XRFeitoria

Release 0.6.2

XRFeitoria Authors

Apr 11, 2024

BEGINNER'S GUIDE

XRFeitoria is a rendering toolbox for generating synthetic data photorealistic with ground-truth annotations. It is a part of the OpenXRLab project.

CHAPTER

ONE

MAJOR FEATURES

- Support rendering photorealistic images with ground-truth annotations.
- Support multiple engine backends, including Unreal Engine and Blender.
- Support assets/camera management, including import, export, and delete.
- Support a CLI tool to render images from a mesh file.

CHAPTER

TWO

INSTALLATION

```
pip install xrfeitoria

# to use visualization tools
pip install xrfeitoria[vis]
```

2.1 Requirements

- Python >= 3.8
- **(optional) Unreal Engine >= 5.1**
  Windows

  Linux
- **(optional) Blender >= 3.0**
  Windows

  Linux

  MacOS

2.1.1 CLI

A CLI tool to render images from a mesh file.

```
xf-render --help

# render a mesh file
xf-render {mesh_file}

# for example
wget https://graphics.stanford.edu/~mdfisher/Data/Meshes/bunny.obj
xf-render bunny.obj
```

2.1.2 Tutorials

A quick start guide to using the `XRFeitoria` package. The following tutorials cover the basic operations needed to render with `XRFeitoria`.

Tutorial01 - Getting Started

Overview

This tutorial demonstrates a minimal application of XRFeitoria. By the end of this tutorial, you will be able to:

- Open Blender/Unreal Engine by XRFeitoria
- Import a mesh
- Add a camera
- Render images and get annotations of the mesh

1. Import XRFeitoria

After installing `pip install xrfeitoria[vis]`, you can import it as follows:

[ ]:
```
import xrfeitoria as xf
```

2. Choose engine

`XRFeitoria` supports both `Blender` and `Unreal Engine`. Choose your engine and replace the following `engine_exec_path` with your own engine path.

Then, initialize XRFeitoria and open the engine by `xf.init_blender` or by `xf.init_unreal`.

[ ]:
```
# Replace with your executable path
engine_exec_path = 'C:/Program Files/Blender Foundation/Blender 3.3/blender.exe'
# engine_exec_path = 'C:/Program Files/Epic Games/UE_5.2/Engine/Binaries/Win64/
↪UnrealEditor-Cmd.exe'
```

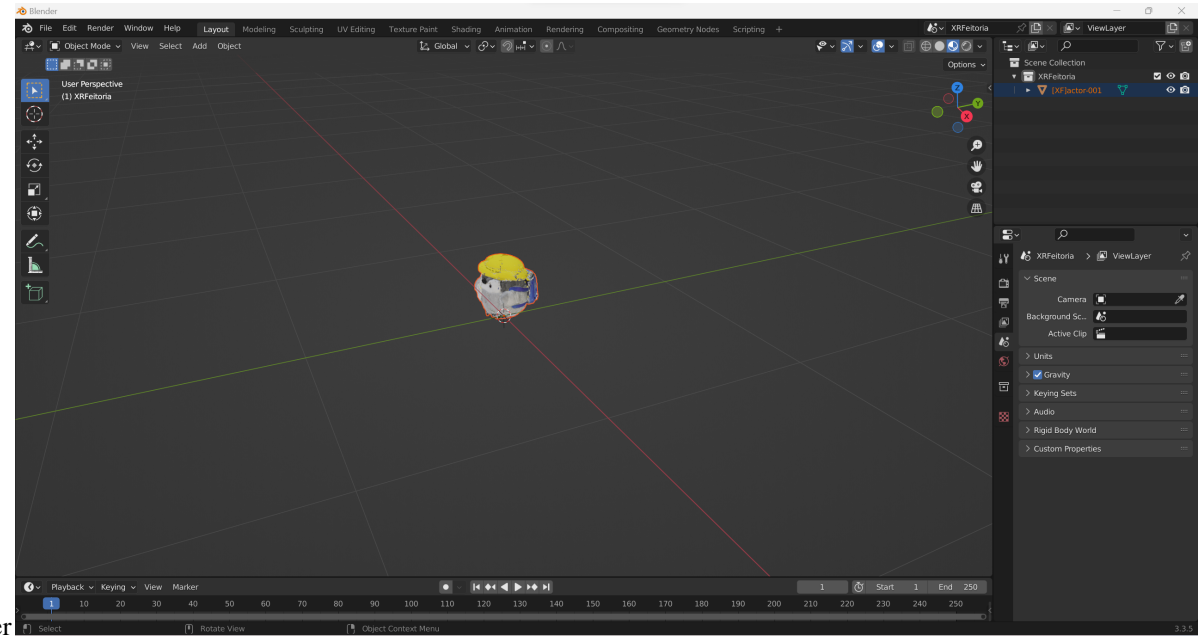
[ ]:
```
from pathlib import Path

exec_path_stem = Path(engine_exec_path).stem.lower()
if 'blender' in exec_path_stem:
    # Open Blender
    render_engine = 'blender'
    xf_runner = xf.init_blender(exec_path=engine_exec_path, background=False, new_
↪process=True)
elif 'unreal' in exec_path_stem:
    # Unreal Engine requires a project to be opened
    # Here we use a sample project, which is downloaded from the following link
    # You can also use your own project
    import shutil

    from xrfeitoria.utils.downloader import download
```

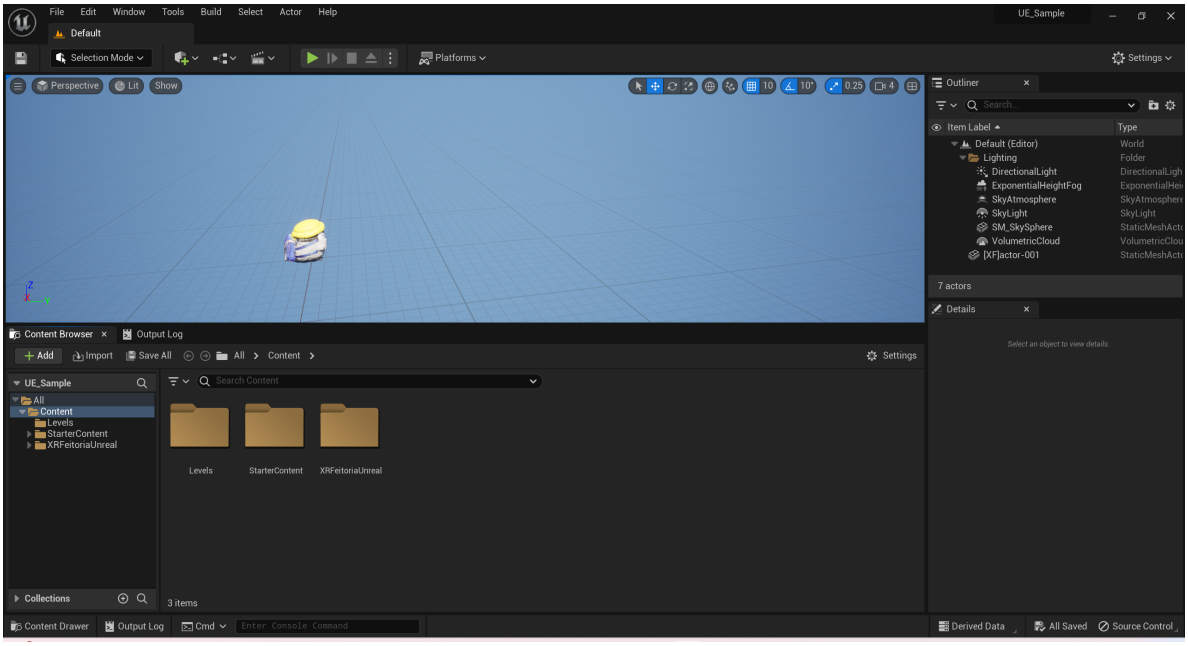
(continues on next page)

(continued from previous page)

```
    unreal_project_zip = download(url='https://openxrlab-share.oss-cn-hongkong.aliyuncs.
↪com/xrfeitoria/tutorials/unreal_project/UE_Sample.zip',
                                  dst_dir="./tutorial01/assets/")
    shutil.unpack_archive(filename=unreal_project_zip, extract_dir='./tutorial01/assets/
↪')

    # Open Unreal Engine
    render_engine = 'unreal'
    xf_runner = xf.init_unreal(exec_path=engine_exec_path,
                               background=False,
                               new_process=True,
                               project_path='./tutorial01/assets/UE_sample/UE_sample.
↪uproject')
```

Now you can see a new Blender/Unreal Engine process has been started.

- Blender 

- Unreal Engine 

3. Import a mesh

Download the scanned Koupen Chan model by the following cell.

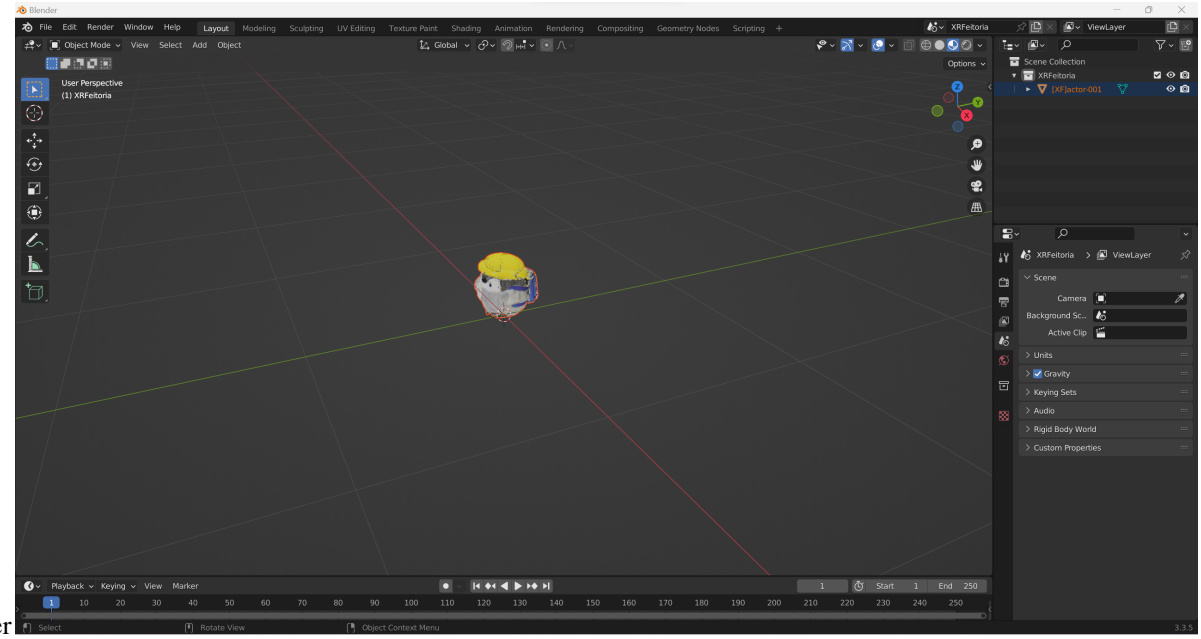
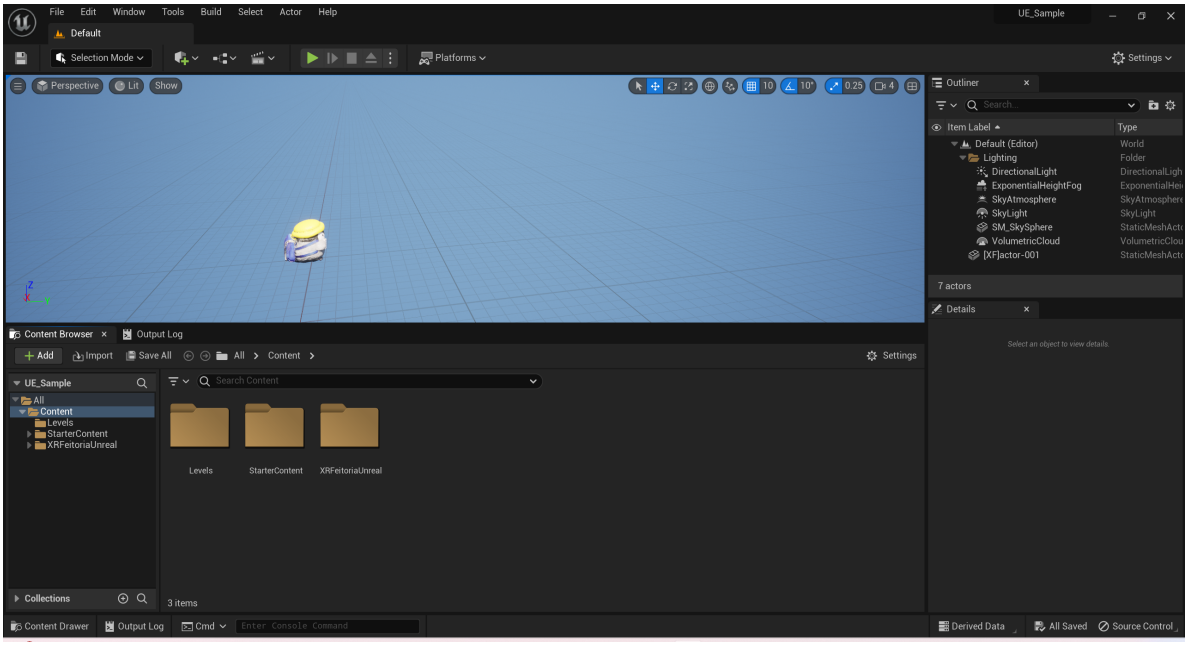
```
[ ]: from xrfeitoria.utils.downloader import download

     # Download the Koupen-chan model
     kc_path = download(url='https://openxrlab-share.oss-cn-hongkong.aliyuncs.com/xrfeitoria/
     ↪tutorials/assets/koupen_chan.fbx',
                        dst_dir="./tutorial01/assets/")
```

Import the .fbx file to create an `Actor` instance.

`Actor` is the container of a mesh. By using `Actor`, you can place a mesh in the space and set its transform data (location, rotation and scale).

```
[ ]: # Import the Koupen-chan
     actor_kc = xf_runner.Actor.import_from_file(file_path=kc_path, stencil_value=255)
```

Switch to the engine window, and you can see the Koupen chan has been *imported*. The space to place `Actor` is called `Level`, and you can *add*, *remove*, or *modify* `Actor` in the `Level`.

- Blender 
- Unreal Engine 

If you use `Unreal Engine`, the `Level` should be saved after been modified.

```
[ ]: # save the level
if render_engine == 'unreal':
    xf_runner.utils.save_current_level()
```

4. Add a sequence

`Sequence` is a multifunctional class in XRFeitoria. It can be used for:

- rendering
- adding transform keys
- grouping different objects

Here, we use it for rendering. Firstly we will add a `Camera` in the `Sequence` by using the function `spawn_camera` and set its location, rotation, and focal length. Then, we will add a render job to the renderer and set the render settings by using the function `add_to_renderer`.

[ ]:
```
from xrfeitoria.data_structure.models import RenderPass

# Use `with` statement to create a sequence, and it will be automatically close the␣
↪sequence after the code block is executed.
sequence_name = 'MySequence'
with xf_runner.Sequence.new(seq_name=sequence_name, replace=True) as seq:

    # Add a camera and make it look at the koupen-chan
    camera_location = (0.0, -0.8, 0.0)
    camera_rotation = xf_runner.utils.get_rotation_to_look_at(location=camera_location,␣
↪target=actor_kc.location)
    camera = seq.spawn_camera(location=camera_location, rotation=camera_rotation, fov=90)

    # Add a render job to renderer
    # In render job, you can specify the output path, resolution, render passes, etc.
    # The ``output_path`` is the path to save the rendered data.
    # The ``resolution`` is the resolution of the rendered image.
    # The ``render_passes`` define what kind of data you want to render, such as img,␣
↪mask, normal, etc.
    # and what kind of format you want to save, such as png, exr, etc.
    seq.add_to_renderer(
        output_path=f'./tutorial01/outputs/{render_engine}/',
        resolution=(1280, 720),
        render_passes=[RenderPass('img', 'png'),
                       RenderPass('mask', 'exr'),
                       RenderPass('normal', 'exr'),
                       RenderPass('diffuse', 'exr')]
    )
```

5. Render

Use the following cell to render and save the images to the `output_path` you set in `seq.add_to_renderer` above.

[ ]:
```
xf_runner.render()
```

Check the `output_path`, and you can see the rendered images and various annotations. The following code shows the rendered image and the corresponding annotations.

[ ]:
```python
import matplotlib.pyplot as plt

from xrfeitoria.utils.viewer import Viewer

xf_viewer = Viewer(sequence_dir=f'./tutorial01/outputs/{render_engine}/{sequence_name}/')
img = xf_viewer.get_img(camera_name=camera.name, frame=0)
diffuse = xf_viewer.get_diffuse(camera_name=camera.name, frame=0)
mask = xf_viewer.get_mask(camera_name=camera.name, frame=0)
normal = xf_viewer.get_normal(camera_name=camera.name, frame=0)

plt.figure(figsize=(20, 20))

plt.subplot(1, 4, 1)
plt.imshow(img)
plt.axis('off')
plt.title('img')

plt.subplot(1, 4, 2)
plt.imshow(diffuse)
plt.axis('off')
plt.title('diffuse')

plt.subplot(1, 4, 3)
plt.imshow(mask)
plt.axis('off')
plt.title('mask')

plt.subplot(1, 4, 4)
plt.imshow(normal)
plt.axis('off')
plt.title('normal')
```

Hint: When using Unreal Engine, if the image of the mask looks weird, try running the notebook again.

6. Final step

This is a good start! Finally, **Do remember** to close the engine.

[ ]:
```python
xf_runner.close()
```

Ref to api docs, you can always use `with` statement to ensure the engine is closed when the codes are finished.

7. Conclusion

In this tutorial, we imported a mesh and rendered images for it. To accomplish this procedure, essential steps should be taken:

- Initialization
- Import an `Actor`
- Add a `Sequence`
- Add a `Camera`

- Render

It is worth mention that while `Sequence` does not directly perform rendering, the creation of `Sequence` is necessary for adding cameras and submitting rendering jobs. And the detailed definitions of the classes Actor, Camera, and Sequence can be referred to the documentation.

Tutorial02 - Randomization

Overview

By the last tutorial, you have learned the concept of `Actor`, `Level` and `Sequence`. In this tutorial, you will learn how to add random transform keys to an `Actor` in order to render multiple poses of it. By the end of this tutorial, you will be able to:

- Initialize XRFeitoria
- Import `Actor` and label it by `stencil value`
- Set the scale of `Actor` in the `Level`
- Create a `Sequence` for rendering and adding transform keys to `Actor`
- Add a camera in the `Sequence`
- Render images and annotations

1. Initialization

Install the following packages that will be used in this tutorial:

[ ]:
```
%pip install objaverse
%pip install scipy
```

Then, similar to *Tutorial01*, specify your engine path and initialize XRFeitoria.

[ ]:
```
import xrfeitoria as xf
```

[ ]:
```
# Replace with your executable path
engine_exec_path = 'C:/Program Files/Blender Foundation/Blender 3.3/blender.exe'
# engine_exec_path = 'C:/Program Files/Epic Games/UE_5.2/Engine/Binaries/Win64/
↪UnrealEditor-Cmd.exe'
```

[ ]:
```
from pathlib import Path

exec_path_stem = Path(engine_exec_path).stem.lower()
if 'blender' in exec_path_stem:
    # Open Blender
    render_engine = 'blender'
    xf_runner = xf.init_blender(exec_path=engine_exec_path, background=False, new_
↪process=True)
elif 'unreal' in exec_path_stem:
    # Unreal Engine requires a project to be opened
    # Here we use a sample project, which is downloaded from the following link
    # You can also use your own project
```

(continues on next page)

(continued from previous page)

```
    import shutil

    from xrfeitoria.utils.downloader import download
    unreal_project_zip = download(url='https://openxrlab-share.oss-cn-hongkong.aliyuncs.
↪com/xrfeitoria/tutorials/unreal_project/UE_Sample.zip',
                                  dst_dir="./tutorial02/assets/")
    shutil.unpack_archive(filename=unreal_project_zip, extract_dir='./tutorial02/assets/
↪')

    # Open Unreal Engine
    render_engine = 'unreal'
    xf_runner = xf.init_unreal(exec_path=engine_exec_path,
                               background=False,
                               new_process=True,
                               project_path='./tutorial02/assets/UE_sample/UE_sample.
↪uproject')
```

Now you can see a new Blender/Unreal Engine process has started.

2. Import meshes

Download some meshes from Objaverse.

```
[ ]: import objaverse

objects = objaverse.load_objects(
    uids=['eb0807309530496aaab9dcff67bf5c31',
          'b4065dd5ce9d46be90db3e1f3e4b9cc1',
          '0176be079c2449e7aaebfb652910a854',
          'f130ebeb60f24ed8bd3714a7ed3ba280',
          '289a2221178843a78ad433705555e16a',
          'b7f7ab9bf7244c3a8851bae3fb0bf741',
        ],
    download_processes=1
)
```

Import the meshes to create `Actor` instances in the `Level`.

Here we set different `stencil_value` for each `Actor`. The stencil value is used to distinguish different Actors when rendering segmentation masks.

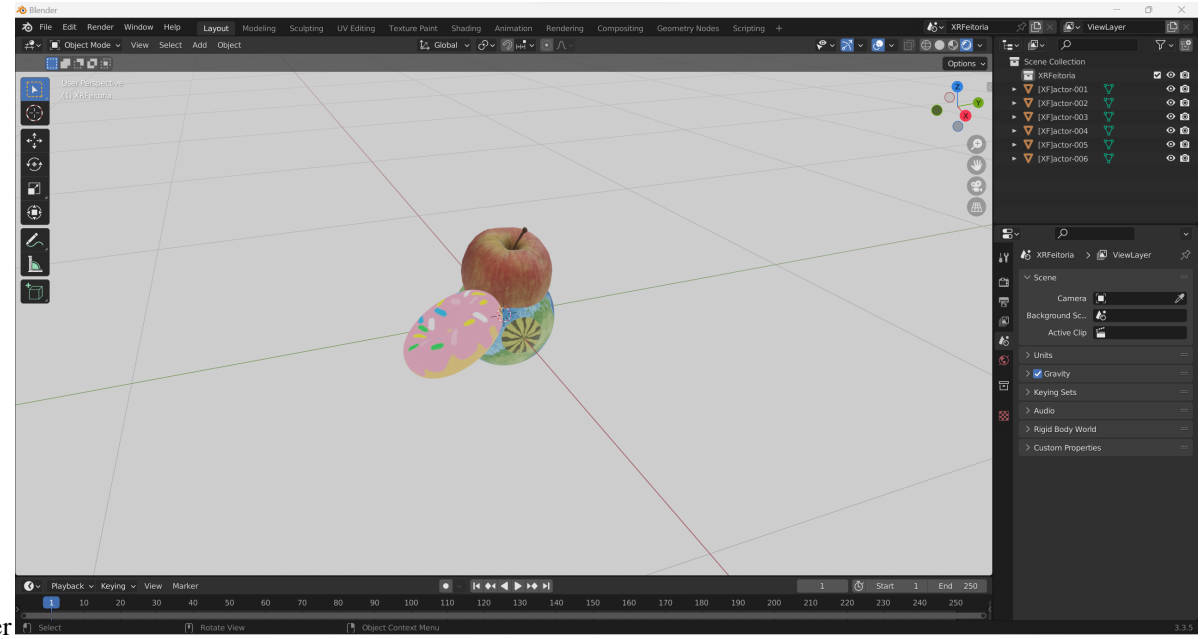
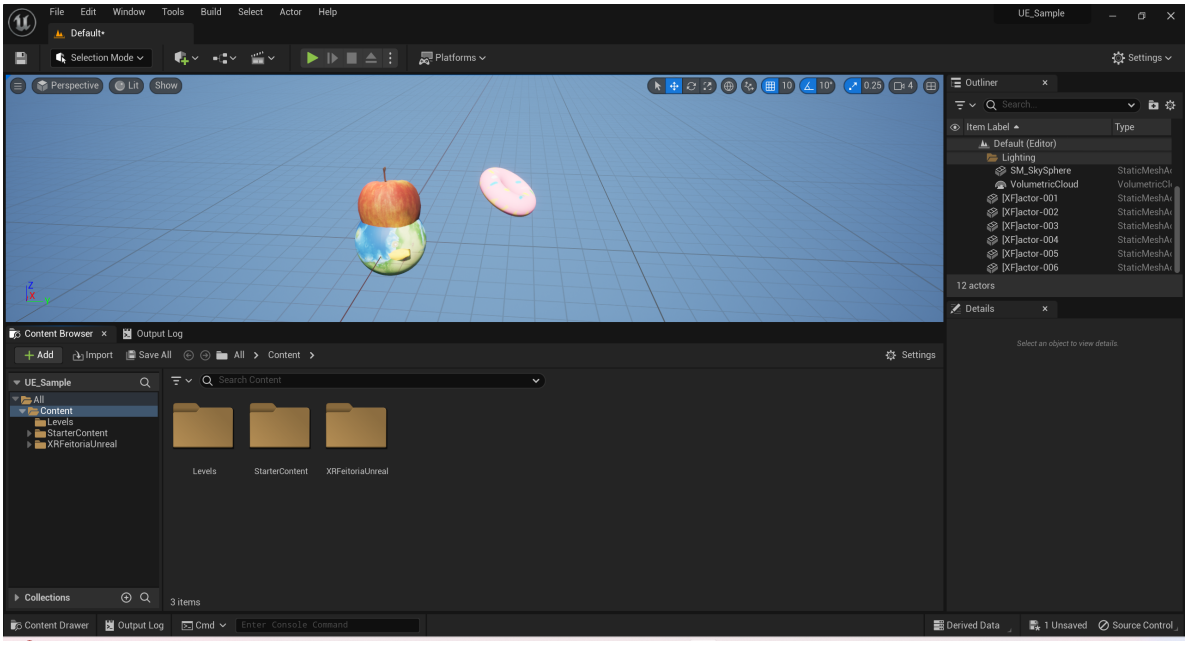
```
[ ]: actors = []
for idx, file_path in enumerate(objects.values()):
    actor = xf_runner.Actor.import_from_file(
        file_path=file_path,
        stencil_value=(idx+1)*10
    )
    actors.append(actor)
```

Switch to the engine window, and you can see the meshes has been imported.

