MuJoCoXR

Linking MuJoCo and a VR environment

2024-08-18

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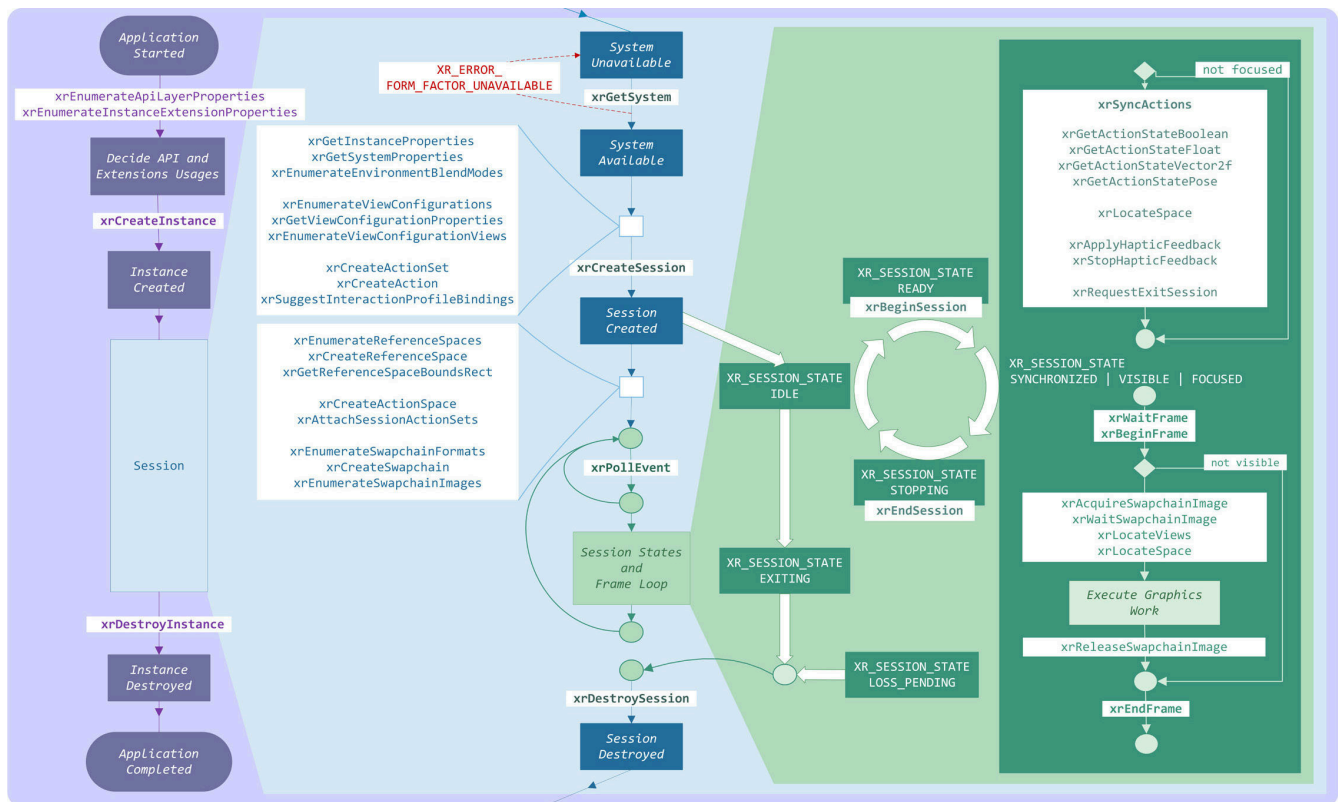
1. Available extensions are fetched to see if the ones needed are present (for instance, the extension that tells OpenXR to use the OpenGL graphics API, the debug utils extension…)
2. An "instance" is created with the application informations and extensions list. Future methods will use this instance.

---

[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[13]
*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.

