2D if self._samples == None else
301             GL.GL_TEXTURE_2D_MULTISAMPLE,
302             self._xr_swapchain_images[image_index].image,
303             0
304         )

```

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 attachment of the framebuffer object.

```

303         mujoco.mj_r_setBuffer(mujoco.mjtFramebuffer.mjFB_OFFSCREEN,
self._mj_context)
304         mujoco.mj_r_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.

```

306         # We copy what MuJoCo rendered on our framebuffer object
307         GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER,
self._mj_context.offFB0)
308         GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER,
self._xr_swapchain_fbo)
309         GL.glBlitFramebuffer(
310             0, 0,
311             self._width_render, self._height,
312             0, 0,
313             self._width_render, self._height,
314             GL.GL_COLOR_BUFFER_BIT,
315             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 attachments 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).

III - 7. 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 lose 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.

IV. Annex - Source Code

```

1  import xr
2  import mujoco
3  import glfw
4  import platform
5  import ctypes
6  import numpy
7  from OpenGL import GL
8  from typing import Optional
9
10 APP_NAME = "MuJoCo XR Viewer"
11 FRUSTUM_NEAR = 0.05
12 FRUSTUM_FAR = 50
13
14 class MujocoXRViewer:
15     def __init__(self, mirror_window = False, debug = False, samples: Optional[int] = None):
16         self._mirror_window = mirror_window
17         self._debug = debug
18         self._samples = samples
19         self._should_quit = False
20
21     def __enter__(self):
22         self._init_xr()
23         self._init_window()
24         self._prepare_xr()
25         self._prepare_mujoco()
26         return self
27
28     def _init_xr(self):
29         """
30         Initializes the OpenXR environment prior to session creation.
31
32         Also fetches informations about the setup, most importantly the render size.
33         """
34         extensions = [xr.KHR_OPENGL_ENABLE_EXTENSION_NAME]
35         instance_create_info = xr.InstanceCreateInfo(
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         )
43
44         if self._debug:
45             def debug_callback_py(severity, _type, data, _user_data):
46                 print(severity, f"{data.contents.function_name.decode()}: {data.contents.message.decode()}")
47                 return True
48
49             debug_messenger = xr.DebugUtilsMessengerCreateInfoEXT(
50                 message_severities=
51                     xr.DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT
52                     | xr.DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT
53                     | xr.DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT
54                     | xr.DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT,
55                 message_types=
56                     xr.DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT
57                     | xr.DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT
58                     | xr.DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT
59                     | xr.DEBUG_UTILS_MESSAGE_TYPE_CONFORMANCE_BIT_EXT,
60                 user_callback=xr.PFN_xrDebugUtilsMessengerCallbackEXT(debug_callback_py)
61             )
62             instance_create_info.next = ctypes.cast(ctypes.pointer(debug_messenger), ctypes.c_void_p)
63             extensions.append(xr.EXT_DEBUG_UTILS_EXTENSION_NAME)
64
65         instance_create_info.enabled_extension_names = extensions
66         self._xr_instance = xr.create_instance(instance_create_info)
67
68         # The following fetches important informations about the setup
69         # (mainly rendering size)
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] ==
72         xr.ViewConfigurationType.PRIMARY_STEREO
73
74         views_config = xr.enumerate_view_configuration_views(self._xr_instance, self._xr_system,
75         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
80
81     pxrGetOpenGLGraphicsRequirementsKHR = ctypes.cast(
82         xr.get_instance_proc_addr(
83             self._xr_instance,
84             "xrGetOpenGLGraphicsRequirementsKHR",
85         ),
86         xr.PFN_xrGetOpenGLGraphicsRequirementsKHR
87     )
88     graphics_result = pxrGetOpenGLGraphicsRequirementsKHR(
89         self._xr_instance,
90         self._xr_system,
91         ctypes.byref(xr.GraphicsRequirementsOpenGLKHR())
92     )
93     graphics_result = xr.exception.check_result(xr.Result(graphics_result))
94     if graphics_result.is_exception():
95         raise graphics_result
96
97 def _init_window(self):
98     """
99     Initializes the GLFW window (and make it hidden if mirrored mode is disabled).
100
101     Creates the OpenGL context that will be used.
102     """
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)
115     # Attempt to disable vsync on the desktop window or
116     # it will interfere with the OpenXR frame loop timing
117     glfw.swap_interval(0)
118
119 def _prepare_xr(self):
120     """
121     Creates the OpenXR session and prepares everything to launch the frames loop.
122     """
123     if platform.system() == 'Windows':
124         from OpenGL import WGL
125         graphics_binding = xr.GraphicsBindingOpenGLWin32KHR()
126         graphics_binding.h_dc = WGL.wglGetCurrentDC()
127         graphics_binding.h_glr = WGL.wglGetCurrentContext()
128     else:
129         from OpenGL import GLX
130         graphics_binding = xr.GraphicsBindingOpenGLXlibKHR()
131         graphics_binding.x_display = GLX.glXGetCurrentDisplay()
132         graphics_binding.glx_context = GLX.glXGetCurrentContext()
133         graphics_binding.glx_drawable = GLX.glXGetCurrentDrawable()
134
135     self._xr_session = xr.create_session(
136         self._xr_instance,
137         xr.SessionCreateInfo(
138             0,
139             self._xr_system,
140             next=ctypes.cast(ctypes.pointer(graphics_binding), ctypes.c_void_p)
141         )
142     )
143     self._xr_session_state = xr.SessionState.IDLE
144
145     self._xr_swapchain = xr.create_swapchain(self._xr_session, xr.SwapchainCreateInfo(
146         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)
156
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)
172
173 def _prepare_mujoco(self):
174     """
175     Prepares the MuJoCo environment.
176     """
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_SIDE_BY_SIDE
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 mjr_defaultFreeCamera
192
193     mujoco.mjv_defaultOption(self._mj_option)
194
195 def _update_mujoco(self):
196     """
197     Updates MuJoCo for one frame.
198     """
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,
201         mujoco.mjtCatBit.mjCAT_ALL, self._mj_scene)
202
203 def _start_xr_frame(self):
204     """
205     Starts a frame in the OpenXR environment.
206
207     Returns:
208         bool: whether or not we should update the scene and maybe render it.
209     """
210     if self._xr_session_state in [
211         xr.SessionState.READY,
212         xr.SessionState.FOCUSED,
213         xr.SessionState.SYNCHRONIZED,
214         xr.SessionState.VISIBLE,
215     ]:
216         self._xr_frame_state = xr.wait_frame(self._xr_session, xr.FrameWaitInfo())
217         xr.begin_frame(self._xr_session, None)
218         return True
219     return False
220
221 def _end_xr_frame(self):
222     xr.end_frame(self._xr_session, xr.FrameEndInfo(
223         self._xr_frame_state.predicted_display_time,
224         xr.EnvironmentBlendMode.OPAQUE,
225         layers=[ctypes.byref(self._xr_projection_layer)] if self._xr_frame_state.should_render else []
226     ))
227
228 def _poll_xr_events(self):

```