Then, we adjust the scale of the Actors to make their sizes equal to 0.3m.

```
[ ]: actor_size = 0.3
for actor in actors:
    _scale = actor_size / max(actor.dimensions)
    actor.scale = (_scale, _scale, _scale)
```

Now they look like:

- Blender 
- Unreal Engine 

If you use `Unreal Engine`, the `Level` should be saved after been modified.

```
[ ]: # save the level
if render_engine == 'unreal':
    xf_runner.utils.save_current_level()
```

3. Add a sequence for rendering and adding transform keys

`Sequence` is a multifunctional class in XRFeitoria. It can be used for: - rendering - adding transform keys - grouping different objects.

Here, we use it for `rendering` and `adding transform keys`.

> `Transform keys` record the transformation (location, rotation and scale) of an `Actor` or a `Camera` at specific frames, and the transformation between two adjacent keys will be interpolated by the specified interpolation method. By adding transform keys, you can render multiple poses of an `Actor`.
>
> In `XRFeitoria`, transform keys are always stored in a list, and the members of the list are SequenceTransformKey object.

Firstly, we randomly generate some transform keys for each actors.

[ ]:
```python
import random

from loguru import logger
from scipy.stats import qmc

from xrfeitoria.data_structure.models import SequenceTransformKey as SeqTransKey

# Set the number of frames
frame_num = 10

# Use a dictionary to store the transform keys of all actors
all_actors_transform_keys = {actor: [] for actor in actors}

# Iterate over all frames
for i in range(frame_num):

    # Generate random locations by Poisson Disk Sampling
    # The minimum distance between two actors is set to be `actor_size` we defined before
    posson_engine = qmc.PoissonDisk(d=2, radius=actor_size)
    sample_location = posson_engine.random(len(actors))

    # Set the transform keys for each actor
    for actor_idx, actor in enumerate(actors):
        actor_scale = actor.scale

        # Get the location from the samples generated by Poisson Disk Sampling
        random_location = (sample_location[actor_idx][0],
                           0.0,
                           sample_location[actor_idx][1])

        # Generate random rotations
        random_rotation = (random.random() * 360.0,
                           random.random() * 360.0,
                           random.random() * 360.0)

        # Generate random scales
        scale = random.uniform(0.5, 1.0)
        random_scale = (scale * actor_scale[0],
                        scale * actor_scale[1],
```

(continues on next page)

(continued from previous page)

```
                        scale * actor_scale[2])

        # Save the transform keys
        all_actors_transform_keys[actor].append(
            SeqTransKey(
                frame=i,
                location=random_location,
                rotation=random_rotation,
                scale=random_scale,
                interpolation='AUTO',
            )
        )
    logger.info(f'Generated transform keys of frame {i}.')
```

Then we create a `Sequence` to apply the transform keys to the actors and render the images.

[ ]:
```
from xrfeitoria.data_structure.models import RenderPass

# Use the `with` statement to create a sequence, and it will be automatically close the␣
↪sequence after the code block is executed.
# The argument `seq_length` controls the number of frames to be rendered.
sequence_name = 'MySequence'
with xf_runner.Sequence.new(seq_name=sequence_name, seq_length=frame_num, replace=True)␣
↪as seq:
    ##############################
    ##### Add transform keys #####
    ##############################

    # The function `use_actor_with_keys` sets transform keys for the actor in the␣
↪sequence.
    # The transform keys are only stored in the sequence.
    # When the sequence is closed, the actor will be restored to its original␣
↪state(without transform keys).
    for actor, keys in all_actors_transform_keys.items():
        seq.use_actor_with_keys(actor=actor, transform_keys=keys)

    ######################
    ##### Rendering #####
    ######################

    # Add a camera and make it look at the specified location
    camera_location = (0.5, -3.0, 0.5)
    camera_rotation = xf_runner.utils.get_rotation_to_look_at(location=camera_location,␣
↪target=(0.5, 0.0, 0.5))
    camera = seq.spawn_camera(location=camera_location, rotation=camera_rotation, fov=45)

    # Add a render job to renderer
    # In render job, you can specify the output path, resolution, render passes, etc.
    # The ``output_path`` is the path to save the rendered data.
    # The ``resolution`` is the resolution of the rendered image.
    # The ``render_passes`` define what kind of data you want to render, such as img,␣
↪depth, normal, etc.
```

(continues on next page)

(continued from previous page)

```
    # and what kind of format you want to save, such as png, jpg, exr, etc.
    seq.add_to_renderer(
        output_path=f'./tutorial02/outputs/{render_engine}/',
        resolution=(1280, 720),
        render_passes=[RenderPass('img', 'png'),
                       RenderPass('mask', 'exr'),
                       RenderPass('normal', 'exr'),
                       RenderPass('diffuse', 'exr')],
    )
```

4. Render

The following code renders all the render jobs and save the images to the `output_path` set in `seq.add_to_renderer` above.

[ ]:
```
# Render
xf_runner.render()
```

Check the `output_path`, and you can see the rendered images and their annotations. Visualize the images and annotations by the following code.

[ ]:
```
import matplotlib.pyplot as plt

from xrfeitoria.utils.viewer import Viewer

xf_viewer = Viewer(sequence_dir=f'./tutorial02/outputs/{render_engine}/{sequence_name}/')

for i in range(frame_num):
    img = xf_viewer.get_img(camera_name=camera.name, frame=i)
    mask = xf_viewer.get_mask(camera_name=camera.name, frame=i)
    normal = xf_viewer.get_normal(camera_name=camera.name, frame=i)
    diffuse = xf_viewer.get_diffuse(camera_name=camera.name, frame=i)

    plt.figure(figsize=(20, 20))

    plt.subplot(1, 4, 1)
    plt.imshow(img)
    plt.axis('off')
    plt.title('img')

    plt.subplot(1, 4, 2)
    plt.imshow(mask)
    plt.axis('off')
    plt.title('mask')

    plt.subplot(1, 4, 3)
    plt.imshow(normal)
    plt.axis('off')
    plt.title('normal')

    plt.subplot(1, 4, 4)
```

(continues on next page)

(continued from previous page)

```
    plt.imshow(diffuse)
    plt.axis('off')
    plt.title('diffuse')
```

Hint: When using Unreal Engine, if the images of the mask look weird, try running the notebook again.

Finally, close the engine by:

[ ]:
```
# Close the engine
xf_runner.close()
```

Ref to api docs, you can always use `with` statement to ensure the engine is closed when the codes are finished.

Tutorial03 - Human NeRF

Overview

This tutorial provides an example of rendering animated skeletal meshes from different points of view. The rendered images can support various research topics, including human pose and shape estimation (HPS) and novel view synthesis for human (Human NeRF). By the end of this tutorial, you will be able to:

- Initialize XRFeitoria
- Import a skeletal mesh with animation
- Import another skeletal mesh without animation and setup animation for it
- Set `Actor`'s location in the `Level`
- Create a `Sequence` for rendering
- Add multiple static cameras in the `Sequence`
- Add a moving camera with transform keys in the `Sequence`
- Render images and annotations

1. Initialization

Then, similar to *Tutorial01*, specify your engine path and initialize XRFeitoria.

[ ]:
```
import xrfeitoria as xf
```

[ ]:
```
# Replace with your executable path
engine_exec_path = 'C:/Program Files/Blender Foundation/Blender 3.3/blender.exe'
# engine_exec_path = 'C:/Program Files/Epic Games/UE_5.2/Engine/Binaries/Win64/
↪UnrealEditor-Cmd.exe'
```

[ ]:
```
from pathlib import Path

exec_path_stem = Path(engine_exec_path).stem.lower()
if 'blender' in exec_path_stem:
    # Open Blender
    render_engine = 'blender'
```

(continues on next page)

(continued from previous page)

```
    xf_runner = xf.init_blender(exec_path=engine_exec_path, background=False, new_
↪process=True)
elif 'unreal' in exec_path_stem:
    # Unreal Engine requires a project to be opened
    # Here we use a sample project, which is downloaded from the following link
    # You can also use your own project
    import shutil

    from xrfeitoria.utils.downloader import download
    unreal_project_zip = download(url='https://openxrlab-share.oss-cn-hongkong.aliyuncs.
↪com/xrfeitoria/tutorials/unreal_project/UE_Sample.zip',
                                  dst_dir="./tutorial03/assets/")
    shutil.unpack_archive(filename=unreal_project_zip, extract_dir='./tutorial03/assets/
↪')

    # Open Unreal Engine
    render_engine = 'unreal'
    xf_runner = xf.init_unreal(exec_path=engine_exec_path,
                               background=False,
                               new_process=True,
                               project_path='./tutorial03/assets/UE_sample/UE_sample.
↪uproject')
```

Now you can see a new Blender/Unreal Engine process has started.

2. Import skeletal meshes to Level

Download the skeletal meshes in SynBody to local folder and import them to `Level`.

[ ]:
```
from xrfeitoria.utils.downloader import download

# Download the skeletal meshes
actor1_path = download('https://openxrlab-share.oss-cn-hongkong.aliyuncs.com/xrfeitoria/
↪tutorials/assets/SMPL-XL/SMPL-XL-00439__Subject_75_F_12.fbx', dst_dir="./tutorial03/
↪assets/")
actor2_path = download('https://openxrlab-share.oss-cn-hongkong.aliyuncs.com/xrfeitoria/
↪tutorials/assets/SMPL-XL/SMPL-XL-00045.fbx', dst_dir="./tutorial03/assets/")
actor2_motion_path = download('https://openxrlab-share.oss-cn-hongkong.aliyuncs.com/
↪xrfeitoria/tutorials/assets/SMPL-XL/walking__15_01.fbx', dst_dir="./tutorial03/assets/
↪")
```

Here we import two actors. The `actor1` has animation and the `actor2` has no animation.

And we set different stencil value for each `Actor` to distinguish different Actors when rendering segmentation masks.

[ ]:
```
# Import the skeletal mesh
actor1 = xf_runner.Actor.import_from_file(file_path=actor1_path, stencil_value=100)
actor2 = xf_runner.Actor.import_from_file(file_path=actor2_path, stencil_value=200)
```

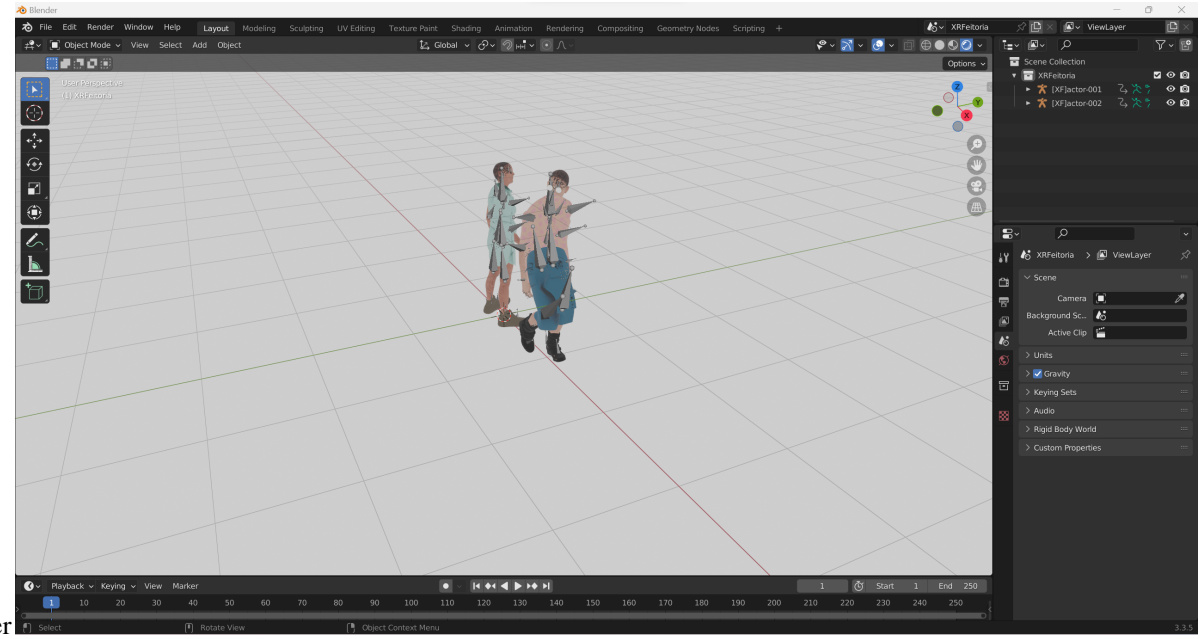
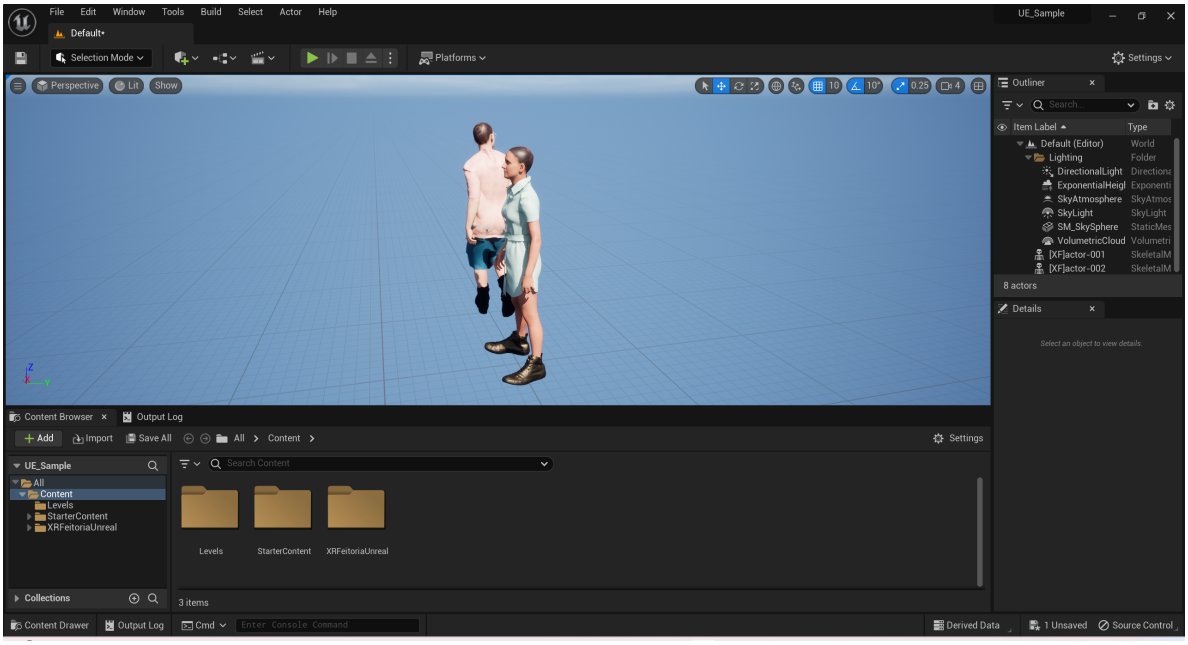
Then, we load an animation from another file and set it to the `actor2`

[ ]:
```
actor2.setup_animation(animation_path=actor2_motion_path)
```

We can also modify the properties of the actors, such as the location, rotation, and scale.

```
[ ]: # Set the location of the two actors to make their distance to be 1.0 meter
actor1_location = actor1.location
actor2_location = actor2.location
actor2.location = (actor1_location[0] + 1.0, actor1_location[1], actor1_location[2])
```

Now they look like:

- Blender 
- Unreal Engine 

If you use `Unreal Engine`, the `Level` should be saved after been modified.

```
[ ]: # save the level
if render_engine == 'unreal':
```

(continues on next page)

(continued from previous page)

```
    xf_runner.utils.save_current_level()
```

3. Add a sequence for rendering

Sequence is a multifunctional class in XRFeitoria. It can be used for: - rendering - adding transform keys - grouping different objects.

Here, we use it for rendering.

[ ]:
```
import math

from xrfeitoria.data_structure.models import RenderPass
from xrfeitoria.data_structure.models import SequenceTransformKey as SeqTransKey

# Use `with` statement to create a sequence, and it will be automatically close the␣
↪sequence after the code block is executed.
# The argument `seq_length` controls the number of frames to be rendered.
sequence_name = 'MySequence'
frame_num = 6
with xf_runner.Sequence.new(seq_name=sequence_name, seq_length=frame_num, replace=True)␣
↪as seq:

    # Get the bounding boxes of the actors
    actor1_bbox = actor1.bound_box
    actor2_bbox = actor2.bound_box

    # Get the center location of the actors
    actor1_center = ((actor1_bbox[0][0] + actor1_bbox[1][0]) / 2, (actor1_bbox[0][1] +␣
↪actor1_bbox[1][1]) / 2, (actor1_bbox[0][2] + actor1_bbox[1][2]) / 2)
    actor2_center = ((actor2_bbox[0][0] + actor2_bbox[1][0]) / 2, (actor2_bbox[0][1] +␣
↪actor2_bbox[1][1]) / 2, (actor2_bbox[0][2] + actor2_bbox[1][2]) / 2)
    actors_center = ((actor1_center[0] + actor2_center[0]) / 2, (actor1_center[1] +␣
↪actor2_center[1]) / 2, (actor1_center[2] + actor2_center[2]) / 2)

    ###########################################################################
    # Add 6 static cameras and a moving camera around the actors for rendering
    ###########################################################################
    # Set cameras' field of view to 90°
    camera_fov = 90
    # Set cameras' distance to 3.0m
    distance_to_actor = 3.0
    # Prepare the transform keys for moving camera
    transform_keys = []
    # calculate the location and rotation of the cameras
    for i in range(6):
        azimuth = 360 / 6 * i
        azimuth_radians = math.radians(azimuth)

        x = distance_to_actor * math.cos(azimuth_radians) + actors_center[0]
        y = distance_to_actor * math.sin(azimuth_radians) + actors_center[1]
        z = 0.0 + actors_center[2]
```

(continues on next page)

(continued from previous page)

```
        location = (x, y, z)
        # Set camera's rotation to look at the actor's center
        rotation = xf_runner.utils.get_rotation_to_look_at(location=location,␣
↪target=actors_center)

        # Add a static camera
        static_camera = seq.spawn_camera(
            camera_name=f'static_camera_{i}',
            location=location,
            rotation=rotation,
            fov=camera_fov,
        )

        # Add a transform key to the moving camera
        transform_keys.append(
            SeqTransKey(
                frame=i,
                location=location,
                rotation=rotation,
                interpolation='AUTO',
            )
        )

    # Add a moving camera rotating around the actors
    moving_camera = seq.spawn_camera_with_keys(
        camera_name=f'moving_camera',
        transform_keys=transform_keys,
        fov=camera_fov,
    )

    # Add a render job to renderer
    # In render job, you can specify the output path, resolution, render passes, etc.
    # The output path is the path to save the rendered data.
    # The resolution is the resolution of the rendered image.
    # The render passes define what kind of data you want to render, such as img, depth,␣
↪normal, etc.
    # and what kind of format you want to save, such as png, exr, etc.
    seq.add_to_renderer(
        output_path=f'./tutorial03/outputs/{render_engine}/',
        resolution=(1280, 720),
        render_passes=[RenderPass('img', 'png'),
                       RenderPass('mask', 'exr'),
                       RenderPass('normal', 'exr'),
                       RenderPass('diffuse', 'exr')]
    )
```

4. Render

The following code renders all the render jobs and save the images to the `output_path` set in `seq.add_to_renderer` above.

```
[ ]: # Render
xf_runner.render()
```

Check the `output_path`, and you can see that for the frame *i*, the image rendered by the moving camera is the same as the image rendered by the *i*th static camera. For example, the `moving_camera/0002.png` is the same as the `static_camera_2/0002.png`.

```
[ ]: import matplotlib.pyplot as plt

from xrfeitoria.utils.viewer import Viewer

xf_viewer = Viewer(sequence_dir=f'./tutorial03/outputs/{render_engine}/{sequence_name}/')

moving_camera_img = xf_viewer.get_img(camera_name='moving_camera', frame=2)
static_camera_img = xf_viewer.get_img(camera_name='static_camera_2', frame=2)

plt.figure(figsize=(20, 20))

plt.subplot(1, 2, 1)
plt.imshow(moving_camera_img)
plt.axis('off')
plt.title('moving_camera/0002.png')

plt.subplot(1, 2, 2)
plt.imshow(static_camera_img)
plt.axis('off')
plt.title('static_camera_2/0002.png')
```

View the rendered images and annotations of the camera `static_camera_2` by:

```
[ ]: import matplotlib.pyplot as plt

from xrfeitoria.utils.viewer import Viewer

xf_viewer = Viewer(sequence_dir=f'./tutorial03/outputs/{render_engine}/{sequence_name}/')

camera_name = 'static_camera_2'
for i in range(frame_num):
    img = xf_viewer.get_img(camera_name=camera_name, frame=i)
    mask = xf_viewer.get_mask(camera_name=camera_name, frame=i)
    normal = xf_viewer.get_normal(camera_name=camera_name, frame=i)
    diffuse = xf_viewer.get_diffuse(camera_name=camera_name, frame=i)

    plt.figure(figsize=(20, 20))

    plt.subplot(1, 4, 1)
    plt.imshow(img)
    plt.axis('off')
```

(continues on next page)

(continued from previous page)

```
    plt.title('img')

    plt.subplot(1, 4, 2)
    plt.imshow(mask)
    plt.axis('off')
    plt.title('mask')

    plt.subplot(1, 4, 3)
    plt.imshow(normal)
    plt.axis('off')
    plt.title('normal')

    plt.subplot(1, 4, 4)
    plt.imshow(diffuse)
    plt.axis('off')
    plt.title('diffuse')
```

Hint: When using Unreal Engine, if the images of the mask look weird, try running the notebook again.

Finally, close the engine by:

```
[ ]: xf_runner.close()
```

Ref to api docs, you can always use `with` statement to ensure the engine is closed when the codes are finished.