---

[13]Extract of the OpenXR Reference Guide

Figure 2 — Lifecycle of an OpenXR session[13]
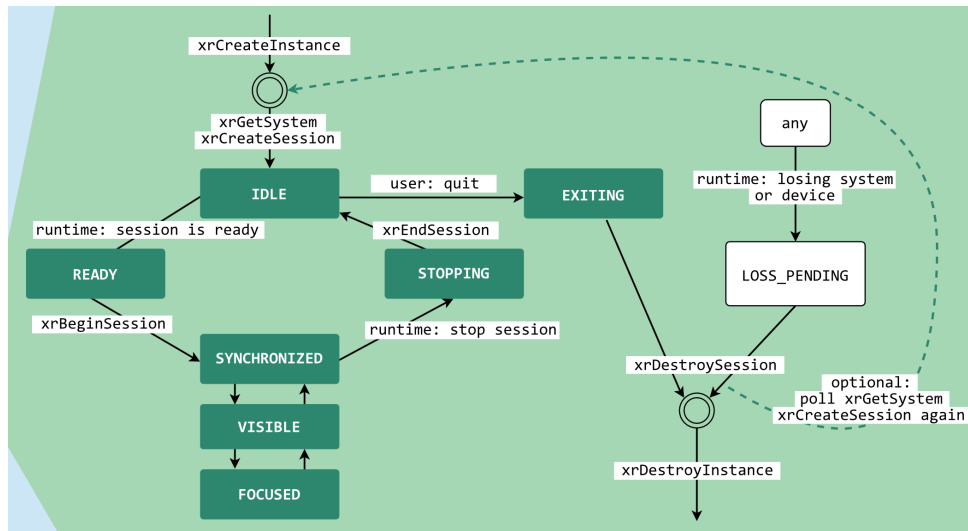
II - 3. MuJoCo

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:

```python
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.[14] `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.

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

---

[14]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` .

```python
104  if not glfw.init():
105      raise RuntimeError("GLFW initialization failed")
106  glfw.window_hint(glfw.DOUBLEBUFFER, False)
107  glfw.window_hint(glfw.RESIZABLE, False)
108  glfw.window_hint(glfw.SAMPLES, 0) # no need for multisampling 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, None,
     None)
113  if self._window is None:
114      raise RuntimeError("Failed to create GLFW window")
115  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 - 4.).

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.

```python
146  self._xr_swapchain = xr.create_swapchain(self._xr_session,
     xr.SwapchainCreateInfo(
```

```
          usage_flags=xr.SwapchainUsageFlags.TRANSFER_DST_BIT |
147   xr.SwapchainUsageFlags.COLOR_ATTACHMENT_BIT |
      xr.SwapchainUsageFlags.SAMPLED_BIT,
148       format=GL.GL_RGBA8,
149       sample_count=1 if self._samples is None else self._samples,
150       array_size=1,
151       face_count=1,
152       mip_count=1,
153       width=self._width_render,
154       height=self._height
155   ))
      self._xr_swapchain_images =
156   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[15] 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.

```
158   self._xr_projection_layer = xr.CompositionLayerProjection(
159       # Default space params are okay: identity quaternion and zero vector.
      Let's use them.
160       space=xr.create_reference_space(self._xr_session,
      xr.ReferenceSpaceCreateInfo()),
161       views = [xr.CompositionLayerProjectionView(
162           sub_image=xr.SwapchainSubImage(
163               swapchain=self._xr_swapchain,
164               image_rect=xr.Rect2Di(
165                   extent=xr.Extent2Di(self._width, self._height),
166                   offset=None if eye_index == 0 else xr.Offset2Di(x =
      self._width) # right eye offset
167               )
168           )
169       ) for eye_index in range(2)]
170   )
171
172   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:

---

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

1. The reference space for the projection is created with the default options (the `STAGE` space type[16] and default orientation and position).
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` *.*

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

The main loop structure looks like this:

```
loop:
```

---

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

```
    poll_events
    if should_quit:
        stop

    if try_start_frame:
        make_a_frame
```

The `poll_events` part is made of this:

```
362  glfw.poll_events()
363  self._poll_xr_events()
364  if glfw.window_should_close(self._window):
365      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 wait for the next XR frame:

```
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      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`. We do *not* start the frame now: it's better to do it the moment right before we will actually render something.