```

228 while True:
229     try:
230         event_buffer = xr.poll_event(self._xr_instance)
231         event_type = xr.StructureType(event_buffer.type)
232         if event_type == xr.StructureType.EVENT_DATA_SESSION_STATE_CHANGED:
233             event = ctypes.cast(
234                 ctypes.byref(event_buffer),
235                 ctypes.POINTER(xr.EventDataSessionStateChanged)).contents
236             self._xr_session_state = xr.SessionState(event.state)
237             match self._xr_session_state:
238                 case xr.SessionState.READY:
239                     if not self._should_quit:
240                         xr.begin_session(self._xr_session,
xr.SessionBeginInfo(xr.ViewConfigurationType.PRIMARY_STEREO))
241                 case xr.SessionState.STOPPING:
242                     # means the session should end BUT it can start again later,
243                     # this happens for instance when the user removes the headset
244                     xr.end_session(self._xr_session)
245                 case xr.SessionState.EXITING | xr.SessionState.LOSS_PENDING:
246                     self._should_quit = True
247             except xr.EventUnavailable:
248                 break # We got all events
249
250 def _update_views(self):
251     _, view_states = xr.locate_views(self._xr_session, xr.ViewLocateInfo(
252         xr.ViewConfigurationType.PRIMARY_STEREO,
253         self._xr_frame_state.predicted_display_time,
254         self._xr_projection_layer.space,
255     ))
256     for eye_index, view_state in enumerate(view_states):
257         self._xr_projection_layer.views[eye_index].fov = view_state.fov
258         self._xr_projection_layer.views[eye_index].pose = view_state.pose
259
260         cam = self._mj_scene.camera[eye_index]
261         cam.pos = list(view_state.pose.position)
262         cam.frustum_near = FRUSTUM_NEAR
263         cam.frustum_far = FRUSTUM_FAR
264         cam.frustum_bottom = numpy.tan(view_state.fov.angle_down) * FRUSTUM_NEAR
265         cam.frustum_top = numpy.tan(view_state.fov.angle_up) * FRUSTUM_NEAR
266         cam.frustum_center = 0.5 * (numpy.tan(view_state.fov.angle_left) + numpy.tan(view_state.fov.angle_right)) *
FRUSTUM_NEAR
267         # no need to set left/right as it will be computed using center
268
269         rot_quat = list(view_state.pose.orientation)
270         # Guess what? OpenXR quaternions are in form (x, y, z, w)
271         # while MuJoCo quaternions are in form (w, x, y, z)...
272         rot_quat = [rot_quat[3], *rot_quat[0:3]]
273
274         forward, up = numpy.zeros(3), numpy.zeros(3)
275         mujoco.mju_rotVecQuat(forward, [0, 0, -1], rot_quat)
276         mujoco.mju_rotVecQuat(up, [0, 1, 0], rot_quat)
277         cam.forward, cam.up = forward.tolist(), up.tolist()
278
279     self._mj_scene.enabletransform = True
280     self._mj_scene.rotate[0] = numpy.cos(0.25 * numpy.pi)
281     self._mj_scene.rotate[1] = numpy.sin(-0.25 * numpy.pi)
282
283 def _render(self):
284     """
285     Renders the scene in the swapchain and eventually mirrors it on the window if needed.
286     """
287     # We first ask to acquire a swapchain image to render onto
288     image_index = xr.acquire_swapchain_image(self._xr_swapchain, xr.SwapchainImageAcquireInfo())
289     xr.wait_swapchain_image(self._xr_swapchain, xr.SwapchainImageWaitInfo(timeout=xr.INFINITE_DURATION))
290
291     # Once we acquired it, we bind the image to our framebuffer object
292     glfw.make_context_current(self._window)
293     GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, self._xr_swapchain_fbo)
294     GL.glFramebufferTexture2D(
295         GL.GL_FRAMEBUFFER,
296         GL.GL_COLOR_ATTACHMENT0,
297         GL.GL_TEXTURE_2D if self._samples == None else GL.GL_TEXTURE_2D_MULTISAMPLE,
298         self._xr_swapchain_images[image_index].image,
299         0
300     )
301
302     # We ask MuJoCo to render on its own offscreen framebuffer
303     mujoco.mjr_setBuffer(mujoco.mjtFramebuffer.mjFB_OFFSCREEN, self._mj_context)

```