Extra Samples

For more advanced examples, see the samples section.

2.1.3 xrfeitoria

Initialize firstly and run xrfeitoria.

Blender

Unreal

```
import xrfeitoria as xf
with xf.init_blender() as xf_runner:
    ...
```

`xf_runner` is an instance of *`XRFeitoriaBlender`*, where contains all the classes and methods to run xrfeitoria.

```
import xrfeitoria as xf
with xf.init_unreal() as xf_runner:
    ...
```

`xf_runner` is an instance of *`XRFeitoriaUnreal`*, where contains all the classes and methods to run xrfeitoria.

After initialized, use members of `xf_runner`.

Ref to members of *XRFeitoriaBlender* and *XRFeitoriaUnreal*.

See also:

Ref to *Tutorial01*.

Ref to docs of *xrfeitoria.init_blender* and *xrfeitoria.init_unreal*.

2.1.4 xrfeitoria.factory

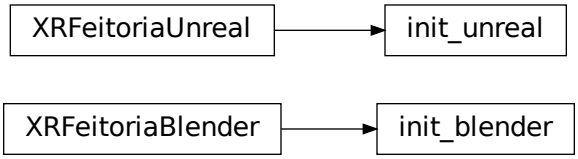


Fig. 1: Inheritance diagram

xrfeitoria.factory.init_blender([...])	Initialize Blender with `XRFeitoria`, which would start Blender as RPC server, and yield a *XRFeitoriaBlender* object.
xrfeitoria.factory.init_unreal([exec_path, ...])	Initialize Unreal with `XRFeitoria`, which would start Unreal as RPC server, and yield a *XRFeitoriaUnreal* object.
xrfeitoria.factory.XRFeitoriaBlender([...])	Factory class contains all the classes and functions for Blender.
xrfeitoria.factory.XRFeitoriaUnreal([...])	Factory class contains all the classes and functions for Unreal.

init_blender

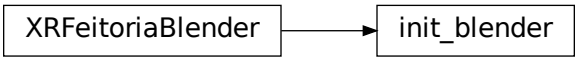


Fig. 2: Inheritance diagram

class xrfeitoria.factory.**init_blender**(*new_process: bool = False*, *exec_path: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *cleanup: bool = True*)

Initialize Blender with `XRFeitoria`, which would start Blender as RPC server, and yield a *XRFeitoriaBlender* object.

Examples

- Use as context manager:

```
>>> import xrfeitoria as xf
>>> with xf.init_blender() as xf_runner:
>>>     ...
```

- Or use directly:

```
>>> import xrfeitoria as xf
>>> xf_runner = xf.init_blender()
>>> ...
>>> xf_runner.close()
```

Yields

XRFeitoriaBlender – XRFeitoria Factory class for Blender.

__init__(*new_process: bool = False*, *exec_path: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *cleanup: bool = True*) → None

Giving arguments to initialize.

Parameters

- **new_process** (*bool, optional*) – Whether to start Blender in a new process. Defaults to False.
- **exec_path** (*Optional[PathLike], optional*) – Path to Blender executable. Defaults to None.
- **project_path** (*Optional[PathLike], optional*) – Path to Blender project. Defaults to None.
- **background** (*bool, optional*) – Whether to start Blender in background. Defaults to False.
- **reload_rpc_code** (*bool, optional*) – whether to reload the registered rpc functions and classes. If you are developing the package or writing a custom remote function, set this to True to reload the code. This will only be in effect when *new_process=False* if the engine process is reused. Defaults to False.
- **replace_plugin** (*bool, optional*) – Whether to replace the plugin. Defaults to False.
- **dev_plugin** (*bool, optional*) – Whether to use the plugin under local directory. If False, would use the plugin downloaded from a remote server. Defaults to False.
- **cleanup** (*bool, optional*) – Whether to clean up the scene. Defaults to True.

Note: If `dev_plugin=True`, the plugin under local directory would be used, which is under `src/XRFeitoriaBlender`. You should git clone first, and then use this option if you want to develop the plugin. Please ref to *How to use the plugin of Blender/Unreal under development*.

```
git clone https://github.com/openxrlab/xrfeitoria.git
cd xrfeitoria
pip install -e .
python -c "import xrfeitoria as xf; xf.init_blender(replace_plugin=True, dev_
↪plugin=True)"
```

init_unreal

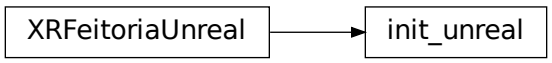


Fig. 3: Inheritance diagram

---

class xrfeitoria.factory.**init_unreal**(*exec_path: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *new_process: bool = False*)

Initialize Unreal with `XRFeitoria`, which would start Unreal as RPC server, and yield a *XRFeitoriaUnreal* object.

Examples

- Use as context manager:

```
>>> import xrfeitoria as xf
>>> with xf.init_unreal() as xf_runner:
>>>     ...
```

- Or use directly:

```
>>> import xrfeitoria as xf
>>> xf_runner = xf.init_unreal()
>>> ...
>>> xf_runner.close()
```

Yields

XRFeitoriaUnreal – XRFeitoria Factory class for Unreal.

__init__(*exec_path: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *new_process: bool = False*) → None

Giving arguments to initialize.

Parameters

- **exec_path** (*Optional[PathLike], optional*) – Path to Unreal executable. Defaults to None.
- **project_path** (*Optional[PathLike], optional*) – Path to Unreal project. Defaults to None.
- **background** (*bool, optional*) – Whether to start Unreal in background. Defaults to False.
- **reload_rpc_code** (*bool, optional*) – whether to reload the registered rpc functions and classes. If you are developing the package or writing a custom remote function, set this to True to reload the code. This will only be in effect when *new_process=False* if the engine process is reused. Defaults to False.
- **replace_plugin** (*bool, optional*) – Whether to replace the plugin. Defaults to False.
- **dev_plugin** (*bool, optional*) – Whether to use the plugin under local directory. If False, would use the plugin downloaded from a remote server. Defaults to False.
- **new_process** (*bool, optional*) – Whether to start Unreal in a new process. Defaults to False.

Note: If `dev_plugin=True`, the plugin under local directory would be used, which is under `src/XRFeitoriaUnreal`. You should git clone first, and then use this option if you want to develop the plugin. Please ref to *How to use the plugin of Blender/Unreal under development*.

```
git clone https://github.com/openxrlab/xrfeitoria.git
cd xrfeitoria
pip install -e .
python -c "import xrfeitoria as xf; xf.init_unreal(replace_plugin=True, dev_
↪plugin=True)"
```

---

close() → None

Close the RPC server.

XRFeitoriaBlender

---

class xrfeitoria.factory.**XRFeitoriaBlender**(*engine_exec: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *new_process: bool = False*)

Factory class contains all the classes and functions for Blender.

Members:

ObjectUtils: Object utilities.

Camera: Camera class.

Actor: Actor class.

Shape: Shape wrapper class.

Renderer: Renderer class.

sequence: Sequence wrapper function.

utils: Utilities functions executed in Blender.

render: Render jobs.

__init__(*engine_exec: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *new_process: bool = False*) → None

Giving arguments to initialize.

Parameters

- **new_process** (*bool, optional*) – Whether to start Blender in a new process. Defaults to False.
- **exec_path** (*Optional[PathLike], optional*) – Path to Blender executable. Defaults to None.
- **project_path** (*Optional[PathLike], optional*) – Path to Blender project. Defaults to None.
- **background** (*bool, optional*) – Whether to start Blender in background. Defaults to False.
- **reload_rpc_code** (*bool, optional*) – whether to reload the registered rpc functions and classes. If you are developing the package or writing a custom remote function, set this to True to reload the code. This will only be in effect when *new_process=False* if the engine process is reused. Defaults to False.
- **replace_plugin** (*bool, optional*) – Whether to replace the plugin. Defaults to False.
- **dev_plugin** (*bool, optional*) – Whether to use the plugin under local directory. If False, would use the plugin downloaded from a remote server. Defaults to False.
- **cleanup** (*bool, optional*) – Whether to clean up the scene. Defaults to True.

XRFeitoriaUnreal

---

class xrfeitoria.factory.**XRFeitoriaUnreal**(*engine_exec: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *new_process: bool = False*)

Factory class contains all the classes and functions for Unreal.

Members:

ObjectUtils: Object utilities.

Camera: Camera class.

Actor: Actor class.

Shape: Shape wrapper class.

Renderer: Renderer class.

sequence: Sequence wrapper function.

utils: Utilities functions executed in Unreal.

render: Render jobs.

__init__(*engine_exec: str | Path | None = None*, *project_path: str | Path | None = None*, *background: bool = False*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *new_process: bool = False*) → None

Giving arguments to initialize.

Parameters

- **exec_path** (*Optional[PathLike], optional*) – Path to Unreal executable. Defaults to None.
- **project_path** (*Optional[PathLike], optional*) – Path to Unreal project. Defaults to None.
- **background** (*bool, optional*) – Whether to start Unreal in background. Defaults to False.
- **reload_rpc_code** (*bool, optional*) – whether to reload the registered rpc functions and classes. If you are developing the package or writing a custom remote function, set this to True to reload the code. This will only be in effect when *new_process=False* if the engine process is reused. Defaults to False.
- **replace_plugin** (*bool, optional*) – Whether to replace the plugin. Defaults to False.
- **dev_plugin** (*bool, optional*) – Whether to use the plugin under local directory. If False, would use the plugin downloaded from a remote server. Defaults to False.
- **new_process** (*bool, optional*) – Whether to start Unreal in a new process. Defaults to False.

2.1.5 xrfeitoria.object

xrfeitoria.object.object_base.ObjectBase(name)	Base class for all objects in the world.
xrfeitoria.object.object_utils.ObjectUtilsBase()	Base class for object utils.
xrfeitoria.object.object_utils.ObjectUtilsBlender()	Object utils class for Blender.
xrfeitoria.object.object_utils.ObjectUtilsUnreal()	Object utils class for Unreal.

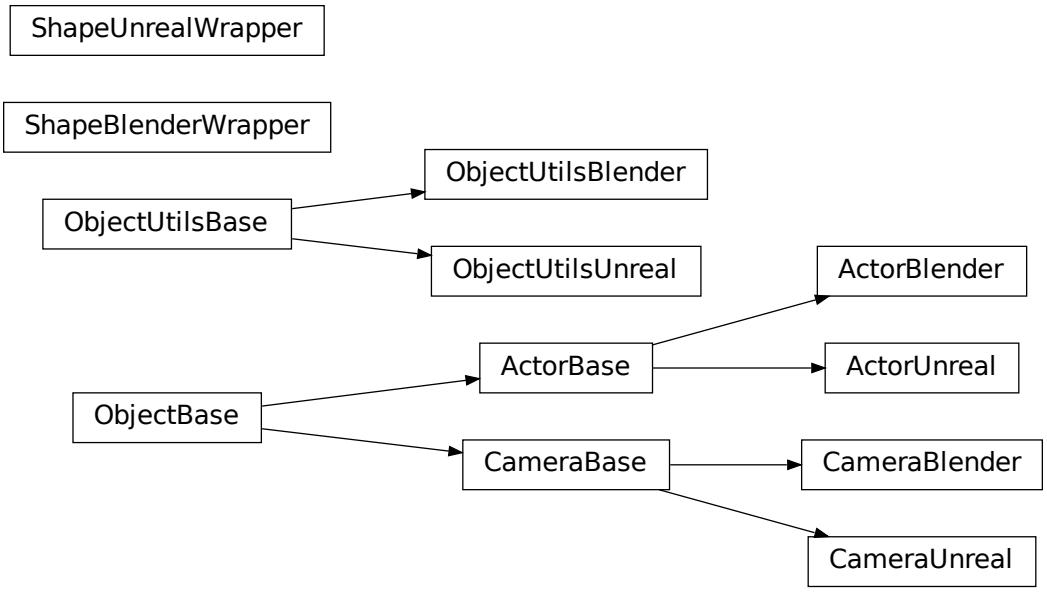


Fig. 4: Inheritance diagram

ObjectBase

---

class xrfeitoria.object.object_base.**ObjectBase**(*name: str*)

Base class for all objects in the world.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

set_transform(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- **location** (*Tuple[float, float, float]*) – in meters

- **rotation** (*Tuple[float, float, float]*) – in degrees
- **scale** (*Tuple[float, float, float]*) – no units, 1.0 is default

property location: Tuple[float, float, float]

Location of the object in the world, in meters.

property name: str

Name of the object.

property rotation: Tuple[float, float, float]

Rotation of the object in the world, in degrees.

property scale: Tuple[float, float, float]

Scale of the object in the world, no units, 1.0 is default.

ObjectUtilsBase

---

class xrfeitoria.object.object_utils.**ObjectUtilsBase**

Base class for object utils.

classmethod get_bound_box(*name: str*) → Tuple[float, float, float]

Get bounding box of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Bounding box of the object.

classmethod get_dimensions(*name: str*) → Tuple[float, float, float]

Get dimensions of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Dimensions of the object.

classmethod get_location(*name: str*) → Tuple[float, float, float]

Get location of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Location of the object.

classmethod get_rotation(*name: str*) → Tuple[float, float, float]

Get rotation of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Rotation of the object.

classmethod get_scale(*name: str*) → Tuple[float, float, float]

Get scale of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Scale of the object.

classmethod get_transform(*name: str*) → Tuple[List, List, List]

Get transform (location, rotation, scale) of the object.

Parameters
name (*str*) – Name of the object.

Returns
Transform – (location 3D vector, rotation 3D vector, scale 3D vector). location: Location of the object. unit: meters. rotation: Rotation of the object. unit: degrees. scale: Scale of the object.

classmethod set_location(*name: str*, *location: Tuple[float, float, float]*)

Set location of the object.

Parameters

- **name** (*str*) – Name of the object.
- **location** (*Vector*) – Location of the object.

classmethod set_name(*name: str*, *new_name: str*)

Set a new name for the object.

Parameters

- **name** (*str*) – Original name of the object.
- **new_name** (*str*) – New name of the object.

classmethod set_rotation(*name: str*, *rotation: Tuple[float, float, float]*)

Set rotation of the object.

Parameters

- **name** (*str*) – Name of the object.
- **rotation** (*Vector*) – Rotation of the object.

classmethod set_scale(*name*, *scale: Tuple[float, float, float]*)

Set scale of the object.

Parameters

- **name** (*str*) – Name of the object.
- **scale** (*Vector*) – Scale of the object.

classmethod set_transform(*name: str*, *location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*)

Set transform (location, rotation, scale) of the object.

Parameters

- **name** (*str*) – Name of the object.
- **location** (*Vector*) – Location of the object.

- **rotation** (*Vector*) – Rotation of the object.
- **scale** (*Vector*) – Scale of the object.

ObjectUtilsBlender

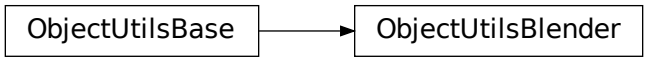


Fig. 5: Inheritance diagram

---

class xrfeitoria.object.object_utils.**ObjectUtilsBlender**

Object utils class for Blender.

classmethod get_bound_box(*name: str*) → Tuple[float, float, float]

Get bounding box of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Bounding box of the object.

classmethod get_dimensions(*name: str*) → Tuple[float, float, float]

Get dimensions of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Dimensions of the object.

classmethod get_location(*name: str*) → Tuple[float, float, float]

Get location of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Location of the object.

classmethod get_rotation(*name: str*) → Tuple[float, float, float]

Get rotation of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Rotation of the object.

classmethod get_scale(*name: str*) → Tuple[float, float, float]

Get scale of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Scale of the object.

classmethod get_transform(*name: str*) → Tuple[List, List, List]

Get transform (location, rotation, scale) of the object.

Parameters
name (*str*) – Name of the object.

Returns
Transform – (location 3D vector, rotation 3D vector, scale 3D vector). location: Location of the object. unit: meters. rotation: Rotation of the object. unit: degrees. scale: Scale of the object.

classmethod set_dimensions(*name: str*, *dimensions: Tuple[float, float, float]*) → None

Set dimensions of the object.

Parameters

- **name** (*str*) – Name of the object.
- **dimensions** (*Vector*) – Dimensions of the object.

classmethod set_location(*name: str*, *location: Tuple[float, float, float]*)

Set location of the object.

Parameters

- **name** (*str*) – Name of the object.
- **location** (*Vector*) – Location of the object.

classmethod set_name(*name: str*, *new_name: str*)

Set a new name for the object.

Parameters

- **name** (*str*) – Original name of the object.
- **new_name** (*str*) – New name of the object.

classmethod set_origin(*name: str*) → None

Set origin of the object to its center.

Parameters
name (*str*) – Name of the object.

classmethod set_rotation(*name: str*, *rotation: Tuple[float, float, float]*)

Set rotation of the object.

Parameters

- **name** (*str*) – Name of the object.
- **rotation** (*Vector*) – Rotation of the object.

classmethod set_scale(*name*, *scale: Tuple[float, float, float]*)

Set scale of the object.

Parameters

- **name** (*str*) – Name of the object.
- **scale** (*Vector*) – Scale of the object.

classmethod set_transform(*name: str*, *location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*)

Set transform (location, rotation, scale) of the object.

Parameters

- **name** (*str*) – Name of the object.
- **location** (*Vector*) – Location of the object.
- **rotation** (*Vector*) – Rotation of the object.
- **scale** (*Vector*) – Scale of the object.

classmethod set_transform_keys(*name: str*, *transform_keys: List[Dict]*)

Set keyframe of the object.

Parameters

- **name** (*str*) – Name of the object.
- **transform_keys** (*List[Dict]*) – Keyframes of transform (frame, location, rotation, scale, and interpolation).

ObjectUtilsUnreal

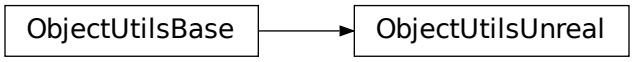


Fig. 6: Inheritance diagram

---

class xrfeitoria.object.object_utils.**ObjectUtilsUnreal**

Object utils class for Unreal.

classmethod get_bound_box(*name: str*) → Tuple[float, float, float]

Get bounding box of the object.

Parameters

name (*str*) – Name of the object.

Returns

Vector – Bounding box of the object.

classmethod get_dimensions(*name: str*) → Tuple[float, float, float]

Get dimensions of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Dimensions of the object.

classmethod get_location(*name: str*) → Tuple[float, float, float]

Get location of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Location of the object.

classmethod get_rotation(*name: str*) → Tuple[float, float, float]

Get rotation of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Rotation of the object.

classmethod get_scale(*name: str*) → Tuple[float, float, float]

Get scale of the object.

Parameters
name (*str*) – Name of the object.

Returns
Vector – Scale of the object.

classmethod get_transform(*name: str*) → Tuple[List, List, List]

Get transform (location, rotation, scale) of the object.

Parameters
name (*str*) – Name of the object.

Returns
Transform – (location 3D vector, rotation 3D vector, scale 3D vector). location: Location of the object. unit: meters. rotation: Rotation of the object. unit: degrees. scale: Scale of the object.

classmethod set_location(*name: str*, *location: Tuple[float, float, float]*)

Set location of the object.

Parameters

- **name** (*str*) – Name of the object.
- **location** (*Vector*) – Location of the object.

classmethod set_name(*name: str*, *new_name: str*)

Set a new name for the object.

Parameters

- **name** (*str*) – Original name of the object.

- **new_name** (*str*) – New name of the object.

classmethod set_rotation(*name: str*, *rotation: Tuple[float, float, float]*)

Set rotation of the object.

Parameters

- **name** (*str*) – Name of the object.
- **rotation** (*Vector*) – Rotation of the object.

classmethod set_scale(*name*, *scale: Tuple[float, float, float]*)

Set scale of the object.

Parameters

- **name** (*str*) – Name of the object.
- **scale** (*Vector*) – Scale of the object.

classmethod set_transform(*name: str*, *location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*)

Set transform (location, rotation, scale) of the object.

Parameters

- **name** (*str*) – Name of the object.
- **location** (*Vector*) – Location of the object.
- **rotation** (*Vector*) – Rotation of the object.
- **scale** (*Vector*) – Scale of the object.

2.1.6 xrfeitoria.actor

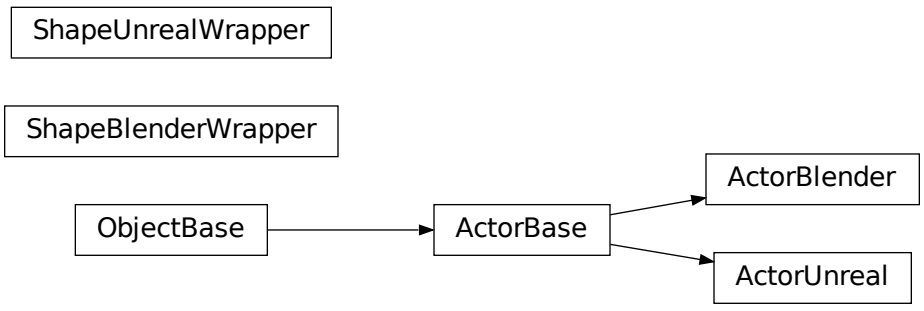


Fig. 7: Inheritance diagram

xrfeitoria.actor.actor_base.ActorBase(name)	Base class for all actors in the world.
xrfeitoria.actor.actor_blender.ActorBlender(name)	Actor class for Blender.
xrfeitoria.actor.actor_unreal.ActorUnreal(name)	Actor class for Unreal Engine.
xrfeitoria.actor.actor_blender.ShapeBlenderWrapper()	Wrapper class for shapes in Blender.
xrfeitoria.actor.actor_unreal.ShapeUnrealWrapper()	Wrapper class for shapes in Unreal Engine.

ActorBase

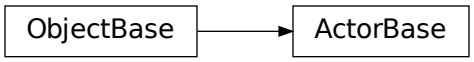


Fig. 8: Inheritance diagram

---

class xrfeitoria.actor.actor_base.**ActorBase**(*name: str*)

Base class for all actors in the world.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

classmethod import_from_file(*file_path: str | Path*, *actor_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*) → *ActorBase*

Imports an actor from a file and returns its corresponding actor.

For Blender, support files in types: fbx, obj, abc, ply, stl.

Note: For fbx file, Blender only support binary format. ASCII format is not supported. (Ref)

Parameters

- **path** (*PathLike*) – the path to the actor file.
- **name** (*str, optional*) – the name of the actor. Defaults to None.
- **location** (*Vector, optional*) – the location of the actor. Defaults to None. unit: meter
- **rotation** (*Vector, optional*) – the rotation of the actor. Defaults to None. unit: degree
- **scale** (*Vector, optional*) – the scale of the actor. Defaults to None.
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the actor. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBase – the actor object.

set_transform(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- **location** (*Tuple[float, float, float]*) – in meters
- **rotation** (*Tuple[float, float, float]*) – in degrees
- **scale** (*Tuple[float, float, float]*) – no units, 1.0 is default

setup_animation(*animation_path: str | Path*, *action_name: str | None = None*) → None

Load an animation from a file and setup for the actor.

For Blender, support files in types: fbx, blend, json.

Note: For fbx file, Blender only support binary format. ASCII format is not supported. (Ref)

Parameters

- **animation_path** (*PathLike*) – Animation file path.
- **action_name** (*Optional[str], optional*) – Name of the action in the animation file. Only required when the file type is `.blend `. Defaults to None.

property bound_box: Tuple[Tuple[float, float, float], Tuple[float, float, float]]

Bounding box of the actor in the world space.

property dimensions: Tuple[float, float, float]

Dimensions of the actor in the world space.

property location: Tuple[float, float, float]

Location of the object in the world, in meters.

property mask_color: Tuple[float, float, float]

Get the mask color of the actor.

RGB values (int) in [0, 255].

property name: str

Name of the object.

property rotation: Tuple[float, float, float]

Rotation of the object in the world, in degrees.

property scale: Tuple[float, float, float]

Scale of the object in the world, no units, 1.0 is default.

property stencil_value: int

Get the stencil value of the actor.

ActorBlender

Fig. 9: Inheritance diagram

---

class xrfeitoria.actor.actor_blender.**ActorBlender**(*name: str*)

Actor class for Blender.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

classmethod import_from_file(*file_path: str | Path*, *actor_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*) → *ActorBase*

Imports an actor from a file and returns its corresponding actor.