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:

```python
371  self._update_mujoco()
372  self._update_views()
373
374  xr.begin_frame(self._xr_session, None)
375  if self._xr_frame_state.should_render:
376      self._render()
377  self._end_xr_frame()
```

The `_update_mujoco` method is really simple:

```python
200  mujoco.mj_step(self._mj_model, self._mj_data)
     mujoco.mjv_updateScene(self._mj_model, self._mj_data, self._mj_option,
201  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:

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

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.

```
302   mujoco.mjr_setBuffer(mujoco.mjtFramebuffer.mjFB_OFFSCREEN,
      self._mj_context)
303   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.

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

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. Performance

For now, everything is single-threaded. If the simulation time is long, it can reduce drastically the framerate and lead to uncomfortable VR experience. It is however possible to split our main loop in two different threads: a "simulation" thread and a "render" thread (see Figure 3).

Figure 3 — How to multithread an OpenXR application[17]
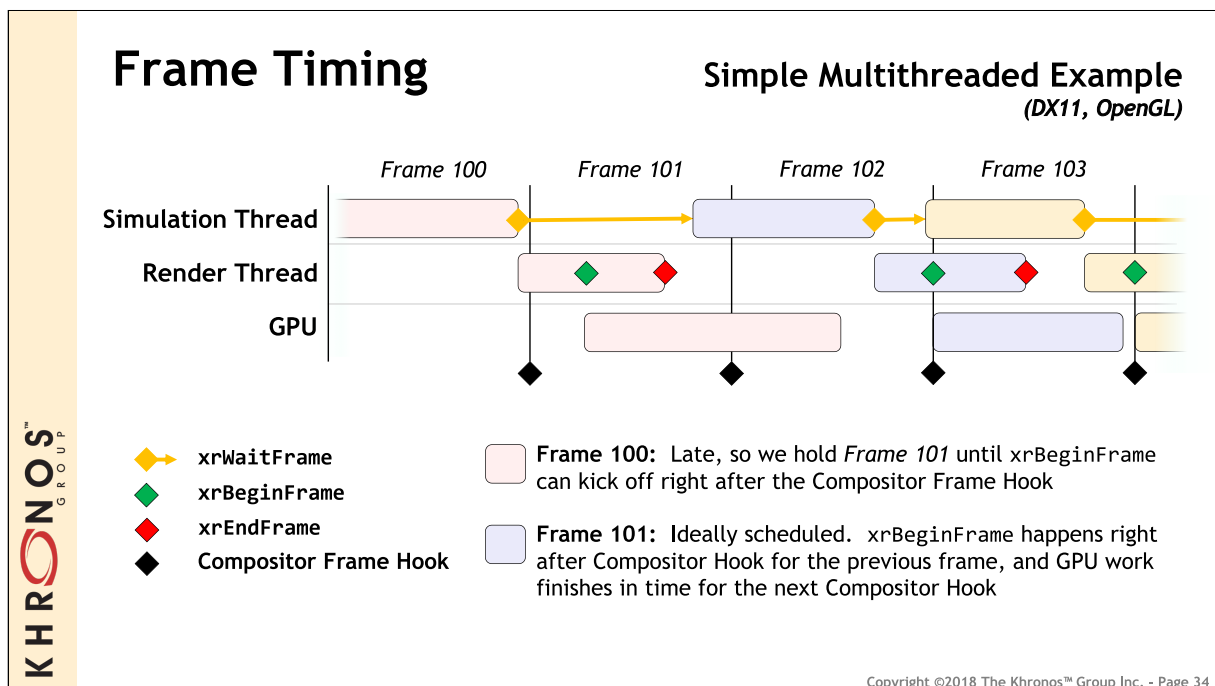
As you can see, the two threads will not be entirely parallel but some of their work will definitely be.