```

304 mujoco.mj_render(mujoco.MjrRect(0, 0, self._width_render, self._height), self._mj_scene, self._mj_context)
305
306 # We copy what MuJoCo rendered on our framebuffer object
307 GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, self._mj_context.offFBO)
308 GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, self._xr_swapchain_fbo)
309 GL.glBlitFramebuffer(
310     0, 0,
311     self._width_render, self._height,
312     0, 0,
313     self._width_render, self._height,
314     GL.GL_COLOR_BUFFER_BIT,
315     GL.GL_NEAREST
316 )
317
318 if self._mirror_window:
319     # We copy the data from the MuJoCo buffer to the window one (0 is the default window fbo)
320     if self._samples is not None:
321         # We first resolve multi-sample if needed
322         GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, self._mj_context.offFBO_r)
323         GL.glBlitFramebuffer(
324             0, 0,
325             self._width_render, self._height,
326             0, 0,
327             self._width_render, self._height,
328             GL.GL_COLOR_BUFFER_BIT,
329             GL.GL_NEAREST
330         )
331         GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, self._mj_context.offFBO_r)
332
333     # We then copy the data to the window
334     GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, 0)
335     GL.glBlitFramebuffer(
336         0, 0,
337         self._width, self._height, # one eye only (left)
338         0, 0,
339         *self._window_size,
340         GL.GL_COLOR_BUFFER_BIT,
341         0x90BA # EXT_framebuffer_multisample_blit_scaled, SCALED_RESOLVE_FASTEST_EXT
342     )
343     xr.release_swapchain_image(self._xr_swapchain, xr.SwapchainImageReleaseInfo())
344
345 def __exit__(self, exc_type, exc_value, traceback):
346     if self._window is not None:
347         glfw.make_context_current(self._window)
348         if self._xr_swapchain_fbo is not None:
349             GL.glDeleteFramebuffers(1, [self._xr_swapchain_fbo])
350             self._xr_swapchain_fbo = None
351         glfw.terminate()
352     if self._xr_swapchain is not None:
353         xr.destroy_swapchain(self._xr_swapchain)
354     if self._xr_session is not None:
355         xr.destroy_session(self._xr_session)
356     if self._xr_instance is not None:
357         # # may break on Linux SteamVR
358         # xr.destroy_instance(self._xr_instance)
359         pass # does not seem to work
360     glfw.terminate()
361
362 def frame(self):
363     glfw.poll_events()
364     self._poll_xr_events()
365     if glfw.window_should_close(self._window):
366         self._should_quit = True
367
368     if self._should_quit:
369         return
370
371     if self._start_xr_frame():
372         self._update_mujoco()
373         self._update_views()
374         if self._xr_frame_state.should_render:
375             self._render()
376         self._end_xr_frame()
377
378 def loop(self):
379     while not self._should_quit:
380         self.frame()

```