For Blender, support files in types: fbx, obj, abc, ply, stl.

Note: For fbx file, Blender only support binary format. ASCII format is not supported. (Ref)

Parameters

- **path** (*PathLike*) – the path to the actor file.
- **name** (*str, optional*) – the name of the actor. Defaults to None.
- **location** (*Vector, optional*) – the location of the actor. Defaults to None. unit: meter
- **rotation** (*Vector, optional*) – the rotation of the actor. Defaults to None. unit: degree
- **scale** (*Vector, optional*) – the scale of the actor. Defaults to None.
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the actor. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBase – the actor object.

set_origin_to_center() → None

Set origin of the object to its center.

set_transform(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- **location** (*Tuple[float, float, float]*) – in meters
- **rotation** (*Tuple[float, float, float]*) – in degrees
- **scale** (*Tuple[float, float, float]*) – no units, 1.0 is default

set_transform_keys(*transform_keys: List[*SequenceTransformKey*]* | SequenceTransformKey) → None

Set transform keys of actor.

Parameters

transform_keys (*List[Dict]*) – Keyframes of transform (frame, location, rotation, scale, and interpolation).

setup_animation(*animation_path: str* | *Path*, *action_name: str* | *None* = *None*) → None

Load an animation from a file and setup for the actor.

For Blender, support files in types: fbx, blend, json.

Note: For fbx file, Blender only support binary format. ASCII format is not supported. (Ref)

Parameters

- **animation_path** (*PathLike*) – Animation file path.
- **action_name** (*Optional[str], optional*) – Name of the action in the animation file. Only required when the file type is `.blend `. Defaults to None.

property bound_box: Tuple[Tuple[float, float, float], Tuple[float, float, float]]

Bounding box of the actor in the world space.

property dimensions: Tuple[float, float, float]

Dimensions of the actor in the world space.

property location: Tuple[float, float, float]

Location of the object in the world, in meters.

property mask_color: Tuple[float, float, float]

Get the mask color of the actor.

RGB values (int) in [0, 255].

property name: str

Name of the object.

property rotation: Tuple[float, float, float]

Rotation of the object in the world, in degrees.

property scale: Tuple[float, float, float]

Scale of the object in the world, no units, 1.0 is default.

property stencil_value: int

Get the stencil value of the actor.

ActorUnreal

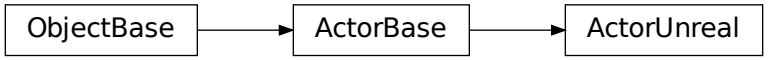


Fig. 10: Inheritance diagram

---

class xrfeitoria.actor.actor_unreal.**ActorUnreal**(*name: str*)

Actor class for Unreal Engine.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

classmethod import_from_file(*file_path: str | Path*, *actor_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*) → *ActorBase*

Imports an actor from a file and returns its corresponding actor.

For Blender, support files in types: fbx, obj, abc, ply, stl.

Note: For fbx file, Blender only support binary format. ASCII format is not supported. (Ref)

Parameters

- **path** (*PathLike*) – the path to the actor file.
- **name** (*str, optional*) – the name of the actor. Defaults to None.
- **location** (*Vector, optional*) – the location of the actor. Defaults to None. unit: meter
- **rotation** (*Vector, optional*) – the rotation of the actor. Defaults to None. unit: degree
- **scale** (*Vector, optional*) – the scale of the actor. Defaults to None.
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the actor. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBase – the actor object.

set_transform(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- **location** (*Tuple[float, float, float]*) – in meters
- **rotation** (*Tuple[float, float, float]*) – in degrees
- **scale** (*Tuple[float, float, float]*) – no units, 1.0 is default

setup_animation(*animation_path: str | Path*, *action_name: str | None = None*) → None

Load an animation from a file and setup for the actor.

For Blender, support files in types: fbx, blend, json.

Note: For fbx file, Blender only support binary format. ASCII format is not supported. (Ref)

Parameters

- **animation_path** (*PathLike*) – Animation file path.
- **action_name** (*Optional[str], optional*) – Name of the action in the animation file. Only required when the file type is `.blend `. Defaults to None.

classmethod spawn_from_engine_path(*engine_path: str*, *name: str | None = None*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*) → *ActorBase*

Spawns an actor in the engine and returns its corresponding actor.

Parameters

- **engine_path** (*str*) – the path to the actor in the engine. For example, '/Game/Engine/BasicShapes/Cube'.
- **name** (*str, optional*) – the name of the actor. Defaults to None, in which case a name will be generated.
- **location** (*Vector, optional*) – the location of the actor. Units are in meters. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – the rotation of the actor. Units are in degrees. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – the scale of the actor. Defaults to (1, 1, 1).

Returns

ActorBase – the actor that was spawned

property bound_box: Tuple[Tuple[float, float, float], Tuple[float, float, float]]

Bounding box of the actor in the world space.

property dimensions: Tuple[float, float, float]

Dimensions of the actor in the world space.

property engine_path: str

Engine path of the actor.

property location: Tuple[float, float, float]

Location of the object in the world, in meters.

property mask_color: Tuple[float, float, float]

Get the mask color of the actor.

RGB values (int) in [0, 255].

property name: str

Name of the object.

property rotation: Tuple[float, float, float]

Rotation of the object in the world, in degrees.

property scale: Tuple[float, float, float]

Scale of the object in the world, no units, 1.0 is default.

property stencil_value: int

Get the stencil value of the actor.

ShapeBlenderWrapper

---

class xrfeitoria.actor.actor_blender.**ShapeBlenderWrapper**

Wrapper class for shapes in Blender.

classmethod spawn(*type: Literal['plane', 'cube', 'sphere', 'ico_sphere', 'cylinder', 'cone']*, *name: str | None = None*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*, ***kwargs*) → *ActorBlender*

Spawn a shape(plane, cube, UV sphere, icosphere, cylinder or cone) in the scene.

Parameters

- **name** (*str*) – Name of the new added shape.
- **mesh_type** (*enum in ['plane', 'cube', 'sphere', 'icosphere', 'cylinder', 'cone']*) – Type of new spawn shape.
- **location** (*Vector, optional*) – Location of the shape. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the shape. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the shape. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the shape. Defaults to 1. Ref to *What is stencil_value* for details.
- ****kwargs** –
  - plane: size. Ref to *ShapeBlenderWrapper.spawn_plane* for details.
  - cube: size. Ref to *ShapeBlenderWrapper.spawn_cube* for details.
  - sphere: radius, segments, ring_count. Ref to *ShapeBlenderWrapper.spawn_sphere* for details.
  - icosphere: radius, subdivisions. Ref to *ShapeBlenderWrapper.spawn_ico_sphere* for details.
  - cylinder: radius, depth, vertices. Ref to *ShapeBlenderWrapper.spawn_cylinder* for details.
  - cone: radius1, radius2, depth, vertices. Ref to *ShapeBlenderWrapper.spawn_cone* for details.

Returns

ActorBlender – New added shape.

classmethod spawn_cone(*name: str*, *radius1: float = 1.0*, *radius2: float = 0.0*, *depth: float = 2.0*, *vertices: int = 32*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*) → *ActorBlender*

Spawn a cone in the engine.

Parameters

- **name** (*str*) – Name of the new added cone.
- **radius1** (*float in [0, inf], optional*) – Radius of the circular base of the cone. Defaults to 1.0. (unit: meter).
- **radius2** (*float in [0, inf], optional*) – Radius of the tip of the cone. Defaults to 0.0. (unit: meter).
- **depth** (*float in [0, inf], optional*) – Height of the cone. Defaults to 2.0. (unit: meter)
- **vertices** (*int in [3, 10000000], optional*) – Number of vertices on the circumference of the base of the cone. Defaults to 32.
- **location** (*Vector, optional*) – Location of the cone. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the cone. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the cone. Defaults to (1, 1, 1).

- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the cone. Defaults to 1. Ref to *What is stencil_value* for details.

Returns
ActorBlender – New added cone.

classmethod spawn_cube(*name: str*, *size: float = 1.0*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*) → *ActorBlender*

Spawn a cube in the engine.

Parameters

- **name** (*str*) – Name of the new added cube.
- **size** (*float in [0, inf], optional*) – Size of the cube. Defaults to 1.0. (unit: meter)
- **location** (*Vector, optional*) – Location of the cube. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the cube. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the cube. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the cube. Defaults to 1. Ref to *What is stencil_value* for details.

Returns
ActorBlender – New added cube.

classmethod spawn_cylinder(*name: str*, *radius: float = 1.0*, *depth: float = 2.0*, *vertices: int = 32*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*) → *ActorBlender*

Spawn a cylinder in the engine.

Parameters

- **name** (*str*) – Name of the new added cylinder.
- **radius** (*float in [0, inf], optional*) – Radius of the cylinder's bases. Defaults to 1.0. (unit: meter)
- **depth** (*float in [0, inf], optional*) – Height of the cylinder. Defaults to 2.0. (unit: meter)
- **vertices** (*int in [3, 10000000], optional*) – Number of vertices on the circumference of the base of the cylinder. Defaults to 32.
- **location** (*Vector, optional*) – Location of the cylinder. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the cylinder. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the cylinder. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the cylinder. Defaults to 1. Ref to *What is stencil_value* for details.

Returns
ActorBlender – New added cylinder.

classmethod spawn_ico_sphere(*name: str*, *radius: float = 1.0*, *subdivisions: int = 2*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*) → *ActorBlender*

Spawn an icosphere in the engine.

Parameters

- **name** (*str*) – Name of the new added icosphere.
- **radius** (*float in [0, inf], optional*) – Radius of the icosphere. Defaults to 1.0. (unit: meter)
- **subdivisions** (*int in [1, 10], optional*) – Number of times of splitting each triangular face on the sphere into four triangles. Defaults to 2.
- **location** (*Vector, optional*) – Location of the icosphere. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the icosphere. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the icosphere. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the icosphere. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBlender – New added icosphere.

classmethod spawn_plane(*name: str*, *size: float = 1.0*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*) → *ActorBlender*

Spawn a plane in the engine.

Parameters

- **name** (*str*) – Name of the new added plane.
- **size** (*float in [0, inf], optional*) – Size of the plane. Defaults to 1.0. (unit: meter)
- **location** (*Vector, optional*) – Location of the plane. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the plane. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the plane. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the plane. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBlender – New added plane.

classmethod spawn_sphere(*name: str*, *radius: float = 1.0*, *segments: int = 32*, *ring_count: int = 16*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*, *stencil_value: int = 1*) → *ActorBlender*

Spawn a UV sphere in the engine.

Parameters

- **name** (*str*) – Name of the new added UV sphere.
- **radius** (*float in [0, inf], optional*) – Radius of the UV sphere. Defaults to 1.0. (unit: meter)
- **segments** (*int in [3, 100000], optional*) – Number of vertical segments on the sphere. Defaults to 32.

- **ring_count** (*int in [3, 100000], optional*) – Number of horizontal segmentson the sphere. Defaults to 16.
- **location** (*Vector, optional*) – Location of the UV sphere. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the UV sphere. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the UV sphere. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the UV sphere. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBlender – New added UV sphere.

ShapeUnrealWrapper

---

class xrfeitoria.actor.actor_unreal.**ShapeUnrealWrapper**

Wrapper class for shapes in Unreal Engine.

classmethod **spawn**(*type: Literal['cube', 'sphere', 'cylinder', 'cone', 'plane']*, *name: str | None = None*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *scale: Tuple[float, float, float] = (1, 1, 1)*) → *ActorUnreal*

Spawns a shape in the engine and returns its corresponding actor.

Parameters

- **mesh_type** (*Literal['cube', 'sphere', 'cylinder', 'cone', 'plane']*) – the type of the shape.
- **name** (*Optional[str], optional*) – the name of the shape. Defaults to None.
- **location** (*Vector, optional*) – the location of the shape. Units are in meters. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – the rotation of the shape. Units are in degrees. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – the scale of the shape. Defaults to (1, 1, 1).

2.1.7 xrfeitoria.camera

---

xrfeitoria.camera.camera_base.CameraBase(name)	Base camera class.
xrfeitoria.camera.camera_blender.CameraBlender(name)	Camera class for Blender.
xrfeitoria.camera.camera_unreal.CameraUnreal(name)	Camera class for Unreal.
xrfeitoria.camera.camera_parameter.CameraParameter(K, R, T)	Camera parameter class for pinhole camera model.

Fig. 11: Inheritance diagram

CameraBase

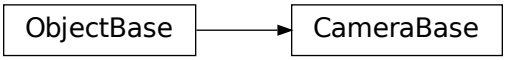


Fig. 12: Inheritance diagram

---

class xrfeitoria.camera.camera_base.**CameraBase**(*name: str*)

Base camera class.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

dump_params(*output_path: str | Path*) → None

Dump the intrinsic and extrinsic parameters of the camera to a file.

Parameters
output_path (*PathLike*) – path to the camera parameter file

get_KRT() → Tuple[List, List, Tuple[float, float, float]]

Get the intrinsic and extrinsic parameters of the camera.

Returns
Tuple[Matrix, Matrix, Vector] – K, R, T

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

`look_at`(*target: Tuple[float, float, float]*) → None

Set the camera to look at the target.

Parameters
`target` (*Vector*) – [x, y, z] coordinates of the target, in units of meters.

`set_KRT`(*K: List*, *R: List*, *T: Tuple[float, float, float]*) → None

Set the intrinsic and extrinsic parameters of the camera.

Parameters

- `K` (*Matrix*) – 3x3 intrinsic matrix
- `R` (*Matrix*) – 3x3 rotation matrix
- `T` (*Vector*) – 3x1 translation vector

`set_transform`(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- `location` (*Tuple[float, float, float]*) – in meters
- `rotation` (*Tuple[float, float, float]*) – in degrees
- `scale` (*Tuple[float, float, float]*) – no units, 1.0 is default

`classmethod spawn`(*camera_name: str | None = None*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *fov: float = 90.0*) → *CameraBase*

Spawn a camera in the engine.

Parameters

- `name` (*str, optional*) – name of the camera
- `location` (*Vector, optional*) – location of the camera. Defaults to (0, 0, 0).
- `rotation` (*Vector, optional*) – rotation of the camera. Defaults to (0, 0, 0).
- `fov` (*float, optional*) – field of view of the camera. Defaults to 90.0.

Returns
CameraBase – New camera

`property fov: float`

Field of view of the camera lens, in degrees.

`property location: Tuple[float, float, float]`

Location of the object in the world, in meters.

`property name: str`

Name of the object.

`property rotation: Tuple[float, float, float]`

Rotation of the object in the world, in degrees.

`property scale: Tuple[float, float, float]`

Scale of the object in the world, no units, 1.0 is default.

CameraBlender

Fig. 13: Inheritance diagram

---

class xrfeitoria.camera.camera_blender.**CameraBlender**(*name: str*)

Camera class for Blender.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

dump_params(*output_path: str | Path*) → None

Dump the intrinsic and extrinsic parameters of the camera to a file.

Parameters
output_path (*PathLike*) – path to the camera parameter file

get_KRT() → Tuple[List, List, Tuple[float, float, float]]

Get the intrinsic and extrinsic parameters of the camera.

Returns
Tuple[Matrix, Matrix, Vector] – K, R, T

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

look_at(*target: Tuple[float, float, float]*) → None

Set the camera to look at the target.

Parameters
target (*Vector*) – [x, y, z] coordinates of the target, in units of meters.

set_KRT(*K: List*, *R: List*, *T: Tuple[float, float, float]*) → None

Set the intrinsic and extrinsic parameters of the camera.

Parameters

- **K** (*Matrix*) – 3x3 intrinsic matrix
- **R** (*Matrix*) – 3x3 rotation matrix
- **T** (*Vector*) – 3x1 translation vector

set_transform(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- **location** (*Tuple[float, float, float]*) – in meters
- **rotation** (*Tuple[float, float, float]*) – in degrees
- **scale** (*Tuple[float, float, float]*) – no units, 1.0 is default

set_transform_keys(*transform_keys: List[*SequenceTransformKey*]* | SequenceTransformKey) → None

Set transform keys of actor.

Parameters

transform_keys (*List[Dict]*) – Keyframes of transform (frame, location, rotation, scale, and interpolation).

classmethod spawn(*camera_name: str | None = None*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *fov: float = 90.0*) → *CameraBase*

Spawn a camera in the engine.

Parameters

- **name** (*str, optional*) – name of the camera
- **location** (*Vector, optional*) – location of the camera. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – rotation of the camera. Defaults to (0, 0, 0).
- **fov** (*float, optional*) – field of view of the camera. Defaults to 90.0.

Returns

CameraBase – New camera

property active: bool

Activaty of the camera.

Only active cameras participate in rendering.

property fov: float

Field of view of the camera lens, in degrees.

property location: Tuple[float, float, float]

Location of the object in the world, in meters.

property name: str

Name of the object.

property rotation: Tuple[float, float, float]

Rotation of the object in the world, in degrees.

property scale: Tuple[float, float, float]

Scale of the object in the world, no units, 1.0 is default.

CameraUnreal

Fig. 14: Inheritance diagram

---

class xrfeitoria.camera.camera_unreal.**CameraUnreal**(*name: str*)

Camera class for Unreal.

__init__(*name: str*) → None

Parameters
name (*str*) – name of the object

delete()

Delete the object from the world.

dump_params(*output_path: str | Path*) → None

Dump the intrinsic and extrinsic parameters of the camera to a file.

Parameters
output_path (*PathLike*) – path to the camera parameter file

get_KRT() → Tuple[List, List, Tuple[float, float, float]]

Get the intrinsic and extrinsic parameters of the camera.

Returns
Tuple[Matrix, Matrix, Vector] – K, R, T

get_transform() → Tuple[List, List, List]

Get the transform of the object in the world.

Returns
Transform – location, rotation, scale

look_at(*target: Tuple[float, float, float]*) → None

Set the camera to look at the target.

Parameters
target (*Vector*) – [x, y, z] coordinates of the target, in units of meters.

set_KRT(*K: List*, *R: List*, *T: Tuple[float, float, float]*) → None

Set the intrinsic and extrinsic parameters of the camera.

Parameters

- **K** (*Matrix*) – 3x3 intrinsic matrix
- **R** (*Matrix*) – 3x3 rotation matrix
- **T** (*Vector*) – 3x1 translation vector

set_transform(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *scale: Tuple[float, float, float]*) → None

Set the transform of the object in the world.

Parameters

- **location** (*Tuple[float, float, float]*) – in meters
- **rotation** (*Tuple[float, float, float]*) – in degrees
- **scale** (*Tuple[float, float, float]*) – no units, 1.0 is default

classmethod spawn(*camera_name: str | None = None*, *location: Tuple[float, float, float] = (0, 0, 0)*, *rotation: Tuple[float, float, float] = (0, 0, 0)*, *fov: float = 90.0*) → *CameraBase*

Spawn a camera in the engine.

Parameters

- **name** (*str, optional*) – name of the camera
- **location** (*Vector, optional*) – location of the camera. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – rotation of the camera. Defaults to (0, 0, 0).
- **fov** (*float, optional*) – field of view of the camera. Defaults to 90.0.

Returns

CameraBase – New camera

property fov: float

Field of view of the camera lens, in degrees.

property location: Tuple[float, float, float]

Location of the object in the world, in meters.

property name: str

Name of the object.

property rotation: Tuple[float, float, float]

Rotation of the object in the world, in degrees.

property scale: Tuple[float, float, float]

Scale of the object in the world, no units, 1.0 is default.

CameraParameter

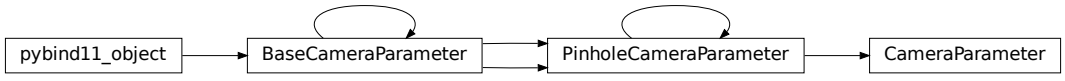


Fig. 15: Inheritance diagram

class xrfeitoria.camera.camera_parameter.**CameraParameter**(*K: List | ndarray*, *R: List | ndarray*, *T: List | ndarray*, *convention: str = 'opencv'*, *world2cam: bool = False*)

Camera parameter class for pinhole camera model.

Inherit from XRPrimer.

ClassName(*self: xrprimer_cpp.camera.BaseCameraParameter*) → str

LoadFile(*filename: str*) → bool

Load camera name and parameters from a dumped json file.

Parameters
filename (*str*) – Path to the dumped json file.

Returns
bool – True if load succeed.

SaveFile(*filename: str*) → bool

Dump camera name and parameters to a json file.

Parameters
filename (*str*) – Path to the dumped json file.

Returns
bool – True if save succeed.

__init__(*K: List | ndarray*, *R: List | ndarray*, *T: List | ndarray*, *convention: str = 'opencv'*, *world2cam: bool = False*)

A camera parameter class for pinhole camera model.

Parameters

- **K** (*Union[list, np.ndarray]*) – Nested list of float32, 4x4 or 3x3 K mat.
- **R** (*Union[list, np.ndarray]*) – Nested list of float32, 3x3 rotation mat.
- **T** (*Union[list, np.ndarray, None]*) – List of float32, T vector.
- **convention** (*str, optional*) – Convention name of this camera. Defaults to 'opencv'.
- **world2cam** (*bool, optional*) – Whether the R, T transform points from world space to camera space. Defaults to True.

clone() → *CameraParameter*

Clone a new CameraParameter instance like self.

Returns
CameraParameter

convert_convention(*dst: str*)

Convert the convention of this camera parameter. In-place.

Parameters
dst (*str*) – The name of destination convention. One of ['opencv', 'blender', 'unreal'].

dump(*filename: str*) → None

Dump camera name and parameters to a json file.

Parameters
filename (*str*) – Path to the dumped json file.

Raises
RuntimeError – Fail to dump a json file.

classmethod from_bin(*file: str | Path*) → *CameraParameter*

Construct a camera parameter data structure from a binary file.

Parameters
file (*PathLike*) – Path to the dumped binary file.

Returns
CameraParameter – An instance of CameraParameter class.

classmethod from_dict(*data: dict*) → *CameraParameter*

Construct a camera parameter data structure from a dict.

Parameters
data (*dict*) – The camera parameter data.

Returns
CameraParameter – An instance of CameraParameter class.

classmethod from_unreal_convention(*location: Tuple[float, float, float]*, *rotation: Tuple[float, float, float]*, *fov: float*, *image_size: Tuple[int, int]*) → *CameraParameter*

Converts camera parameters from Unreal Engine convention to CameraParameter object.

Parameters

- **location** (*Vector*) – The camera location in Unreal Engine convention.
- **rotation** (*Vector*) – The camera rotation in Unreal Engine convention.
- **fov** (*float*) – The camera field of view in degrees.
- **image_size** (*Tuple[int, int]*) – The size of the camera image in pixels (width, height).

Returns
CameraParameter – The converted camera parameters.

classmethod fromfile(*file: str | Path*) → *CameraParameter*

Construct a camera parameter data structure from a json file.

Parameters
filename (*PathLike*) – Path to the dumped json file.

Returns
CameraParameter – An instance of CameraParameter class.

get_extrinsic() → List

Get the camera matrix of RT.

Returns
List – A list of float32, 3x4 RT mat.

get_extrinsic_r() → list

Get extrinsic rotation matrix.

Returns
list – Nested list of float32, 3x3 R mat.

`get_extrinsic_t`() → list

Get extrinsic translation vector.

Returns
list – Nested list of float32, T vec of length 3.

`get_intrinsic`(*k_dim: int = 3*) → list

Get intrinsic K matrix.

Parameters
k_dim (*int, optional*) – If 3, returns a 3x3 mat. Else if 4, returns a 4x4 mat. Defaults to 3.

Raises
ValueError – k_dim is neither 3 nor 4.

Returns
list – Nested list of float32, 4x4 or 3x3 K mat.

`get_projection_matrix`() → List

Get the camera matrix of K@RT.

Returns
List – A list of float32, 3x4 K@RT mat.

`intrinsic33`() → ndarray

Get an intrinsic matrix in shape (3, 3).

Returns
ndarray – An ndarray of intrinsic matrix.

`inverse_extrinsic`() → None

Inverse the direction of extrinsics, between world to camera and camera to world.