To implement this, we first need to define which work will be given to which thread:
- **Simulation thread:** event polling, frame waiting and simulation stepping.
- **Render thread:** view update, frame begin/end, rendering.

Beware: until now, we have grouped the simulation stepping and rendering preparation in one single step (`_update_mujoco`). This should now be split: `mj_step` will be kept in the simulation thread, while `mjv_updateScene` will be put in the rendering thread. This is because `mj_step` can be called while the simulation is being rendered, whereas `mjv_updateScene` cannot (because it contains the geometry data to be rendered).

Now we know where to put which work, the only thing left to do is to split the original `frame` function in two different threads, and use a synchronization structure to signal to the render thread to do its work when simulation is over (a `threading.Semaphore` is suited for this).

A race condition might happen when you remove the headset in this multithreaded setup: `xr.end_session` might be called by the simulation thread while a frame is still being rendered by the other thread. To avoid this, use another synchronization structure (a `threading.Condition` is fine).

IV - 3. 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:

```python
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)
```

---

[17]Extract of the OpenXR presentation at GDC 2018

```
15       orientation = [space_location.pose.orientation[3],
      *space_location.pose.orientation[0:3]]
16       mujoco.mjv_room2model(hand_pos, hand_rot,
      list(space_location.pose.position), orientation, self._mj_scene)
17       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 - 4. 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
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          glfw.make_context_current(None) # To let other threads use the context if needed
27          return self
28
29      def _init_xr(self):
30          """
31          Initializes the OpenXR environment prior to session creation.
32
33          Also fetches informations about the setup, most importantly the render size.
34          """
35          extensions = [xr.KHR_OPENGL_ENABLE_EXTENSION_NAME]
36          instance_create_info = xr.InstanceCreateInfo(
37              application_info=xr.ApplicationInfo(
38                  application_name=APP_NAME,
39                  engine_name="pyopenxr",
40                  engine_version=xr.PYOPENXR_CURRENT_API_VERSION,
41                  api_version=xr.Version(1, 0, xr.XR_VERSION_PATCH)
42              )
43          )
44
45          if self._debug:
46              def debug_callback_py(severity, _type, data, _user_data):
47                  print(severity, f"{data.contents.function_name.decode()}: {data.contents.message.decode()}")
48                  return True
49
50              debug_messenger = xr.DebugUtilsMessengerCreateInfoEXT(
51                  message_severities=
52                      xr.DebugUtilsMessageSeverityFlagsEXT.VERBOSE_BIT
53                      | xr.DebugUtilsMessageSeverityFlagsEXT.INFO_BIT
54                      | xr.DebugUtilsMessageSeverityFlagsEXT.WARNING_BIT
55                      | xr.DebugUtilsMessageSeverityFlagsEXT.ERROR_BIT,
56                  message_types=
57                      xr.DebugUtilsMessageTypeFlagsEXT.GENERAL_BIT
58                      | xr.DebugUtilsMessageTypeFlagsEXT.VALIDATION_BIT
59                      | xr.DebugUtilsMessageTypeFlagsEXT.PERFORMANCE_BIT
60                      | xr.DebugUtilsMessageTypeFlagsEXT.CONFORMANCE_BIT,
61                  user_callback=xr.PFN_xrDebugUtilsMessengerCallbackEXT(debug_callback_py)
62              )
63              instance_create_info.next = ctypes.cast(ctypes.pointer(debug_messenger), ctypes.c_void_p)
64              extensions.append(xr.EXT_DEBUG_UTILS_EXTENSION_NAME)
65
66          instance_create_info.enabled_extension_names = extensions
67          self._xr_instance = xr.create_instance(instance_create_info)
68