`load`(*filename: str*) → None

Load camera name and parameters from a dumped json file.

Parameters
filename (*str*) – Path to the dumped json file.

Raises

- **FileNotFoundError** – File not found at filename.
- **ValueError** – Content in filename is not correct.

`model_dump`() → dict

Dump camera parameters to a dict.

`set_KRT`(*K: list | ndarray | None = None*, *R: list | ndarray | None = None*, *T: list | ndarray | None = None*, *world2cam: bool | None = None*) → None

Set K, R to matrix and T to vector.

Parameters

- **K** (*Union[list, np.ndarray, None]*) – Nested list of float32, 4x4 or 3x3 K mat. Defaults to None, intrisic will not be changed.
- **R** (*Union[list, np.ndarray, None]*) – Nested list of float32, 3x3 R mat. Defaults to None, extrisic_r will not be changed.
- **T** (*Union[list, np.ndarray, None]*) – List of float32, T vector. Defaults to None, extrisic_t will not be changed.

- **world2cam** (*Union[bool, None], optional*) – Whether the R, T transform points from world space to camera space. Defaults to None, self.world2cam will not be changed.

set_intrinsic(*mat3x3: list | ndarray | None = None*, *width: int | None = None*, *height: int | None = None*, *fx: float | None = None*, *fy: float | None = None*, *cx: float | None = None*, *cy: float | None = None*, *perspective: bool = True*) → None

Set the intrinsic of a camera. Note that mat3x3 has a higher priority than fx, fy, cx, cy.

Parameters

- **mat3x3** (*list, optional*) – A nested list of intrinsic matrix, in shape (3, 3). If mat is given, fx, fy, cx, cy will be ignored. Defaults to None.
- **width** (*int*) – Width of the screen.
- **height** (*int*) – Height of the screen.
- **fx** (*float, optional*) – Focal length. Defaults to None.
- **fy** (*float, optional*) – Focal length. Defaults to None.
- **cx** (*float, optional*) – Camera principal point. Defaults to None.
- **cy** (*float, optional*) – Camera principal point. Defaults to None.
- **perspective** (*bool, optional*) – Whether it is a perspective camera, if not, it's orthographics. Defaults to True.

set_resolution(*height: int*, *width: int*) → None

Set resolution of the camera.

Parameters

- **height** (*int*) – Height of the screen.
- **width** (*int*) – Width of the screen.

property convention

str

Type
transform convention, default is opencv

property extrinsic: ndarray[Any, dtype[float32]]

Get the extrinsic matrix of RT.

Returns
ndarray – An ndarray of float32, 3x4 RT mat.

property extrinsic_r

numpy.ndarray[numpy.float32[3, 3]] or list

Type
camera extrinsics R

property extrinsic_t

numpy.ndarray[numpy.float32[3, 1]] or list

Type
camera extrinsics T

property height

int

Type
camera image height

property intrinsic

numpy.ndarray[numpy.float32[4, 4]] or list

Type
camera intrinsic (4x4)

property name

camera tag name

property projection_matrix: ndarray[Any, dtype[float32]]

Get the camera matrix of K@RT.

Returns
ndarray – An ndarray of float32, 3x4 K@RT mat.

property width

int

Type
camera image width

property world2cam

bool

Type
world to camera flag

2.1.8 xrfeitoria.sequence

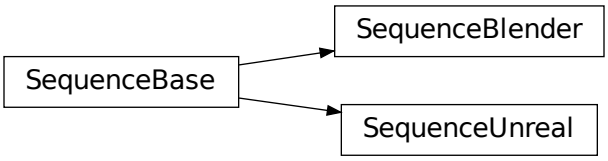


Fig. 16: Inheritance diagram

xrfeitoria.sequence.sequence_base.SequenceBase()	Base sequence class.
xrfeitoria.sequence.sequence_blender.SequenceBlender()	Sequence class for Blender.
xrfeitoria.sequence.sequence_unreal.SequenceUnreal()	Sequence class for Unreal.

SequenceBase

---

class xrfeitoria.sequence.sequence_base.**SequenceBase**

Base sequence class.

classmethod add_to_renderer(*output_path: str | Path*, *resolution: Tuple[int, int]*, *render_passes: List[RenderPass]*, ***kwargs*)

Add a rendering job with specific settings.

Parameters

- **output_path** (*PathLike*) – Output path of the rendered images.
- **resolution** (*Tuple[int, int]*) – Resolution of the rendered images.
- **render_passes** (*List[RenderPass]*) – Render passes.
- **kwargs** – Other engine specific arguments.

classmethod close() → None

Close the opened sequence.

classmethod import_actor(*file_path: str | Path*, *actor_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*) → *ActorBase*

Imports an actor from a file and adds it to the sequence.

Parameters

- **file_path** (*PathLike*) – The path to the file containing the actor to import.
- **actor_name** (*Optional[str], optional*) – The name to give the imported actor. If not provided, a name will be generated automatically. Defaults to None.
- **location** (*Vector, optional*) – The initial location of the actor. Defaults to None.
- **rotation** (*Vector, optional*) – The initial rotation of the actor. Defaults to None.
- **scale** (*Vector, optional*) – The initial scale of the actor. Defaults to None.
- **stencil_value** (*int, optional*) – The stencil value to use for the actor. Defaults to 1.

Returns

ActorBase – The imported actor.

classmethod spawn_camera(*location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *fov: float = 90.0*, *camera_name: str | None = None*) → *CameraBase*

Spawn a new camera in the sequence.

Parameters

- **location** (*Vector*) – Location of the camera.
- **rotation** (*Vector*) – Rotation of the camera.
- **fov** (*float in (0.0, 180.0), optional*) – Field of view of the camera len. Defaults to 90.0. (unit: degrees)
- **camera_name** (*str, optional*) – Name of the camera. Defaults to None.

classmethod spawn_camera_with_keys(*transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *fov: float = 90.0*, *camera_name: str | None = None*) → *CameraBase*

Spawn a new camera with keyframes in the sequence.

Parameters

- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **fov** (*float in (0.0, 180.0), optional*) – Field of view of the camera len. Defaults to 90.0. (unit: degrees)
- **camera_name** (*str, optional*) – Name of the camera. Defaults to 'Camera'.

classmethod spawn_shape(*type: Literal['plane', 'cube', 'sphere', 'cylinder', 'cone']*, *shape_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*, ***kwargs*) → *ActorBase*

Spawn a shape in the sequence.

Parameters

- **type** (*str*) – Type of new spawn shape. One of ["plane", "cube", "sphere", "cylinder", "cone"]
- **shape_name** (*str*) – Name of the new added shape. Defaults to None.
- **location** (*Vector, optional*) – Location of the shape. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the shape. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the shape. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the shape. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ShapeBlender or ShapeUnreal – New added shape.

classmethod spawn_shape_with_keys(*transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *type: Literal['plane', 'cube', 'sphere', 'cylinder', 'cone']*, *shape_name: str | None = None*, *stencil_value: int = 1*, ***kwargs*) → *ActorBase*

Spawn a shape with keyframes in the sequence.

Parameters

- **shape_name** (*str*) – Name of the new added shape.
- **type** (*str*) – Type of new spawn shape. One of ["plane", "cube", "sphere", "cylinder", "cone"]
- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the cone. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ShapeBlender or ShapeUnreal – New added shape.

classmethod use_actor(*actor: ActorBase*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int | None = None*, *anim_asset_path: str | None = None*) → None

Use the specified level actor in the sequence. The location, rotation, scale, stencil_value and anim_asset set in this method are only used in the sequence. These peoperties of the actor in the level will be restored after the sequence is closed.

Parameters

- **actor** (*ActorUnreal or ActorBlender*) – The actor to add to the sequence.
- **location** (*Optional[Vector]*) – The initial location of the actor. If None, the actor's current location is used. unit: meter.
- **rotation** (*Optional[Vector]*) – The initial rotation of the actor. If None, the actor's current rotation is used. unit: degree.
- **scale** (*Optional[Vector]*) – The initial scale of the actor. If None, the actor's current scale is used.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to None. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str]*) – For blender, this argument is the name of an action(bpy.types.Action).
- **Engine** (*For Unreal*) –
- **actor.** (*this argument is the engine path to the animation asset to use for the*) –
- **None** (*Default to None. If*) –
- **used** (*the actor's current animation is*) –
- **used.** (*else the specified animation is*) –

classmethod use_actor_with_keys(*actor: ActorBase*, *transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *stencil_value: int | None = None*, *anim_asset_path: str | None = None*) → None

Use the specified level actor in the sequence. The transform_keys, stencil_value and anim_asset set in this method are only used in the sequence. These peoperties of the actor in the level will be restored after the sequence is closed.

Parameters

- **actor** (*ActorUnreal or ActorBlender*) – The actor to use in the sequence.
- **transform_keys** (*Union[TransformKeys, List[TransformKeys]]*) – The transform keys to use with the actor.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to None. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str]*) – For blender, this argument is the name of an action(bpy.types.Action).
- **Engine** (*For Unreal*) –
- **actor.** (*this argument is the engine path to the animation asset to use for the*) –
- **None** (*Default to None. If*) –

- **used** (*the actor's current animation is*) –
- **used.** (*else the specified animation is*) –

classmethod use_camera(*camera:* CameraBase, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *fov: float | None = None*) → None

Use the specified level camera in the sequence. The location, rotation and fov set in this method are only used in the sequence. The location, rotation and fov of the camera in the level will be restored after the sequence is closed.

Parameters

- **camera** (*CameraUnreal or CameraBlender*) – The camera to use in the sequence.
- **location** (*Optional[Vector], optional*) – The location of the camera. Defaults to None. unit: meter.
- **rotation** (*Optional[Vector], optional*) – The rotation of the camera. Defaults to None. unit: degree.
- **fov** (*float, optional*) – The field of view of the camera. Defaults to None. unit: degree.

classmethod use_camera_with_keys(*camera:* CameraBase, *transform_keys: List[*SequenceTransformKey*] |* SequenceTransformKey, *fov: float | None = None*) → None

Use the specified level camera in the sequence. The transform_keys and fov set in this method are only used in the sequence. The location, rotation and fov of the camera in the level will be restored after the sequence is closed.

Parameters

- **camera** (*CameraUnreal or CameraBlender*) – The camera to use.
- **transform_keys** (*TransformKeys*) – The transform keys to use.
- **fov** (*float, optional*) – The field of view to use. Defaults to None. unit: degree.

SequenceBlender

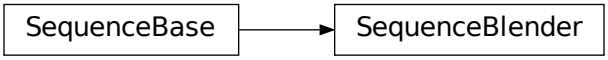


Fig. 17: Inheritance diagram

---

class xrfeitoria.sequence.sequence_blender.**SequenceBlender**

Sequence class for Blender.

classmethod add_to_renderer(*output_path: str | Path*, *resolution: Tuple[int, int]*, *render_passes: List[RenderPass]*, ***kwargs*)

Add a rendering job with specific settings.

Parameters

- **output_path** (*PathLike*) – Output path of the rendered images.
- **resolution** (*Tuple[int, int]*) – Resolution of the rendered images.
- **render_passes** (*List[RenderPass]*) – Render passes.
- **kwargs** – Other engine specific arguments.

classmethod close() → None

Close the opened sequence.

classmethod import_actor(*file_path: str | Path*, *actor_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*) → *ActorBase*

Imports an actor from a file and adds it to the sequence.

Parameters

- **file_path** (*PathLike*) – The path to the file containing the actor to import.
- **actor_name** (*Optional[str], optional*) – The name to give the imported actor. If not provided, a name will be generated automatically. Defaults to None.
- **location** (*Vector, optional*) – The initial location of the actor. Defaults to None.
- **rotation** (*Vector, optional*) – The initial rotation of the actor. Defaults to None.
- **scale** (*Vector, optional*) – The initial scale of the actor. Defaults to None.
- **stencil_value** (*int, optional*) – The stencil value to use for the actor. Defaults to 1.

Returns

ActorBase – The imported actor.

classmethod import_actor_with_keys(*file_path: str | Path*, *transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *actor_name: str | None = None*, *stencil_value: int = 1*) → *ActorBlender*

Import an actor from a file to the sequence and add transform keyframes to the it.

Parameters

- **name** (*str*) – Name of the actor in Blender.
- **path** (*PathLike*) – File path used for importing the actor.
- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the actor. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ActorBlender – New imported actor.

classmethod spawn_camera(*location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *fov: float = 90.0*, *camera_name: str | None = None*) → *CameraBase*

Spawn a new camera in the sequence.

Parameters

- **location** (*Vector*) – Location of the camera.
- **rotation** (*Vector*) – Rotation of the camera.
- **fov** (*float in (0.0, 180.0), optional*) – Field of view of the camera len. Defaults to 90.0. (unit: degrees)
- **camera_name** (*str, optional*) – Name of the camera. Defaults to None.

classmethod spawn_camera_with_keys(*transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *fov: float = 90.0*, *camera_name: str | None = None*) → *CameraBase*

Spawn a new camera with keyframes in the sequence.

Parameters

- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **fov** (*float in (0.0, 180.0), optional*) – Field of view of the camera len. Defaults to 90.0. (unit: degrees)
- **camera_name** (*str, optional*) – Name of the camera. Defaults to 'Camera'.

classmethod spawn_shape(*type: Literal['plane', 'cube', 'sphere', 'cylinder', 'cone']*, *shape_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*, ***kwargs*) → *ActorBase*

Spawn a shape in the sequence.

Parameters

- **type** (*str*) – Type of new spawn shape. One of ["plane", "cube", "sphere", "cylinder", "cone"]
- **shape_name** (*str*) – Name of the new added shape. Defaults to None.
- **location** (*Vector, optional*) – Location of the shape. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the shape. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the shape. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the shape. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ShapeBlender or ShapeUnreal – New added shape.

classmethod spawn_shape_with_keys(*transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *type: Literal['plane', 'cube', 'sphere', 'cylinder', 'cone']*, *shape_name: str | None = None*, *stencil_value: int = 1*, ***kwargs*) → *ActorBase*

Spawn a shape with keyframes in the sequence.

Parameters

- **shape_name** (*str*) – Name of the new added shape.
- **type** (*str*) – Type of new spawn shape. One of ["plane", "cube", "sphere", "cylinder", "cone"]
- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the cone. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ShapeBlender or ShapeUnreal – New added shape.

classmethod use_actor(*actor: ActorBase*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int | None = None*, *anim_asset_path: str | None = None*) → None

Use the specified level actor in the sequence. The location, rotation, scale, stencil_value and anim_asset set in this method are only used in the sequence. These peoperties of the actor in the level will be restored after the sequence is closed.

Parameters

- **actor** (*ActorUnreal or ActorBlender*) – The actor to add to the sequence.
- **location** (*Optional[Vector]*) – The initial location of the actor. If None, the actor's current location is used. unit: meter.
- **rotation** (*Optional[Vector]*) – The initial rotation of the actor. If None, the actor's current rotation is used. unit: degree.
- **scale** (*Optional[Vector]*) – The initial scale of the actor. If None, the actor's current scale is used.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to None. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str]*) – For blender, this argument is the name of an action(bpy.types.Action).
- **Engine** (*For Unreal*) –
- **actor.** (*this argument is the engine path to the animation asset to use for the*) –
- **None** (*Default to None. If*) –
- **used** (*the actor's current animation is*) –
- **used.** (*else the specified animation is*) –

classmethod use_actor_with_keys(*actor: ActorBase*, *transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *stencil_value: int | None = None*, *anim_asset_path: str | None = None*) → None

Use the specified level actor in the sequence. The transform_keys, stencil_value and anim_asset set in this method are only used in the sequence. These peoperties of the actor in the level will be restored after the sequence is closed.

Parameters

- **actor** (*ActorUnreal or ActorBlender*) – The actor to use in the sequence.

- **transform_keys** (*Union[TransformKeys, List[TransformKeys]]*) – The transform keys to use with the actor.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to None. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str]*) – For blender, this argument is the name of an action(bpy.types.Action).
- **Engine** (*For Unreal*) –
- **actor.** (*this argument is the engine path to the animation asset to use for the*) –
- **None** (*Default to None. If*) –
- **used** (*the actor's current animation is*) –
- **used.** (*else the specified animation is*) –

classmethod use_camera(*camera:* CameraBase, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *fov: float | None = None*) → None

Use the specified level camera in the sequence. The location, rotation and fov set in this method are only used in the sequence. The location, rotation and fov of the camera in the level will be restored after the sequence is closed.

Parameters

- **camera** (CameraUnreal *or* CameraBlender) – The camera to use in the sequence.
- **location** (*Optional[Vector], optional*) – The location of the camera. Defaults to None. unit: meter.
- **rotation** (*Optional[Vector], optional*) – The rotation of the camera. Defaults to None. unit: degree.
- **fov** (*float, optional*) – The field of view of the camera. Defaults to None. unit: degree.

classmethod use_camera_with_keys(*camera:* CameraBase, *transform_keys: List[*SequenceTransformKey*]* | SequenceTransformKey, *fov: float | None = None*) → None

Use the specified level camera in the sequence. The transform_keys and fov set in this method are only used in the sequence. The location, rotation and fov of the camera in the level will be restored after the sequence is closed.

Parameters

- **camera** (CameraUnreal *or* CameraBlender) – The camera to use.
- **transform_keys** (*TransformKeys*) – The transform keys to use.
- **fov** (*float, optional*) – The field of view to use. Defaults to None. unit: degree.

SequenceUnreal

Fig. 18: Inheritance diagram

---

class xrfeitoria.sequence.sequence_unreal.**SequenceUnreal**

Sequence class for Unreal.

classmethod add_audio(*audio_asset_path: str*, *start_frame: int | None = None*, *end_frame: int | None = None*) → None

Add an audio track to the sequence.

Parameters

- **audio_asset_path** (*str*) – The path to the audio asset in the engine.
- **start_frame** (*Optional[int], optional*) – The start frame of the audio track. Defaults to None.
- **end_frame** (*Optional[int], optional*) – The end frame of the audio track. Defaults to None.

classmethod add_to_renderer(*output_path: str | Path*, *resolution: Tuple[int, int]*, *render_passes: List[RenderPass]*, *file_name_format: str = '{sequence_name}/{render_pass}/{camera_name}/{frame_number}'*, *console_variables: Dict[str, float] = {'r.MotionBlurQuality': 0}*, *anti_aliasing: AntiAliasSetting | None = None*, *export_vertices: bool = False*, *export_skeleton: bool = False*, *export_audio: bool = False*) → None

Add the sequence to the renderer's job queue. Can only be called after the sequence is instantiated using `new()` or `open()`.

Parameters

- **output_path** (*PathLike*) – The path where the rendered output will be saved.
- **resolution** (*Tuple[int, int]*) – The resolution of the output. (width, height)
- **render_passes** (*List[RenderPass]*) – The list of render passes to be rendered.
- **file_name_format** (*str, optional*) – The format of the output file name. Defaults to `{sequence_name}/{render_pass}/{camera_name}/{frame_number}`.
- **console_variables** (*Dict[str, float], optional*) – The console variables to be set before rendering. Defaults to {'r.MotionBlurQuality': 0}. Ref to *What is stencil_value* for details.
- **anti_aliasing** (*Optional[RenderJobUnreal.AntiAliasSetting], optional*) – The anti-aliasing settings for the render job. Defaults to None.

- **export_vertices** (*bool, optional*) – Whether to export vertices. Defaults to False.
- **export_skeleton** (*bool, optional*) – Whether to export the skeleton. Defaults to False.
- **export_audio** (*bool, optional*) – Whether to export audio. Defaults to False.

Examples

```
>>> import xrfeitoria as xf
>>> from xrfeitoria.data_structure.models import RenderPass
>>> with xf.init_blender() as xf_runner:
...     seq = xf_runner.Sequence.new(seq_name='test'):
...         seq.add_to_renderer(
...             output_path=...,
...             resolution=...,
...             render_passes=[RenderPass('img', 'png')],
...         )
...     xf_runner.render()
```

static check_motion_data(*actor_asset_path: str*, *motion_data: List[Dict[str, Dict[str, float | List[float]]]]*) → List[Dict[str, Dict[str, float | List[float]]]]

Check the motion data for a given actor against the skeleton in the engine. Checks if the bone names in the motion data are a subset of the bone names in the skeleton of the actor. If not, the extra bone names in the motion data are ignored.

Parameters

- **actor_asset_path** (*str*) – The asset path of the actor in the engine.
- **motion_data** (*List[MotionFrame]*) – The motion data to be checked.

Returns

List[MotionFrame] – The checked motion data.

classmethod close() → None

Close the opened sequence.

classmethod get_map_path() → str

Returns the path to the map corresponding to the sequence in the Unreal Engine.

Returns

str – engine path to the map corresponding to the sequence.

classmethod get_playback() → Tuple[int, int]

Get the playback range for the sequence.

Returns

Tuple[int, int] – The start and end frame of the playback range.

classmethod get_seq_path() → str

Returns the path to the sequence in the Unreal Engine.

Returns

str – engine path to the sequence.

classmethod import_actor(*file_path: str | Path*, *actor_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*) → *ActorBase*

Imports an actor from a file and adds it to the sequence.

Parameters

- **file_path** (*PathLike*) – The path to the file containing the actor to import.
- **actor_name** (*Optional[str], optional*) – The name to give the imported actor. If not provided, a name will be generated automatically. Defaults to None.
- **location** (*Vector, optional*) – The initial location of the actor. Defaults to None.
- **rotation** (*Vector, optional*) – The initial rotation of the actor. Defaults to None.
- **scale** (*Vector, optional*) – The initial scale of the actor. Defaults to None.
- **stencil_value** (*int, optional*) – The stencil value to use for the actor. Defaults to 1.

Returns

ActorBase – The imported actor.

classmethod save() → None

Save the sequence.

classmethod set_camera_cut_playback(*start_frame: int | None = None*, *end_frame: int | None = None*) → None

Set the playback range for the sequence.

Parameters

- **start_frame** (*Optional[int]*) – The start frame of the playback range. If not provided, the default start frame will be used.
- **end_frame** (*Optional[int]*) – The end frame of the playback range. If not provided, the default end frame will be used.

Returns

None

classmethod set_playback(*start_frame: int | None = None*, *end_frame: int | None = None*) → None

Set the playback range for the sequence.

Parameters

- **start_frame** (*Optional[int]*) – The start frame of the playback range. If not provided, the default start frame will be used.
- **end_frame** (*Optional[int]*) – The end frame of the playback range. If not provided, the default end frame will be used.

Returns

None

classmethod show() → None

Show the sequence in the engine.

classmethod spawn_actor(*actor_asset_path: str*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *actor_name: str | None = None*, *stencil_value: int = 1*, *anim_asset_path: str | None = None*, *motion_data: List[Dict[str, Dict[str, float | List[float]]]] | None = None*) → *ActorUnreal*

Spawns an actor in the Unreal Engine at the specified location, rotation, and scale.

Parameters

- **cls** – The class object.
- **actor_asset_path** (*str*) – The actor asset path in engine to spawn.
- **location** (*Optional[Vector, optional]*) – The location to spawn the actor at. unit: meter.
- **rotation** (*Optional[Vector, optional]*) – The rotation to spawn the actor with. unit: degree.
- **scale** (*Optional[Vector], optional*) – The scale to spawn the actor with. Defaults to None.
- **actor_name** (*Optional[str], optional*) – The name to give the spawned actor. Defaults to None.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to 1. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str], optional*) – The engine path to the animation asset of the actor. Defaults to None.
- **motion_data** (*Optional[List[MotionFrame]]*) – The motion data used for FK animation.

Returns

ActorUnreal – The spawned actor object.

classmethod spawn_actor_with_keys(*actor_asset_path: str*, *transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *actor_name: str | None = None*, *stencil_value: int = 1*, *anim_asset_path: str | None = None*, *motion_data: List[Dict[str, Dict[str, float | List[float]]]] | None = None*) → *ActorUnreal*

Spawns an actor in the Unreal Engine with the given asset path, transform keys, actor name, stencil value, and animation asset path.

Parameters

- **actor_asset_path** (*str*) – The actor asset path in engine to spawn.
- **transform_keys** (*TransformKeys*) – The transform keys of the actor.
- **actor_name** (*Optional[str], optional*) – The name of the actor. Defaults to None.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to 1. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str], optional*) – The engine path to the animation asset of the actor. Defaults to None.
- **motion_data** (*Optional[List[MotionFrame]]*) – The motion data used for FK animation.

Returns

ActorUnreal – The spawned actor.

classmethod spawn_camera(*location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *fov: float = 90.0*, *camera_name: str | None = None*) → *CameraBase*

Spawn a new camera in the sequence.

Parameters

- **location** (*Vector*) – Location of the camera.
- **rotation** (*Vector*) – Rotation of the camera.
- **fov** (*float in (0.0, 180.0), optional*) – Field of view of the camera len. Defaults to 90.0. (unit: degrees)
- **camera_name** (*str, optional*) – Name of the camera. Defaults to None.

classmethod spawn_camera_with_keys(*transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *fov: float = 90.0*, *camera_name: str | None = None*) → *CameraBase*

Spawn a new camera with keyframes in the sequence.

Parameters

- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **fov** (*float in (0.0, 180.0), optional*) – Field of view of the camera len. Defaults to 90.0. (unit: degrees)
- **camera_name** (*str, optional*) – Name of the camera. Defaults to 'Camera'.

classmethod spawn_shape(*type: Literal['plane', 'cube', 'sphere', 'cylinder', 'cone']*, *shape_name: str | None = None*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int = 1*, ***kwargs*) → *ActorBase*

Spawn a shape in the sequence.

Parameters

- **type** (*str*) – Type of new spawn shape. One of ["plane", "cube", "sphere", "cylinder", "cone"]
- **shape_name** (*str*) – Name of the new added shape. Defaults to None.
- **location** (*Vector, optional*) – Location of the shape. Defaults to (0, 0, 0).
- **rotation** (*Vector, optional*) – Rotation of the shape. Defaults to (0, 0, 0).
- **scale** (*Vector, optional*) – Scale of the shape. Defaults to (1, 1, 1).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the shape. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ShapeBlender or ShapeUnreal – New added shape.

classmethod spawn_shape_with_keys(*transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *type: Literal['plane', 'cube', 'sphere', 'cylinder', 'cone']*, *shape_name: str | None = None*, *stencil_value: int = 1*, ***kwargs*) → *ActorBase*

Spawn a shape with keyframes in the sequence.

Parameters

- **shape_name** (*str*) – Name of the new added shape.

- **type** (*str*) – Type of new spawn shape. One of ["plane", "cube", "sphere", "cylinder", "cone"]
- **transform_keys** (*TransformKeys*) – Keyframes of transform (location, rotation, and scale).
- **stencil_value** (*int in [0, 255], optional*) – Stencil value of the cone. Defaults to 1. Ref to *What is stencil_value* for details.

Returns

ShapeBlender or ShapeUnreal – New added shape.

classmethod use_actor(*actor: ActorBase*, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *scale: Tuple[float, float, float] | None = None*, *stencil_value: int | None = None*, *anim_asset_path: str | None = None*) → None

Use the specified level actor in the sequence. The location, rotation, scale, stencil_value and anim_asset set in this method are only used in the sequence. These peoperties of the actor in the level will be restored after the sequence is closed.

Parameters

- **actor** (*ActorUnreal or ActorBlender*) – The actor to add to the sequence.
- **location** (*Optional[Vector]*) – The initial location of the actor. If None, the actor's current location is used. unit: meter.
- **rotation** (*Optional[Vector]*) – The initial rotation of the actor. If None, the actor's current rotation is used. unit: degree.
- **scale** (*Optional[Vector]*) – The initial scale of the actor. If None, the actor's current scale is used.
- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to None. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str]*) – For blender, this argument is the name of an action(bpy.types.Action).
- **Engine** (*For Unreal*) –
- **actor.** (*this argument is the engine path to the animation asset to use for the*) –
- **None** (*Default to None. If*) –
- **used** (*the actor's current animation is*) –
- **used.** (*else the specified animation is*) –

classmethod use_actor_with_keys(*actor: ActorBase*, *transform_keys: List[SequenceTransformKey] | SequenceTransformKey*, *stencil_value: int | None = None*, *anim_asset_path: str | None = None*) → None

Use the specified level actor in the sequence. The transform_keys, stencil_value and anim_asset set in this method are only used in the sequence. These peoperties of the actor in the level will be restored after the sequence is closed.

Parameters

- **actor** (*ActorUnreal or ActorBlender*) – The actor to use in the sequence.
- **transform_keys** (*Union[TransformKeys, List[TransformKeys]]*) – The transform keys to use with the actor.

- **stencil_value** (*int in [0, 255], optional*) – The stencil value to use for the spawned actor. Defaults to None. Ref to *What is stencil_value* for details.
- **anim_asset_path** (*Optional[str]*) – For blender, this argument is the name of an action(bpy.types.Action).
- **Engine** (*For Unreal*) –
- **actor.** (*this argument is the engine path to the animation asset to use for the*) –
- **None** (*Default to None. If*) –
- **used** (*the actor's current animation is*) –
- **used.** (*else the specified animation is*) –

classmethod use_camera(*camera:* CameraBase, *location: Tuple[float, float, float] | None = None*, *rotation: Tuple[float, float, float] | None = None*, *fov: float | None = None*) → None

Use the specified level camera in the sequence. The location, rotation and fov set in this method are only used in the sequence. The location, rotation and fov of the camera in the level will be restored after the sequence is closed.

Parameters

- **camera** (CameraUnreal *or* CameraBlender) – The camera to use in the sequence.
- **location** (*Optional[Vector], optional*) – The location of the camera. Defaults to None. unit: meter.
- **rotation** (*Optional[Vector], optional*) – The rotation of the camera. Defaults to None. unit: degree.
- **fov** (*float, optional*) – The field of view of the camera. Defaults to None. unit: degree.

classmethod use_camera_with_keys(*camera:* CameraBase, *transform_keys:* *List[*SequenceTransformKey*]* | SequenceTransformKey, *fov: float | None = None*) → None

Use the specified level camera in the sequence. The transform_keys and fov set in this method are only used in the sequence. The location, rotation and fov of the camera in the level will be restored after the sequence is closed.

Parameters

- **camera** (CameraUnreal *or* CameraBlender) – The camera to use.
- **transform_keys** (*TransformKeys*) – The transform keys to use.
- **fov** (*float, optional*) – The field of view to use. Defaults to None. unit: degree.

xrfeitoria.sequence.sequence_wrapper	Sequence wrapper functions.

sequence_wrapper

Sequence wrapper functions.

xrfeitoria.sequence.sequence_wrapper.**sequence_wrapper_blender**(*seq_name: str*, *level: str = 'XRFeitoria'*, *seq_fps: int = 30*, *seq_length: int = 1*, *replace: bool = False*) → *SequenceBlender* | ContextManager[*SequenceBlender*]

Create a new sequence and close the sequence after exiting it.

Parameters

- **seq_name** (*str*) – The name of the sequence.
- **level** (*str, optional*) – The level to associate the sequence with. Defaults to 'XRFeitoria'.
- **seq_fps** (*int, optional*) – The frames per second of the sequence. Defaults to 30.
- **seq_length** (*int, optional*) – The length of the sequence. Defaults to 1.
- **replace** (*bool, optional*) – Whether to replace an existing sequence with the same name. Defaults to False.

Returns

SequenceBlender – The created SequenceBlender object.

xrfeitoria.sequence.sequence_wrapper.**sequence_wrapper_unreal**(*seq_name: str*, *seq_dir: str | None = None*, *level: str | None = None*, *seq_fps: int = 30*, *seq_length: int = 1*, *replace: bool = False*) → *SequenceUnreal* | ContextManager[*SequenceUnreal*]

Create a new sequence, open it in editor, and close the sequence after exiting it.

Parameters

- **seq_name** (*str*) – The name of the sequence.
- **seq_dir** (*Optional[str], optional*) – The directory where the sequence is located. Defaults to None. Falls back to the default sequence path (/Game/XRFeitoriaUnreal/Sequences).
- **level** (*Optional[str], optional*) – The level to associate the sequence with. Defaults to None.
- **seq_fps** (*int, optional*) – The frames per second of the sequence. Defaults to 30.
- **seq_length** (*int, optional*) – The length of the sequence in frames. Defaults to 1.
- **replace** (*bool, optional*) – Whether to replace an existing sequence with the same name. Defaults to False.

Returns

SequenceUnreal – The created SequenceUnreal object.

2.1.9 xrfeitoria.renderer

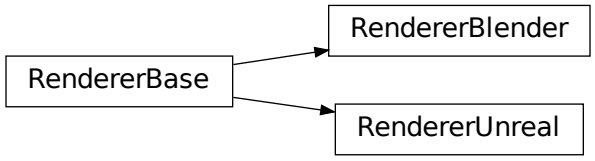


Fig. 19: Inheritance diagram

xrfeitoria.renderer.renderer_base.RendererBase()	Base renderer class.
xrfeitoria.renderer.renderer_blender.RendererBlender()	Renderer class for Blender.
xrfeitoria.renderer.renderer_unreal.RendererUnreal()	Renderer class for Unreal.

RendererBase

class xrfeitoria.renderer.renderer_base.**RendererBase**

Base renderer class.

abstract classmethod add_job()

Add a rendering job to the renderer queue.

This method should be implemented in subclasses, e.g., *RenderBlender* and *RenderUnreal*.

classmethod clear()

Clear all rendering jobs in the renderer queue.

abstract classmethod render_jobs()

Render all jobs in the renderer queue.

This method should be implemented in subclasses, e.g., *RenderBlender* and *RenderUnreal*.

RendererBlender

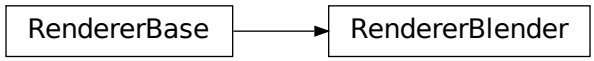


Fig. 20: Inheritance diagram

---

class xrfeitoria.renderer.renderer_blender.**RendererBlender**

Renderer class for Blender.

classmethod add_job(*sequence_name: str*, *output_path: str | Path*, *resolution: Tuple[int, int]*, *render_passes: List[RenderPass]*, *render_engine:* RenderEngineEnumBlender | *Literal['cycles', 'eevee', 'workbench'] = 'cycles'*, *render_samples: int = 128*, *transparent_background: bool = False*, *arrange_file_structure: bool = True*) → None

Add a rendering job with specific settings.

Parameters

- **output_path** (*PathLike*) – Output path of the rendered images.
- **resolution** (*Tuple[int, int]*) – Resolution of the rendered images.
- **render_passes** (*List[RenderPass]*) – Render passes.
- **scene_name** (*str, optional*) – Name of the scene be rendered. Defaults to 'XRFeitoria'.
- **render_engine** (*Union[RenderEngineEnumBlender, Literal['cycles', 'eevee', 'workbench']], optional*) – Render engine. Defaults to cycles.
- **render_samples** (*int, optional*) – Render samples. Defaults to 128.
- **transparent_background** (*bool, optional*) – Transparent background. Defaults to False.
- **arrange_file_structure** (*bool, optional*) – Arrange output images to every camera's folder. Defaults to True.

classmethod clear()

Clear all rendering jobs in the renderer queue.

classmethod render_jobs(*use_gpu: bool = True*) → None

Render all jobs in the render queue, and this method will clear the render queue after rendering.

Parameters

use_gpu (*bool, optional*) – Use GPU to render. Defaults to True.

RendererUnreal

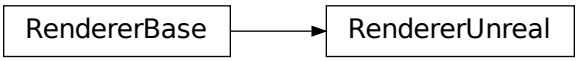


Fig. 21: Inheritance diagram

---

class xrfeitoria.renderer.renderer_unreal.**RendererUnreal**

Renderer class for Unreal.

classmethod add_job(*map_path: str*, *sequence_path: str*, *output_path: str | Path*, *resolution: Tuple[int, int]*, *render_passes: List[RenderPass]*, *file_name_format: str = '{sequence_name}/{render_pass}/{camera_name}/{frame_number}'*, *console_variables: Dict[str, float] = {'r.MotionBlurQuality': 0}*, *anti_aliasing: AntiAliasSetting | None = None*, *export_audio: bool = False*) → None

Add a rendering job to the renderer queue.

Parameters

- **map_path** (*str*) – Path to the map file.
- **sequence_path** (*str*) – Path to the sequence file.
- **output_path** (*PathLike*) – Path to the output folder.
- **resolution** (*Tuple[int, int]*) – Resolution of the output image.
- **render_passes** (*List[RenderPass]*) – Render passes to render.
- **file_name_format** (*str, optional*) – File name format of the output image. Defaults to `{sequence_name}/{render_pass}/{camera_name}/{frame_number}`.
- **console_variables** (*Dict[str, float], optional*) – Console variables to set. Defaults to `{'r.MotionBlurQuality': 0}`. Ref to *What is console_variables* for details.
- **anti_aliasing** (*Optional[RenderJobUnreal.AntiAliasSetting], optional*) – Anti aliasing setting. Defaults to None.
- **export_audio** (*bool, optional*) – Whether to export audio. Defaults to False.

Note: The motion blur is turned off by default. If you want to turn it on, please set `r.MotionBlurQuality` to a non-zero value in `console_variables`.

---

classmethod clear()

Clear all rendering jobs in the renderer queue.

classmethod render_jobs() → None

Render all jobs in the renderer queue.

This method starts the rendering process by setting up a socket connection with the Unreal Engine, rendering the all the render jobs, and then waiting for the engine to send a message indicating that the rendering is complete. If the engine crashes during the rendering process, an error message is logged and the program exits with an error code.

After the rendering is complete, the renderer will perform post-processing in the output_path, including converting camera parameters, vertices, and actor_infos.

Also, this method will clear the render queue after rendering.

classmethod save_queue(*path: str | Path*) → None

Saves the current render queue to a manifest file at the specified path.

Parameters

path (*PathLike*) – The path to save the manifest file to.

2.1.10 xrfeitoria.utils

Remote Functions

xrfeitoria.utils.functions.blender_functions	Remote functions for blender.
xrfeitoria.utils.functions.unreal_functions	Remote functions for unreal.

blender_functions

Remote functions for blender.

xrfeitoria.utils.functions.blender_functions.**apply_motion_data_to_actor**(*motion_data: List[Dict[str, Dict[str, float | List[float]]]]*, *actor_name: str*, *is_first_frame_as_origin: bool = True*) → None

Applies motion data to a given actor in Blender.

Parameters

- **motion_data** (*List[MotionFrame]*) – A list of dictionaries containing motion data for the actor.
- **actor_name** (*str*) – The name of the actor to apply the motion data to.
- **is_first_frame_as_origin** (*bool, optional*) – Whether to set the first frame as the origin. Defaults to True.

xrfeitoria.utils.functions.blender_functions.**apply_shape_keys_to_mesh**(*shape_keys: List[Dict[str, float]]*, *mesh_name: str*) → None

Apply shape keys to the given mesh.

Parameters

- **shape_keys** (*List[Dict[str, float]]*) – A list of dictionaries representing the shape keys and their values.
- **mesh_name** (*str*) – Name of the mesh.

xrfeitoria.utils.functions.blender_functions.**check_sequence**(*seq_name: str*) → bool

Check whether the sequence exists.

Parameters
seq_name (*str*) – Name of the sequence.

Returns
bool – True if the sequence exists.

xrfeitoria.utils.functions.blender_functions.**cleanup_unused**()

Cleanup all the unused data in the current blend file.

xrfeitoria.utils.functions.blender_functions.**enable_gpu**(*gpu_num: int = 1*)

Enable GPU in blender.

Parameters
gpu_num (*int, optional*) – Number of GPUs to use. Defaults to 1.

xrfeitoria.utils.functions.blender_functions.**export_vertices**(*export_path: str | Path*, *use_animation: bool = True*) → None

Export vertices in the world space of all objects in the active scene to npz file, with animation (export N frame) if use_animation is True. export_path will be an npz file with a single key 'verts' which is a numpy array in shape of (N, V, 3), where N is the number of frames, V is the number of vertices.

Parameters

- **export_path** (*PathLike*) – Path to export vertices. (Require a folder)
- **use_animation** (*bool, optional*) – Export with animation. Defaults to True.

xrfeitoria.utils.functions.blender_functions.**get_all_object_in_current_level**(*obj_type: Literal['MESH', 'ARMATURE', 'CAMERA']*) → List[str]

Get all the objects in the current level.

Parameters
obj_type (*Literal["MESH", "ARMATURE", "CAMERA"]*) – Object type.

Returns
List[str] – List of object names.

xrfeitoria.utils.functions.blender_functions.**get_all_object_in_seq**(*seq_name: str*, *obj_type: Literal['MESH', 'ARMATURE', 'CAMERA']*) → List[str]

Get all the objects in the given sequence.

Parameters

- **seq_name** (*str*) – Name of the sequence.

- **obj_type** (*Literal["MESH", "ARMATURE", "CAMERA"]*) – Object type.

Returns
List[str] – List of object names.

xrfeitoria.utils.functions.blender_functions.**get_frame_range**() → Tuple[int, int]

Get the frame range of the active scene.

Returns
Tuple[int, int] – Start frame and End frame.

xrfeitoria.utils.functions.blender_functions.**get_keys_range**() → Tuple[int, int]

Get the max keyframe range of all the objects in the active scene.

Returns
Tuple[int, int] – Start frame and End frame.

xrfeitoria.utils.functions.blender_functions.**get_rotation_to_look_at**(*location: Tuple[float, float, float]*, *target: Tuple[float, float, float]*) → Tuple[float, float, float]

Get the rotation of an object to look at another object.

Parameters

- **location** (*Vector*) – Location of the object.
- **target** (*Vector*) – Location of the target object.

Returns
Vector – Rotation of the object.

xrfeitoria.utils.functions.blender_functions.**import_file**(*file_path: str | Path*) → None

Import file to blender. This function will not return an instance of the imported actor. The file type is determined by the file extension. Supported file types: fbx, obj, abc, ply, stl.

Note: For fbx file, only support binary format. ASCII format is not supported. Ref: https://docs.blender.org/manual/en/3.6/addons/import_export/scene_fbx.html#id4

---

xrfeitoria.utils.functions.blender_functions.**init_scene_and_collection**(*name: str*, *cleanup: bool = False*) → None

Init the default scene and default collection.

Parameters

- **name** (*str*) – Name of the default scene and default collection.
- **cleanup** (*bool, optional*) – Clean up the all the scenes, collections and objects. Defaults to False.

xrfeitoria.utils.functions.blender_functions.**install_plugin**(*plugin_path: str | Path*, *plugin_name_blender: str | None = None*)

Install plugin in blender.

Parameters
path (*PathLike*) – Path to the plugin.

xrfeitoria.utils.functions.blender_functions.**is_background_mode**(*warning: bool = False*) → bool

Check whether Blender is running in background mode.

Returns
bool – True if Blender is running in background mode.

xrfeitoria.utils.functions.blender_functions.**new_level**(*name: str*) → None

Create a new level.

Parameters
name (*str*) – Name of the level.

xrfeitoria.utils.functions.blender_functions.**render_viewport**(*animation: bool = False*) → None

Render the current viewport. Disable multiview when rendering, and restore it after rendering.

Parameters
animation (*bool, optional*) – Render animation. Defaults to False.

xrfeitoria.utils.functions.blender_functions.**save_blend**(*save_path: str | Path = None*, *pack: bool = False*)

Save the current blend file to the given path. If no path is given, save to the current blend file path.

Parameters

- **save_path** (*PathLike, optional*) – Path to save the blend file. Defaults to None.
- **pack** (*bool, optional*) – Automatically pack all external data into .blend file. Defaults to False.

Raises
Exception – Failed to set autopack.

xrfeitoria.utils.functions.blender_functions.**set_active_level**(*level_name: str*)

Sets the active level in XRFeitoria Blender Factory.

Parameters
level_name (*str*) – The name of the level to set as active. (e.g. 'Scene')

Example

```
>>> import xrfeitoria as xf
>>> xf_runner = xf.init_blender()
>>> xf_runner.utils.set_active_level('Scene')  # Return to default level defined by␣
↪blender
```

xrfeitoria.utils.functions.blender_functions.**set_env_color**(*color: Tuple[float, float, float, float]*, *intensity: float = 1.0*)

Set the color of the environment light.

Parameters

- **color** (*Tuple[float, float, float, float]*) – RGBA color of the environment light.
- **intensity** (*float, optional*) – Intensity of the environment light. Defaults to 1.0.

xrfeitoria.utils.functions.blender_functions.**set_frame_current**(*frame: int*) → None

Set current frame of the active scene.

Parameters
frame (*int*) – Frame number.

xrfeitoria.utils.functions.blender_functions.**set_frame_range**(*start: int*, *end: int*) → None

Set frame range of the active scene.

Parameters

- **start** (*int*) – Start frame.
- **end** (*int*) – End frame.

xrfeitoria.utils.functions.blender_functions.**set_hdr_map**(*hdr_map_path: str | Path*) → None

Set HDR map of the active scene.

Parameters
hdr_map_path (*'PathLike'*) – Path of the HDR map file.

unreal_functions

Remote functions for unreal.

xrfeitoria.utils.functions.unreal_functions.**check_asset_in_engine**(*path: str*, *raise_error: bool = False*) → bool

Check if an asset exists in the engine.

Parameters

- **path** (*str*) – asset path in unreal, e.g. "/Game/Resources/Mat0"
- **raise_error** (*bool*) – raise error if the asset does not exist

Returns
bool – True if the asset exists, False otherwise

Raises
ValueError – if the asset does not exist and raise_error is True

xrfeitoria.utils.functions.unreal_functions.**delete_asset**(*asset_path: str*) → None

Delete asset.

Parameters
asset_path (*str*) – asset path in unreal, e.g. "/Game/Resources/Mat0"

xrfeitoria.utils.functions.unreal_functions.**duplicate_asset**(*src_path: str*, *dst_path: str*, *replace: bool = False*) → None

Duplicate asset in unreal.

Parameters

- **src_path** (*str*) – source asset path in unreal, e.g. "/Game/Resources/Mat0"
- **dst_path** (*str*) – destination asset path in unreal, e.g. "/Game/Resources/Mat1"

xrfeitoria.utils.functions.unreal_functions.**get_mask_color**(*stencil_value: int*) → Tuple[int, int, int]

Retrieves the RGB color value associated with the given stencil value.

Parameters
stencil_value (*int*) – The stencil value for which to retrieve the color.

Returns
Tuple[int, int, int] – The RGB color value associated with the stencil value.

xrfeitoria.utils.functions.unreal_functions.**get_mask_color_file**() → str

Returns the path of the mask color file.

Returns
str – The path of the mask color file.

xrfeitoria.utils.functions.unreal_functions.**get_rotation_to_look_at**(*location: Tuple[float, float, float]*, *target: Tuple[float, float, float]*) → Tuple[float, float, float]

Get the rotation of an object to look at another object.

Parameters

- **location** (*Vector*) – Location of the object.
- **target** (*Vector*) – Location of the target object.

Returns
Vector – Rotation of the object.

xrfeitoria.utils.functions.unreal_functions.**get_skeleton_names**(*actor_asset_path: str*) → List[str]

Retrieves the names of the bones in the skeleton of a SkeletalMeshActor (also can be child class of it).

Parameters
actor_asset_path (*str*) – The asset path of the SkeletalMeshActor.

Returns
List[str] – The names of the bones in the skeleton.

xrfeitoria.utils.functions.unreal_functions.**import_anim**(*path: str*, *skeleton_path: str*, *dest_path: str | None = None*, *replace: bool = True*) → List[str]

Import animation to the default asset path.

Parameters

- **path** (*str*) – The file path to import, e.g. "D:/assets/SMPL_XL-Animation.fbx".
- **skeleton_path** (*str*) – The path to the skeleton, e.g. "/Game/XRFeitoriaUnreal/Assets/SMPL_XL_Skeleton".
- **dest_path** (*str, optional*) – The destination directory in the engine. Defaults to None, falls back to {skeleton_path.parent}/Animation.
- **replace** (*bool, optional*) – whether to replace the existing asset. Defaults to True.