69          # The following fetches important informations about the setup
70          # (mainly rendering size)
71          self._xr_system = xr.get_system(self._xr_instance, xr.SystemGetInfo(xr.FormFactor.HEAD_MOUNTED_DISPLAY))
72          assert xr.enumerate_view_configurations(self._xr_instance, self._xr_system)[0] ==
    xr.ViewConfigurationType.PRIMARY_STEREO
73
```

```
74          views_config = xr.enumerate_view_configuration_views(self._xr_instance, self._xr_system,
        xr.ViewConfigurationType.PRIMARY_STEREO)
75          assert len(views_config) == 2
76          assert views_config[0].recommended_image_rect_width == views_config[1].recommended_image_rect_width
77          assert views_config[0].recommended_image_rect_height == views_config[1].recommended_image_rect_height
78
79          self._width, self._height = views_config[0].recommended_image_rect_width,
        views_config[0].recommended_image_rect_height
80          self._width_render = self._width * 2
81
82          pxrGetOpenGLGraphicsRequirementsKHR = ctypes.cast(
83              xr.get_instance_proc_addr(
84                  self._xr_instance,
85                  "xrGetOpenGLGraphicsRequirementsKHR",
86              ),
87              xr.PFN_xrGetOpenGLGraphicsRequirementsKHR
88          )
89          graphics_result = pxrGetOpenGLGraphicsRequirementsKHR(
90              self._xr_instance,
91              self._xr_system,
92              ctypes.byref(xr.GraphicsRequirementsOpenGLKHR())
93          )
94          graphics_result = xr.exception.check_result(xr.Result(graphics_result))
95          if graphics_result.is_exception():
96              raise graphics_result
97
98      def _init_window(self):
99          """
100         Initializes the GLFW window (and make it hidden if mirrored mode is disabled).
101
102         Creates the OpenGL context that will be used.
103         """
104         if not glfw.init():
105             raise RuntimeError("GLFW initialization failed")
106         glfw.window_hint(glfw.DOUBLEBUFFER, False)
107         glfw.window_hint(glfw.RESIZABLE, False)
108         glfw.window_hint(glfw.SAMPLES, 0) # no need for multisampling 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, None, None)
113         if self._window is None:
114             raise RuntimeError("Failed to create GLFW window")
115         glfw.make_context_current(self._window)
116         # Attempt to disable vsync on the desktop window or
117         # it will interfere with the OpenXR frame loop timing
118         glfw.swap_interval(0)
119
120     def _prepare_xr(self):
121         """
122         Creates the OpenXR session and prepares everything to launch the frames loop.
123         """
124         if platform.system() == 'Windows':
125             from OpenGL import WGL
126             graphics_binding = xr.GraphicsBindingOpenGLWin32KHR()
127             graphics_binding.h_dc = WGL.wglGetCurrentDC()
128             graphics_binding.h_glrc = WGL.wglGetCurrentContext()
129         else:
130             from OpenGL import GLX
131             graphics_binding = xr.GraphicsBindingOpenGLXlibKHR()
132             graphics_binding.x_display = GLX.glXGetCurrentDisplay()
133             graphics_binding.glx_context = GLX.glXGetCurrentContext()
134             graphics_binding.glx_drawable = GLX.glXGetCurrentDrawable()
135
136         self._xr_session = xr.create_session(
137             self._xr_instance,
138             xr.SessionCreateInfo(
139                 xr.SessionCreateFlags.NONE,
140                 self._xr_system,
141                 next=ctypes.cast(ctypes.pointer(graphics_binding), ctypes.c_void_p)
142             )
143         )
144         self._xr_session_state = xr.SessionState.IDLE
145
146         self._xr_swapchain = xr.create_swapchain(self._xr_session, xr.SwapchainCreateInfo(
147             usage_flags=xr.SwapchainUsageFlags.TRANSFER_DST_BIT | xr.SwapchainUsageFlags.COLOR_ATTACHMENT_BIT |
        xr.SwapchainUsageFlags.SAMPLED_BIT,
```

```python
148            format=GL.GL_RGBA8,