Returns
List[str] – A list of paths to the imported animations, e.g. ["/Game/XRFeitoriaUnreal/Assets/SMPL_XL-Animation"].

xrfeitoria.utils.functions.unreal_functions.**import_asset**(*path: str | List[str]*, *dst_dir_in_engine: str | None = None*, *replace: bool = True*, *with_parent_dir: bool = True*) → str | List[str]

Import assets to the default asset path.

Parameters

- **path** (*Union[str, List[str]]*) – a file path or a list of file paths to import, e.g. "D:/assets/SMPL_XL.fbx"

- **dst_dir_in_engine** (*Optional[str], optional*) – destination directory in the engine. Defaults to None falls back to '/Game/XRFeitoriaUnreal/Assets'
- **replace** (*bool, optional*) – whether to replace the existing asset. Defaults to True.
- **with_parent_dir** (*bool, optional*) – whether to create a parent directory that contains the imported asset. If False, the imported asset will be in *dst_dir_in_engine* directly. Defaults to True.

Returns

Union[str, List[str]] – a path or a list of paths to the imported assets, e.g. "/Game/XRFeitoriaUnreal/Assets/SMPL_XL"

xrfeitoria.utils.functions.unreal_functions.**new_seq_data**(*asset_path: str*, *sequence_path: str*, *map_path: str*) → None

Create a new data asset of sequence data.

Parameters

- **asset_path** (*str*) – path of the data asset.
- **sequence_path** (*str*) – path of the sequence asset.
- **map_path** (*str*) – path of the map asset.

Returns

unreal.DataAsset – the created data asset.

Notes

SequenceData Properties:

- "SequencePath": str
- "MapPath": str

xrfeitoria.utils.functions.unreal_functions.**open_asset**(*asset_path: str*) → None

Open asset.

Parameters

asset_path (*str*) – asset path in unreal, e.g. "/Game/Resources/Mat0"

xrfeitoria.utils.functions.unreal_functions.**open_level**(*asset_path: str*) → None

Open a level.

Parameters

asset_path (*str*) – asset path in unreal, e.g. "/Game/Maps/Map0"

xrfeitoria.utils.functions.unreal_functions.**save_current_level**(*asset_path: str | None = None*) → None

Save the current opened level.

Animation utils

xrfeitoria.utils.anim.motion	Motion data structure and related functions.
xrfeitoria.utils.anim.utils	Utilities for animation data loading and dumping.

motion

Motion data structure and related functions.

class xrfeitoria.utils.anim.motion.**Motion**(*transl: ndarray*, *body_poses: ndarray*, *n_frames: int | None = None*, *fps: float = 30.0*)

Wrap motion data. Provide methods to get transform info for 3D calculations.

The motion data will be used along with *Skeleton* instance in retargeting, and the local spaces of bones are all defined in such skeletons.

__init__(*transl: ndarray*, *body_poses: ndarray*, *n_frames: int | None = None*, *fps: float = 30.0*) → None

Transl & body_poses are in the space of corresponding *Skeleton* instance.

convert_fps(*fps: float*)

Converts the frames per second (fps) of the animation to the specified value.

Parameters
fps (*float*) – The desired frames per second.

Raises

- **NotImplementedError** – If the fps is greater than the current fps.
- **NotImplementedError** – If the fps is less than the current fps when undividable.

copy() → *Motion*

Return a copy of the motion instance.

cut_motion(*start_frame: int | None = None*, *end_frame: int | None = None*)

Cut the motion sequence to a given number of frames (to [start_frame, end_frame])

Parameters

- **start_frame** (*Optional[int], optional*) – The start frame to cut to. Defaults to None.
- **end_frame** (*Optional[int], optional*) – The end frame to cut to. Defaults to None.

Raises

- **AssertionError** – If the start frame is less than 0.
- **AssertionError** – If the end frame is greater than the number of frames in the motion sequence.
- **AssertionError** – If the start frame is greater than or equal to the end frame.

cut_transl()

Cut the transl to zero.

This will make the animation stay in place, like root motion.

get_bone_matrix_basis(*bone_name: str*, *frame=0*) → ndarray

pose2rest: relative to the bone space at rest pose.

Parameters

- **bone_name** (*str*) – bone name
- **frame** (*int, optional*) – frame index. Defaults to 0.

Returns

np.ndarray – transform matrix like [[R, T], [0, 1]]

get_motion_data() → List[Dict[str, Dict[str, float | List[float]]]]

Returns a list of dictionaries containing *rotation* and *location* for each bone of each frame in the animation.

Each dictionary contains bone names as keys and a nested dictionary as values. The nested dictionary contains 'rotation' and 'location' keys, which correspond to the rotation and location of the bone in that frame.

Returns

List[MotionFrame] – A list of dictionaries containing motion data for each frame of the animation.

insert_rest_pose()

Insert rest pose to the first frame.

sample_motion(*n_frames: int*)

Randomly sample motions, picking n_frames from the original motion sequence. The indices are totally random using *np.random.choice*.

Parameters

n_frames (*int*) – The number of frames to sample. Randomly sampled from the original motion sequence.

Raises

AssertionError – If the number of frames to sample is less than or equal to 0.

slice_motion(*frame_interval: int*)

Slice the motion sequence by a given frame interval.

Parameters

frame_interval (*int*) – The frame interval to use for slicing the motion sequence.

Raises

TypeError – If the frame interval is not an integer.

BONE_NAMES: List[str]

BONE_NAME_TO_IDX: Dict[str, int]

PARENTS: List[int]

class xrfeitoria.utils.anim.motion.**SMPLMotion**(*transl: ndarray*, *body_poses: ndarray*, *n_frames: int | None = None*, *fps: float = 30.0*)

__init__(*transl: ndarray*, *body_poses: ndarray*, *n_frames: int | None = None*, *fps: float = 30.0*) → None

Transl & body_poses are in the space of corresponding *Skeleton* instance.

copy() → *SMPLMotion*

Return a copy of the motion instance.

dump_humandata(*filepath: str | Path*, *betas: ndarray*, *meta: Dict[str, Any] | None = None*, *global_orient_offset: ndarray = array([0., 0., 0.])*, *transl_offset: ndarray = array([0., 0., 0.])*, *root_location_t0: ndarray | None = None*, *pelvis_location_t0: ndarray | None = None*) → None

Dump the motion data to a humandata file at the given *filepath*.

Parameters

- **filepath** (*PathLike*) – The filepath to dump the motion data to.
- **betas** (*np.ndarray*) – The betas array.
- **meta** (*Optional[Dict[str, Any]]*) – Additional metadata. Defaults to None.
- **global_orient_offset** (*np.ndarray*) – The global orientation offset. Defaults to np.zeros(3).
- **transl_offset** (*np.ndarray*) – The translation offset. Defaults to np.zeros(3).
- **root_location_t0** (*Optional[np.ndarray]*) – The root location at time 0. Defaults to None.
- **pelvis_location_t0** (*Optional[np.ndarray]*) – The pelvis location at time 0. Defaults to None.

Note: HumanData is a structure of smpl/smplx data defined in https://github.com/open-mmlab/mmhuman3d/blob/main/docs/human_data.md

The humandata file is a npz file containing the following keys:

```
motion_data = {
    '__data_len__': n_frames,
    'smpl': {
        'betas': betas,  # (1, 10)
        'transl': transl,  # (n_frames, 3)
        'global_orient': global_orient,  # (n_frames, 3)
        'body_pose': body_pose,  # (n_frames, 69)
    },
    'meta': {'gender': 'neutral'},  # optional
}
```

---

classmethod from_amass_data(*amass_data*, *insert_rest_pose: bool*) → *SMPLMotion*

Create a Motion instance from AMASS data (SMPL)

Parameters

- **amass_data** (*dict*) – A dictionary containing the AMASS data.
- **insert_rest_pose** (*bool*) – Whether to insert a rest pose at the beginning of the motion.

Returns

SMPLMotion – A SMPLMotion instance containing the AMASS data.

classmethod from_smpl_data(*smpl_data: ~typing.Dict[str, ~numpy.ndarray]*, *fps: float = 30.0*, *insert_rest_pose: bool = False*, *global_orient_adj: ~scipy.spatial.transform._rotation.Rotation | None = <scipy.spatial.transform._rotation.Rotation object>*, *vector_convertor: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = <bound method ConverterMotion.vec_humandata2smplx of <class 'xrfeitoria.utils.converter.ConverterMotion'>>*) → *SMPLMotion*

Create SMPLMotion instance from smpl_data.

smpl_data should be a dict like object, with required keys: ['betas', 'body_pose', 'global_orient'] and optional key: ['transl']

Parameters

- **smpl_data** – dict with require keys ["body_pose", "global_orient"] and optional key ["transl"]
- **insert_rest_pose** (*bool*) – whether to insert a rest pose at the 0th-frame.

Returns

SMPLMotion – An instance of SMPLMotion containing the smpl_data.

```
BONE_NAMES: List[str] = ['pelvis', 'left_hip', 'right_hip', 'spine1', 'left_knee',
'right_knee', 'spine2', 'left_ankle', 'right_ankle', 'spine3', 'left_foot',
'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder',
'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw']
```

```
BONE_NAME_TO_IDX: Dict[str, int] = {'head': 15, 'jaw': 22, 'left_ankle': 7,
'left_collar': 13, 'left_elbow': 18, 'left_foot': 10, 'left_hip': 1, 'left_knee': 4,
'left_shoulder': 16, 'left_wrist': 20, 'neck': 12, 'pelvis': 0, 'right_ankle': 8,
'right_collar': 14, 'right_elbow': 19, 'right_foot': 11, 'right_hip': 2,
'right_knee': 5, 'right_shoulder': 17, 'right_wrist': 21, 'spine1': 3, 'spine2': 6,
'spine3': 9}
```

```
GLOBAL_ORIENT_ADJUSTMENT = <scipy.spatial.transform._rotation.Rotation object>
```

```
NAMES = ['Pelvis', 'Hip_L', 'Hip_R', 'Spine1', 'Knee_L', 'Knee_R', 'Spine2',
'Ankle_L', 'Ankle_R', 'Chest', 'Toes_L', 'Toes_R', 'Neck', 'Scapula_L', 'Scapula_R',
'Head', 'Shoulder_L', 'Shoulder_R', 'Elbow_L', 'Elbow_R', 'Wrist_L', 'Wrist_R']
```

```
NAME_TO_SMPL_IDX = {'Ankle_L': 7, 'Ankle_R': 8, 'Chest': 9, 'Elbow_L': 18,
'Elbow_R': 19, 'Head': 15, 'Hip_L': 1, 'Hip_R': 2, 'Knee_L': 4, 'Knee_R': 5, 'Neck':
12, 'Pelvis': 0, 'Scapula_L': 13, 'Scapula_R': 14, 'Shoulder_L': 16, 'Shoulder_R':
17, 'Spine1': 3, 'Spine2': 6, 'Toes_L': 10, 'Toes_R': 11, 'Wrist_L': 20, 'Wrist_R':
21}
```

```
PARENTS: List[int] = [-1, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 12, 12, 13, 14,
16, 17, 18, 19, 20, 21]
```

```
SMPL_IDX_TO_NAME: Dict[int, str] = {0: 'Pelvis', 1: 'Hip_L', 2: 'Hip_R', 3:
'Spine1', 4: 'Knee_L', 5: 'Knee_R', 6: 'Spine2', 7: 'Ankle_L', 8: 'Ankle_R', 9:
'Chest', 10: 'Toes_L', 11: 'Toes_R', 12: 'Neck', 13: 'Scapula_L', 14: 'Scapula_R',
15: 'Head', 16: 'Shoulder_L', 17: 'Shoulder_R', 18: 'Elbow_L', 19: 'Elbow_R', 20:
'Wrist_L', 21: 'Wrist_R', 22: '', 23: ''}
```

class xrfeitoria.utils.anim.motion.**SMPLXMotion**(*transl: ndarray*, *body_poses: ndarray*, *n_frames: int | None = None*, *fps: float = 30.0*)

__init__(*transl: ndarray*, *body_poses: ndarray*, *n_frames: int | None = None*, *fps: float = 30.0*) → None

Transl & body_poses are in the space of corresponding *Skeleton* instance.

copy() → *SMPLXMotion*

Return a copy of the motion instance.

dump_humandata(*filepath: str | Path*, *betas: ndarray*, *meta: Dict[str, Any] | None = None*, *global_orient_offset: ndarray = array([0., 0., 0.])*, *transl_offset: ndarray = array([0., 0., 0.])*, *root_location_t0: ndarray | None = None*, *pelvis_location_t0: ndarray | None = None*) → None

Dump the motion data to a humandata file at the given *filepath*.

Parameters

- **filepath** (*PathLike*) – The filepath to dump the motion data to.
- **betas** (*np.ndarray*) – The betas array.
- **meta** (*Optional[Dict[str, Any]]*) – Additional metadata. Defaults to None.
- **global_orient_offset** (*np.ndarray*) – The global orientation offset. Defaults to np.zeros(3).
- **transl_offset** (*np.ndarray*) – The translation offset. Defaults to np.zeros(3).
- **root_location_t0** (*Optional[np.ndarray]*) – The root location at time 0. Defaults to None.
- **pelvis_location_t0** (*Optional[np.ndarray]*) – The pelvis location at time 0. Defaults to None.

Note: HumanData is a structure of smpl/smplx data defined in https://github.com/open-mmlab/mmhuman3d/blob/main/docs/human_data.md

The humandata file is a npz file containing the following keys:

```
humandata = {
    '__data_len__': n_frames,
    'smplx': {
        'betas': betas,  # (1, 10)
        'transl': transl,  # (n_frames, 3)
        'global_orient': global_orient,  # (n_frames, 3)
        'body_pose': body_pose,  # (n_frames, 63)
        'jaw_pose': jaw_pose,  # (n_frames, 3)
        'leye_pose': leye_pose,  # (n_frames, 3)
        'reye_pose': reye_pose,  # (n_frames, 3)
        'left_hand_pose': left_hand_pose,  # (n_frames, 45)
        'right_hand_pose': right_hand_pose,  # (n_frames, 45)
        'expression': expression,  # (n_frames, 10)
    },
    'meta': {'gender': 'neutral'},  # optional
}
```

---

classmethod from_amass_data(*amass_data*, *insert_rest_pose: bool*, *flat_hand_mean: bool = True*) → *SMPLXMotion*

Create a Motion instance from AMASS data (SMPLX)

Parameters

- **amass_data** (*dict*) – A dictionary containing the AMASS data.
- **insert_rest_pose** (*bool*) – Whether to insert a rest pose at the beginning of the motion.
- **flat_hand_mean** (*bool*) – Whether to use the flat hand mean pose.

Returns
SMPLXMotion – A SMPLXMotion instance containing the AMASS data.

Raises
AssertionError – If the surface model type in the AMASS data is not 'smplx'.

classmethod from_smplx_data(*smplx_data: ~typing.Dict[str, ~numpy.ndarray], fps: float = 30.0, insert_rest_pose: bool = False, flat_hand_mean: bool = False, global_orient_adj: ~scipy.spatial.transform._rotation.Rotation | None = <scipy.spatial.transform._rotation.Rotation object>, vector_convertor: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = <bound method ConverterMotion.vec_humandata2smplx of <class 'xrfeitoria.utils.converter.ConverterMotion'>>*) → *SMPLXMotion*

Create SMPLXMotion instance from smplx_data.

smplx_data should be a dict like object, with required keys: ['betas', "body_pose", "global_orient"] and optional key: ['transl', 'jaw_pose', 'leye_pose', 'reye_pose', 'left_hand_pose', 'right_hand_pose', 'expression']

Parameters

- **smplx_data** – require keys ["body_pose", "global_orient"] and optional key ["transl"]
- **fps** (*float*) – the motion's FPS. Defaults to 30.0.
- **insert_rest_pose** (*bool*) – whether to insert a rest pose at the 0th-frame. Defaults to False.
- **flat_hand_mean** (*bool*) – whether the hands with zero rotations are flat hands. Defaults to False.
- **global_orient_adj** (*spRotation, None*) –
- **vector_convertor** – a function applies to smplx_data's translation.

Returns
SMPLXMotion – An instance of SMPLXMotion containing the smplx_data.

```
BONE_NAMES: List[str] = ['pelvis', 'left_hip', 'right_hip', 'spine1', 'left_knee',
'right_knee', 'spine2', 'left_ankle', 'right_ankle', 'spine3', 'left_foot',
'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder',
'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw',
'left_eye_smplhf', 'right_eye_smplhf', 'left_index1', 'left_index2', 'left_index3',
'left_middle1', 'left_middle2', 'left_middle3', 'left_pinky1', 'left_pinky2',
'left_pinky3', 'left_ring1', 'left_ring2', 'left_ring3', 'left_thumb1',
'left_thumb2', 'left_thumb3', 'right_index1', 'right_index2', 'right_index3',
'right_middle1', 'right_middle2', 'right_middle3', 'right_pinky1', 'right_pinky2',
'right_pinky3', 'right_ring1', 'right_ring2', 'right_ring3', 'right_thumb1',
'right_thumb2', 'right_thumb3']
```

```
BONE_NAME_TO_IDX: Dict[str, int] = {'head': 15, 'jaw': 22, 'left_ankle': 7,
'left_collar': 13, 'left_elbow': 18, 'left_eye_smplhf': 23, 'left_foot': 10,
'left_hip': 1, 'left_index1': 25, 'left_index2': 26, 'left_index3': 27, 'left_knee':
4, 'left_middle1': 28, 'left_middle2': 29, 'left_middle3': 30, 'left_pinky1': 31,
'left_pinky2': 32, 'left_pinky3': 33, 'left_ring1': 34, 'left_ring2': 35,
'left_ring3': 36, 'left_shoulder': 16, 'left_thumb1': 37, 'left_thumb2': 38,
'left_thumb3': 39, 'left_wrist': 20, 'neck': 12, 'pelvis': 0, 'right_ankle': 8,
'right_collar': 14, 'right_elbow': 19, 'right_eye_smplhf': 24, 'right_foot': 11,
'right_hip': 2, 'right_index1': 40, 'right_index2': 41, 'right_index3': 42,
'right_knee': 5, 'right_middle1': 43, 'right_middle2': 44, 'right_middle3': 45,
'right_pinky1': 46, 'right_pinky2': 47, 'right_pinky3': 48, 'right_ring1': 49,
'right_ring2': 50, 'right_ring3': 51, 'right_shoulder': 17, 'right_thumb1': 52,
'right_thumb2': 53, 'right_thumb3': 54, 'right_wrist': 21, 'spine1': 3, 'spine2': 6,
'spine3': 9}

GLOBAL_ORIENT_ADJUSTMENT = <scipy.spatial.transform._rotation.Rotation object>

NAMES = ['Pelvis', 'Hip_L', 'Hip_R', 'Spine1', 'Knee_L', 'Knee_R', 'Spine2',
'Ankle_L', 'Ankle_R', 'Chest', 'Toes_L', 'Toes_R', 'Neck', 'Scapula_L', 'Scapula_R',
'Head', 'Shoulder_L', 'Shoulder_R', 'Elbow_L', 'Elbow_R', 'Wrist_L', 'Wrist_R',
'index_A_L', 'index_B_L', 'index_C_L', 'middle_A_L', 'middle_B_L', 'middle_C_L',
'pinky_A_L', 'pinky_B_L', 'pinky_C_L', 'ring_A_L', 'ring_B_L', 'ring_C_L',
'thumb_A_L', 'thumb_B_L', 'thumb_C_L', 'index_A_R', 'index_B_R', 'index_C_R',
'middle_A_R', 'middle_B_R', 'middle_C_R', 'pinky_A_R', 'pinky_B_R', 'pinky_C_R',
'ring_A_R', 'ring_B_R', 'ring_C_R', 'thumb_A_R', 'thumb_B_R', 'thumb_C_R']

NAME_TO_SMPL_IDX = {'Ankle_L': 7, 'Ankle_R': 8, 'Chest': 9, 'Elbow_L': 18,
'Elbow_R': 19, 'Head': 15, 'Hip_L': 1, 'Hip_R': 2, 'Knee_L': 4, 'Knee_R': 5, 'Neck':
12, 'Pelvis': 0, 'Scapula_L': 13, 'Scapula_R': 14, 'Shoulder_L': 16, 'Shoulder_R':
17, 'Spine1': 3, 'Spine2': 6, 'Toes_L': 10, 'Toes_R': 11, 'Wrist_L': 20, 'Wrist_R':
21, 'index_A_L': 22, 'index_A_R': 37, 'index_B_L': 23, 'index_B_R': 38, 'index_C_L':
24, 'index_C_R': 39, 'middle_A_L': 25, 'middle_A_R': 40, 'middle_B_L': 26,
'middle_B_R': 41, 'middle_C_L': 27, 'middle_C_R': 42, 'pinky_A_L': 28, 'pinky_A_R':
43, 'pinky_B_L': 29, 'pinky_B_R': 44, 'pinky_C_L': 30, 'pinky_C_R': 45, 'ring_A_L':
31, 'ring_A_R': 46, 'ring_B_L': 32, 'ring_B_R': 47, 'ring_C_L': 33, 'ring_C_R': 48,
'thumb_A_L': 34, 'thumb_A_R': 49, 'thumb_B_L': 35, 'thumb_B_R': 50, 'thumb_C_L': 36,
'thumb_C_R': 51}

PARENTS: List[int] = [-1, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 12, 12, 13, 14,
16, 17, 18, 19, 20, 21, 20, 22, 23, 20, 25, 26, 20, 28, 29, 20, 31, 32, 20, 34, 35,
21, 37, 38, 21, 40, 41, 21, 43, 44, 21, 46, 47, 21, 49, 50]

SMPLX_IDX_TO_NAME: Dict[int, str] = {0: 'Pelvis', 1: 'Hip_L', 2: 'Hip_R', 3:
'Spine1', 4: 'Knee_L', 5: 'Knee_R', 6: 'Spine2', 7: 'Ankle_L', 8: 'Ankle_R', 9:
'Chest', 10: 'Toes_L', 11: 'Toes_R', 12: 'Neck', 13: 'Scapula_L', 14: 'Scapula_R',
15: 'Head', 16: 'Shoulder_L', 17: 'Shoulder_R', 18: 'Elbow_L', 19: 'Elbow_R', 20:
'Wrist_L', 21: 'Wrist_R', 22: 'index_A_L', 23: 'index_B_L', 24: 'index_C_L', 25:
'middle_A_L', 26: 'middle_B_L', 27: 'middle_C_L', 28: 'pinky_A_L', 29: 'pinky_B_L',
30: 'pinky_C_L', 31: 'ring_A_L', 32: 'ring_B_L', 33: 'ring_C_L', 34: 'thumb_A_L',
35: 'thumb_B_L', 36: 'thumb_C_L', 37: 'index_A_R', 38: 'index_B_R', 39: 'index_C_R',
40: 'middle_A_R', 41: 'middle_B_R', 42: 'middle_C_R', 43: 'pinky_A_R', 44:
'pinky_B_R', 45: 'pinky_C_R', 46: 'ring_A_R', 47: 'ring_B_R', 48: 'ring_C_R', 49:
'thumb_A_R', 50: 'thumb_B_R', 51: 'thumb_C_R'}
```

xrfeitoria.utils.anim.motion.**get_humandata**(*smpl_x_data: Dict[str, ndarray]*, *smpl_x_type: Literal['smpl', 'smplx']*, *betas: ndarray*, *meta: Dict[str, Any] | None = None*, *global_orient_offset: ndarray = array([0., 0., 0.])*, *transl_offset: ndarray = array([0., 0., 0.])*, *root_location_t0: ndarray | None = None*, *pelvis_location_t0: ndarray | None = None*) → Dict[str, Any]

Get human data for a given set of parameters.

Parameters

- **smpl_x_data** (*Dict[str, np.ndarray]*) – Dictionary containing the SMPL-X data.
- **smpl_x_type** (*Literal['smpl', 'smplx']*) – Type of SMPL-X model.
- **betas** (*np.ndarray*) – Array of shape (n, 10) representing the shape parameters.
- **meta** (*Optional[Dict[str, Any]], optional*) – Additional metadata. Defaults to None.
- **global_orient_offset** (*np.ndarray*) – Array of shape (n, 3) representing the global orientation offset.
- **transl_offset** (*np.ndarray*) – Array of shape (3,) representing the translation offset.
- **root_location_t0** (*Optional[np.ndarray], optional*) – Array of shape (3,) representing the root location at time t=0. Defaults to None.
- **pelvis_location_t0** (*Optional[np.ndarray], optional*) – Array of shape (3,) representing the pelvis location at time t=0. Defaults to None.