149            sample_count=1 if self._samples is None else self._samples,
150            array_size=1,
151            face_count=1,
152            mip_count=1,
153            width=self._width_render,
154            height=self._height
155        ))
156        self._xr_swapchain_images = xr.enumerate_swapchain_images(self._xr_swapchain, xr.SwapchainImageOpenGLKHR)
157
158        self._xr_projection_layer = xr.CompositionLayerProjection(
159            # Default space params are okay: identity quaternion and zero vector. Let's use them.
160            space=xr.create_reference_space(self._xr_session, xr.ReferenceSpaceCreateInfo()),
161            views = [xr.CompositionLayerProjectionView(
162                sub_image=xr.SwapchainSubImage(
163                    swapchain=self._xr_swapchain,
164                    image_rect=xr.Rect2Di(
165                        extent=xr.Extent2Di(self._width, self._height),
166                        offset=None if eye_index == 0 else xr.Offset2Di(x = self._width) # right eye offset
167                    )
168                )
169            ) for eye_index in range(2)]
170        )
171
172        self._xr_swapchain_fbo = GL.glGenFramebuffers(1)
173
174    def _prepare_mujoco(self):
175        """
176        Prepares the MuJoCo environment.
177        """
178        self._mj_model = mujoco.MjModel.from_xml_path("assets/balloons.xml")
179        self._mj_data = mujoco.MjData(self._mj_model)
180        self._mj_scene = mujoco.MjvScene(self._mj_model, 1000)
181        self._mj_scene.stereo = mujoco.mjtStereo.mjSTEREO_SIDEBYSIDE
182
183        # We want the visualization properties set BEFORE creation of the context,
184        # otherwise we would have to call mjr_resizeOffscreen.
185        self._mj_model.vis.global_.offwidth = self._width_render
186        self._mj_model.vis.global_.offheight = self._height
187        self._mj_model.vis.quality.offsamples = 0 if self._samples is None else self._samples
188
189        self._mj_context = mujoco.MjrContext(self._mj_model, mujoco.mjtFontScale.mjFONTSCALE_100)
190        self._mj_camera = mujoco._structs.MjvCamera()
191        self._mj_option = mujoco.MjvOption()
192        # We do NOT want to call mjv_defaultFreeCamera
193
194        mujoco.mjv_defaultOption(self._mj_option)
195
196    def _update_mujoco(self):
197        """
198        Updates MuJoCo for one frame.
199        """
200        mujoco.mj_step(self._mj_model, self._mj_data)
201        mujoco.mjv_updateScene(self._mj_model, self._mj_data, self._mj_option, None, self._mj_camera,
    mujoco.mjtCatBit.mjCAT_ALL, self._mj_scene)
202
203    def _wait_xr_frame(self):
204        """
205        Wait to begin the next OpenXR frame.
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            return True
218        return False
219
220    def _end_xr_frame(self):
221        xr.end_frame(self._xr_session, xr.FrameEndInfo(
222            self._xr_frame_state.predicted_display_time,
223            xr.EnvironmentBlendMode.OPAQUE,
224            layers=[ctypes.byref(self._xr_projection_layer)] if self._xr_frame_state.should_render else []
225        ))
```

```python
226
227     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         GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, self._xr_swapchain_fbo)
293         GL.glFramebufferTexture2D(
294             GL.GL_FRAMEBUFFER,
295             GL.GL_COLOR_ATTACHMENT0,
296             GL.GL_TEXTURE_2D if self._samples == None else GL.GL_TEXTURE_2D_MULTISAMPLE,
297             self._xr_swapchain_images[image_index].image,
298             0
299         )
300
301         # We ask MuJoCo to render on its own offscreen framebuffer
```

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

```
380            glfw.make_context_current(self._window)
381            while not self._should_quit:
382                self.frame()
383
384    if __name__ == "__main__":
385        with MujocoXRViewer(debug=True, mirror_window=True, samples=8) as mjxr:
386            mjxr.loop()
```