Returns

dict – Dictionary containing the human data.

utils

Utilities for animation data loading and dumping.

xrfeitoria.utils.anim.utils.**dump_humandata**(*motion: SMPLMotion | SMPLXMotion*, *save_filepath: str | Path*, *meta_filepath: str | Path*, *actor_name: str | None = None*) → None

Dump human data to a file. This function must be associate with a meta file provided by SMPL-XL.

Parameters

- **motion** (*Union[SMPLMotion, SMPLXMotion]*) – The motion data to be dumped.
- **save_filepath** (*PathLike*) – The file path to save the dumped data.
- **meta_filepath** (*PathLike*) – The file path to the meta information, storing the parameters of the SMPL-XL model.
- **actor_name** (*Optional[str], optional*) – The name of the actor. Defaults to None.

Note: HumanData is a structure of smpl/smplx data defined in https://github.com/open-mmlab/mmhuman3d/blob/main/docs/human_data.md

The humandata file is a npz file containing the following keys:

```
humandata = {
    '__data_len__': n_frames,
    'smplx': {
        'betas': betas,  # (1, 10)
        'transl': transl,  # (n_frames, 3)
        'global_orient': global_orient,  # (n_frames, 3)
        'body_pose': body_pose,  # (n_frames, 63)
        'jaw_pose': jaw_pose,  # (n_frames, 3)
        'leye_pose': leye_pose,  # (n_frames, 3)
        'reye_pose': reye_pose,  # (n_frames, 3)
        'left_hand_pose': left_hand_pose,  # (n_frames, 45)
        'right_hand_pose': right_hand_pose,  # (n_frames, 45)
        'expression': expression,  # (n_frames, 10)
    },
    'meta': {'gender': 'neutral', 'actor_name': '(XF)actor-001'},  # optional
}
```

---

xrfeitoria.utils.anim.utils.**load_amass_motion**(*input_amass_smpl_x_path: str | Path*, *is_smplx: bool = True*) → *SMPLMotion* | *SMPLXMotion*

Load AMASS SMPLX motion data. Only for SMPLX motion for now.

Parameters

input_amass_smpl_x_path (*PathLike*) – Path to AMASS SMPL/SMPLX motion data.

Returns

Union[SMPLMotion, SMPLXMotion] – Motion data, which consists of data read from AMASS file.

xrfeitoria.utils.anim.utils.**load_humandata_motion**(*input_humandata_path: str | Path*) → *SMPLMotion* | *SMPLXMotion*

Load humandata SMPL / SMPLX motion data.

HumanData is a structure of smpl/smplx data defined in https://github.com/open-mmlab/mmhuman3d/blob/main/docs/human_data.md

Parameters

input_humandata_path (*PathLike*) – Path to humandata SMPL / SMPLX motion data.

Returns

Union[SMPLMotion, SMPLXMotion] – Motion data, which consists of data read from humandata file.

xrfeitoria.utils.anim.utils.**refine_smpl_x**(*smpl_x_filepath: Path*, *meta_filepath: Path*, *replace_smpl_x_file: bool = False*, *offset_location: ndarray = array([0., 0., 0.])*, *offset_rotation: ndarray = array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])*) → None

Refine translation and rotation of SMPL-X parameters.

xrfeitoria.utils.anim.utils.**refine_smpl_x_from_actor_info**(*smpl_x_filepath: Path*, *meta_filepath: Path*, *actor_info_file: Path*, *replace_smpl_x_file: bool = False*)

Refine translation and rotation of SMPL-X parameters from actor info file.

RPC runner

xrfeitoria.utils.runner	Runner for starting blender or unreal as a rpc server.

runner

Runner for starting blender or unreal as a rpc server.

class xrfeitoria.utils.runner.**BlenderRPCRunner**(*new_process: bool = False*, *engine_exec: str | Path | None = None*, *project_path: str | Path = ''*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *background: bool = True*)

static get_pid() → int

Get blender process id.

get_src_plugin_path() → Path

Get plugin source zip path.

static test_connection(*debug: bool = False*) → bool

Test connection.

class xrfeitoria.utils.runner.**RPCRunner**(*new_process: bool = False*, *engine_exec: str | Path | None = None*, *project_path: str | Path = ''*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *background: bool = True*)

A wrapper for starting blender as a rpc server.

__init__(*new_process: bool = False*, *engine_exec: str | Path | None = None*, *project_path: str | Path = ''*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *background: bool = True*) → None

Initialize RPCRunner.

Parameters

- **new_process** (*bool, optional*) – whether to start a new process. Defaults to False.
- **engine_exec** (*Optional[PathLike], optional*) – path to engine executable. Defaults to None.
- **project_path** (*PathLike, optional*) – path to project. Defaults to ‘’.
- **reload_rpc_code** (*bool, optional*) – whether to reload the registered rpc functions and classes. If you are developing the package or writing a custom remote function, set this to True to reload the code. This will only be in effect when *new_process=False* if the engine process is reused. Defaults to False.
- **replace_plugin** (*bool, optional*) – whether to replace the plugin installed for the engine. Defaults to False.
- **dev_plugin** (*bool, optional*) – Whether to use the plugin under local directory. If False, would use the plugin downloaded from a remote server. Defaults to False.
- **background** (*bool, optional*) – whether to run the engine in background without GUI. Defaults to True.

`check_engine_alive()` → None

Check if the engine process is alive. This function should be called in a separate thread.

Raises
`RuntimeError` – if engine process is not alive.

`check_engine_alive_psutil()` → None

Check if the engine process is alive using psutil. This function should be called in a separate thread.

Raises
`RuntimeError` – if engine process is not alive.

`abstract static get_pid()` → int

`get_process_output(process: Popen)` → str

Get process output when process is exited with non-zero code.

`abstract get_src_plugin_path()` → Path

Get plugin source path, including downloading or symlinking/copying from local directory.

priority:
if *self.dev_plugin=False*: download from url > build from source

if *self.dev_plugin=True*: build from source

Returns
Path – plugin source path

`reuse()` → bool

Try to reuse existing engine process.

Returns
bool – whether the engine process is reused.

Raises
`RuntimeError` – if *new_process=True* but an existing engine process is found.

`start()` → None

Start rpc server.

`stop()` → None

Stop rpc server.

`abstract static test_connection(debug: bool = False)` → None

`wait_for_start(process: Popen)` → None

Wait 3 minutes for RPC server to start.

After 3 minutes, ask user if they want to quit if it takes too long.

Parameters
`process` (*subprocess.Popen*) – process to wait for.

`property dst_plugin_dir: Path`

Get plugin directory to install.

`property installed_plugin_version: str`

Get plugin version installed.

`property plugin_info: PluginInfo`

property plugin_url: str | None

property port: int

Get RPC port depending on engine type.

class xrfeitoria.utils.runner.**UnrealRPCRunner**(*new_process: bool = False*, *engine_exec: str | Path | None = None*, *project_path: str | Path = ''*, *reload_rpc_code: bool = False*, *replace_plugin: bool = False*, *dev_plugin: bool = False*, *background: bool = True*)

UnrealRPCRunner.

static get_pid() → int

Get unreal process id.

get_src_plugin_path() → Path

Get plugin source directory.

static test_connection(*debug: bool = False*) → None

Test connection.

xrfeitoria.utils.runner.**plugin_info_type**

alias of PluginInfo

Tools

xrfeitoria.utils.tools	Utils tools for logging and progress bar.
xrfeitoria.utils.viewer	Utils for loading images and annotations.
xrfeitoria.utils.projector	Project 3D points to 2D points, and draw 3D points on image, using XRPrimer camera.
xrfeitoria.utils.validations	Validations for the arguments.

tools

Utils tools for logging and progress bar.

xrfeitoria.utils.tools.**setup_logger**(*level: Literal['RPC', 'TRACE', 'DEBUG', 'INFO', 'SUCCESS', 'WARNING', 'ERROR', 'CRITICAL'] = 'INFO'*, *log_path: PathLike | None = None*, *replace: bool = True*) → loguru.Logger

Setup logging to file and console.

Parameters

- **level** (*Literal['TRACE', 'DEBUG', 'INFO', 'SUCCESS', 'WARNING', 'ERROR', 'CRITICAL']*, *optional*) – logging level. Defaults to 'INFO', find more in https://loguru.readthedocs.io/en/stable/api/logger.html. The order of the levels is:
  - 'RPC' (custom level): logging RPC messages which are sent by RPC protocols.
  - 'TRACE': logging engine output like console output of blender.
  - 'DEBUG': logging debug messages.
  - 'INFO': logging info messages.
  - ...

- **log_path** (*Path, optional*) – path to save the log file. Defaults to None.
- **replace** (*bool, optional*) – replace the log file if exists. Defaults to True.

viewer

Utils for loading images and annotations.

class xrfeitoria.utils.viewer.**Viewer**(*sequence_dir: str | Path*)

Utils for loading images and annotations.

Examples

```
>>> viewer = Viewer('/path/to/sequence')
>>> camera_name = 'cam'
>>> frame = 0
>>> img = viewer.get_img(camera_name=camera_name, frame=frame)
>>> mask = viewer.get_mask(camera_name=camera_name, frame=frame)
>>> depth = viewer.get_depth(camera_name=camera_name, frame=frame)
>>> flow = viewer.get_flow(camera_name=camera_name, frame=frame)
>>> normal = viewer.get_normal(camera_name=camera_name, frame=frame)
>>> diffuse = viewer.get_diffuse(camera_name=camera_name, frame=frame)
```

__init__(*sequence_dir: str | Path*) → None

Initialize with the sequence directory.

Parameters

sequence_dir (*PathLike*) – path to the sequence directory

get_depth(*camera_name: str*, *frame: int*, *inverse: bool = False*, *depth_rescale: float = 1.0*) → ndarray

Get depth of the given frame ('depth/{frame:04d}.*')

Parameters

- **camera_name** (*str*) – the camera name
- **frame** (*int*) – the frame number
- **inverse** (*bool, optional*) – whether to inverse the depth. If True, white (255) represents the farthest, and black (0) represents the nearest. if False, white (255) represents the nearest, and black (0) represents the farthest. Defaults to False.
- **depth_rescale** (*float, optional*) – scaling the depth to map it into (0, 255). `depth = depth / depth_rescale`. Depth values greater than *depth_rescale* will be clipped. Defaults to 1.0.

Returns

np.ndarray – depth of shape (H, W, 3)

get_diffuse(*camera_name: str*, *frame: int*) → ndarray

Get diffuse image of the given frame ('diffuse/{frame:04d}.*')

Parameters

- **camera_name** (*str*) – the camera name
- **frame** (*int*) – the frame number

Returns
np.ndarray – image of shape (H, W, 3)

`get_flow`(*camera_name: str*, *frame: int*) → ndarray
Get optical flow of the given frame ('flow/{frame:04d}.*')

Parameters

- **camera_name** (*str*) – the camera name
- **frame** (*int*) – the frame number

Returns
np.ndarray – optical flow of shape (H, W, 3)

`get_img`(*camera_name: str*, *frame: int*) → ndarray
Get rgb image of the given frame ('img/{frame:04d}.*')

Parameters

- **camera_name** (*str*) – the camera name
- **frame** (*int*) – the frame number

Returns
np.ndarray – image of shape (H, W, 3)

`get_mask`(*camera_name: str*, *frame: int*) → ndarray
Get mask of the given frame ('mask/{frame:04d}.*')

Parameters

- **camera_name** (*str*) – the camera name
- **frame** (*int*) – the frame number

Returns
np.ndarray – image of shape (H, W, 3)

`get_normal`(*camera_name: str*, *frame: int*) → ndarray
Get normal map of the given frame ('normal/{frame:04d}.*')

Parameters

- **camera_name** (*str*) – the camera name
- **frame** (*int*) – the frame number

Returns
np.ndarray – normal map of shape (H, W, 3)

`DEPTH = 'depth'`

`DIFFUSE = 'diffuse'`

`FLOW = 'flow'`

`IMG = 'img'`

`MASK = 'mask'`

`NORMAL = 'normal'`

`property camera_names: List[str]`

`property frame_num: int`

projector

Project 3D points to 2D points, and draw 3D points on image, using XRPrimer camera.

xrfeitoria.utils.projector.**draw_points3d**(*points3d: ndarray*, *camera_param:* CameraParameter, *image: ndarray | None = None*, *color: Tuple[int, int, int] = (255, 0, 0)*) → ndarray

Draw 3d points on canvas. The 3d points will be projected to 2d points first. Then draw the 2d points on a canvas of the same size as the image. If image is not None, the canvas will be drawn on the image. Otherwise, the canvas will be returned which is a binary image, where 1 means the points are drawn.

Parameters

- **point3d** (*np.ndarray*) – [N, 3] points to project, where N is the number of points, and 3 is the location of each point. In convention of opencv.
- **camera_param** (*PinholeCameraParameter*) – camera parameter
- **image** (*Optional[np.ndarray], optional*) – [height, width, channel]. Defaults to None. If not None, the canvas will be drawn on the image with the given color.
- **color** (*Tuple[int, int, int], optional*) – color of the points. Defaults to (255, 0, 0).

Returns

np.ndarray – If image is None, return a binary image [H, W], where 1 means the points are drawn, dtype=np.bool. Otherwise, return a image [H, W, C] with the points drawn on it, dtype=np.uint8.

xrfeitoria.utils.projector.**points2d_to_canvas**(*points2d: ndarray*, *resolution: Tuple[int, int]*) → ndarray

Draw 2d points on canvas. The canvas is a binary image, where 1 means the points are drawn.

Parameters

- **points2d** (*np.ndarray*) – [N, 2] points to draw
- **resolution** (*Tuple[int, int]*) – [height, width] of the canvas

Returns

np.ndarray – [height, width] binary image, where 1 means the points are drawn, dtype=np.bool

xrfeitoria.utils.projector.**project_points3d**(*points3d: ndarray*, *camera_param:* CameraParameter) → ndarray

Project 3D point to 2D point.

Parameters

- **point3d** (*np.ndarray*) – [N, 3] points to project, where N is the number of points, and 3 is the location of each point. In convention of 'opencv'.
- **camera_param** (*PinholeCameraParameter*) – camera parameter in convention of xrprimer

Returns

np.ndarray – [N, 2] projected 2d points, dtype=np.float32

validations

Validations for the arguments.

class xrfeitoria.utils.validations.**Validator**

classmethod validate_argument_type(*value*, *typelist: List[Type]*) → None

Validate the type of an argument.

Parameters

- **value** (*Any*) – The value to be validated.
- **typelist** (*List[Type]*) – The list of types to be validated.

Raises

TypeError – If the type of the argument is not in the typelist.

classmethod validate_vector(*value*, *length: int*)

Validate the type and length of a vector.

Parameters

- **value** (*Any*) – The value to be validated.
- **length** (*int*) – The length of the vector.

Raises

TypeError – If the type of the argument is not a vector, or the length of the vector is not equal to the given length.

2.1.11 xrfeitoria.data_structure

Models

> https://docs.pydantic.dev/latest/usage/models/

xrfeitoria.data_structure.models.RenderPass	Render pass model contains render layer and image format.
xrfeitoria.data_structure.models.RenderJobBlender	Render job model for Blender.
xrfeitoria.data_structure.models.RenderJobUnreal	Render job model for Unreal.
xrfeitoria.data_structure.models.SequenceTransformKey	Transform key model for object (actor/camera) in sequence.

RenderPass

pydantic model xrfeitoria.data_structure.models.**RenderPass**

Render pass model contains render layer and image format.

Supported render layer and image format:

RenderLayerBlender

RenderLayerUnreal

ImageFormat

RenderOutputEnumBlender:

- img
- mask
- depth
- flow
- normal
- diffuse
- denoising_depth

RenderOutputEnumUnreal:

- img
- mask
- depth
- flow
- normal
- diffuse
- metallic
- roughness
- specular
- tangent
- basecolor

ImageFileFormatEnum:

- png
- bmp
- jpg
- exr

Used in:

- *RenderJobBlender*
- *RenderJobUnreal*
- *Renderer.add_job*
- *Sequence.add_to_renderer*
- ...

Examples

define

RenderJobBlender

RenderJobUnreal

seq.add_to_renderer

```python
from xrfeitoria.data_structure.models import RenderPass
RenderPass('img', 'png')
RenderPass('mask', 'exr')
RenderPass('normal', 'jpg')
...
```

```python
from xrfeitoria.data_structure.models import RenderJobBlender, RenderPass
RenderJobBlender(
    sequence_name=...,
    output_path=...,
    resolution=...,
    render_passes=[RenderPass('img', 'png')],
)
```

```python
from xrfeitoria.data_structure.models import RenderJobUnreal, RenderPass
RenderJobUnreal(
    map_path=...,
    sequence_path=...,
    output_path=...,
    resolution=...,
    render_passes=[RenderPass('img', 'png')],
)
```

```python
import xrfeitoria as xf
from xrfeitoria.data_structure.models import RenderPass

with xf.init_blender() as xf_runner:
    seq = xf_runner.Sequence.new(seq_name='test'):
        seq.add_to_renderer(
            output_path=...,
            resolution=...,
            render_passes=[RenderPass('img', 'png')],
        )

    xf_runner.render()
```

```
{
   "title": "RenderPass",
   "description": "Render pass model contains render layer and image format.\n\
↪nSupported render layer and image format:\n\n    .. tabs::\n        .. tab::␣
↪RenderLayerBlender\n\n            :class:`~xrfeitoria.data_structure.constants.
↪RenderOutputEnumBlender`:\n\n                - img\n                - mask\n    ␣
↪          - depth\n                - flow\n                - normal\n          ␣
↪   - diffuse\n                - denoising_depth\n\n        .. tab::␣
```

(continues on next page)

(continued from previous page)

```
↪RenderLayerUnreal\n\n            :class:`~xrfeitoria.data_structure.constants.
↪RenderOutputEnumUnreal`:\n\n                - img\n                - mask\n      ␣
↪         - depth\n                - flow\n                - normal\n            ␣
↪  - diffuse\n                - metallic\n                - roughness\n           ␣
↪    - specular\n                - tangent\n                - basecolor\n\n       ␣
↪.. tab:: ImageFormat\n\n            :class:`~xrfeitoria.data_structure.constants.
↪ImageFileFormatEnum`:\n\n                - png\n                - bmp\n          ␣
↪     - jpg\n                - exr\n\nUsed in:\n    - :class:`~xrfeitoria.data_
↪structure.models.RenderJobBlender`\n    - :class:`~xrfeitoria.data_structure.
↪models.RenderJobUnreal`\n    - :meth:`Renderer.add_job <xrfeitoria.renderer.
↪renderer_base.RendererBase.add_job>`\n    - :meth:`Sequence.add_to_renderer
↪<xrfeitoria.sequence.sequence_base.SequenceBase.add_to_renderer>`\n    - ...\n\
↪nExamples:\n\n    .. tabs::\n        .. tab:: define\n\n            .. code-block:
↪: python\n                :linenos:\n\n                from xrfeitoria.data_
↪structure.models import RenderPass\n                RenderPass('img', 'png')\n   ␣
↪            RenderPass('mask', 'exr')\n                RenderPass('normal', 'jpg
↪')\n                ...\n\n        .. tab:: RenderJobBlender\n\n            ..␣
↪code-block:: python\n                :linenos:\n                :emphasize-lines:␣
↪6\n\n                from xrfeitoria.data_structure.models import␣
↪RenderJobBlender, RenderPass\n                RenderJobBlender(\n               ␣
↪   sequence_name=...,\n                    output_path=...,\n                   ␣
↪resolution=...,\n                    render_passes=[RenderPass('img', 'png')],\n ␣
↪              )\n\n        .. tab:: RenderJobUnreal\n\n            .. code-block::
↪ python\n                :linenos:\n                :emphasize-lines: 7\n\n      ␣
↪         from xrfeitoria.data_structure.models import RenderJobUnreal, RenderPass\
↪n                RenderJobUnreal(\n                    map_path=...,\n          ␣
↪        sequence_path=...,\n                    output_path=...,\n               ␣
↪    resolution=...,\n                    render_passes=[RenderPass('img', 'png')],
↪\n                )\n\n        .. tab:: seq.add_to_renderer\n\n            ..␣
↪code-block:: python\n                :linenos:\n                :emphasize-lines:␣
↪9\n\n                import xrfeitoria as xf\n                from xrfeitoria.
↪data_structure.models import RenderPass\n\n                with xf.init_blender()␣
↪as xf_runner:\n                    seq = xf_runner.Sequence.new(seq_name='test'):\
↪n                        seq.add_to_renderer(\n                            output_
↪path=...,\n                            resolution=...,\n                       ␣
↪  render_passes=[RenderPass('img', 'png')],\n                        )\n\n      ␣
↪            xf_runner.render()",
  "type": "object",
  "properties": {
     "render_layer": {
        "anyOf": [
           {
              "$ref": "#/$defs/RenderOutputEnumBlender"
           },
           {
              "$ref": "#/$defs/RenderOutputEnumUnreal"
           }
        ],
        "description": "Render layer of the render pass.",
        "title": "Render Layer"
     },
     "image_format": {
```

(continues on next page)

(continued from previous page)


```
        "allOf": [
          {
            "$ref": "#/$defs/ImageFileFormatEnum"
          }
        ],
        "description": "Image format of the render pass."
      }
    },
    "$defs": {
      "ImageFileFormatEnum": {
        "description": "Image file format enum.",
        "enum": [
          "PNG",
          "BMP",
          "JPEG",
          "JPEG",
          "OPEN_EXR"
        ],
        "title": "ImageFileFormatEnum",
        "type": "string"
      },
      "RenderOutputEnumBlender": {
        "description": "Render layer enum of Blender.",
        "enum": [
          "Image",
          "IndexOB",
          "Depth",
          "Denoising Depth",
          "Vector",
          "Normal",
          "DiffCol",
          "actor_infos.json",
          "camera_params"
        ],
        "title": "RenderOutputEnumBlender",
        "type": "string"
      },
      "RenderOutputEnumUnreal": {
        "description": "Render layer enum of Unreal.",
        "enum": [
          "img",
          "mask",
          "depth",
          "flow",
          "normal",
          "diffuse",
          "metallic",
          "roughness",
          "specular",
          "tangent",
          "basecolor",
          "lineart",
```


(continues on next page)

(continued from previous page)

```
            "vertices",
            "skeleton",
            "actor_infos",
            "camera_params",
            "Audio"
         ],
         "title": "RenderOutputEnumUnreal",
         "type": "string"
      }
   },
   "required": [
      "render_layer",
      "image_format"
   ]
}
```

Fields

- *image_format (xrfeitoria.data_structure.constants.ImageFileFormatEnum)*
- *render_layer (xrfeitoria.data_structure.constants.RenderOutputEnumBlender | xrfeitoria.data_structure.constants.RenderOutputEnumUnreal)*

field image_format: *ImageFileFormatEnum* [Required]

Image format of the render pass.

field render_layer: *RenderOutputEnumBlender* | *RenderOutputEnumUnreal* [Required]

Render layer of the render pass.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

RenderJobBlender

pydantic model xrfeitoria.data_structure.models.**RenderJobBlender**

Render job model for Blender.

```
{
   "title": "RenderJobBlender",
   "description": "Render job model for Blender.",
   "type": "object",
   "properties": {
      "sequence_name": {
         "description": "Name of the sequence of the render job.",
         "title": "Sequence Name",
         "type": "string"
      },
      "output_path": {
         "anyOf": [
            {
```

(continues on next page)

(continued from previous page)


```
                "type": "string"
            },
            {
                "format": "path",
                "type": "string"
            }
        ],
        "description": "Output path of the render job.",
        "title": "Output Path"
    },
    "resolution": {
        "description": "Resolution of the images rendered by the render job.",
        "maxItems": 2,
        "minItems": 2,
        "prefixItems": [
            {
                "type": "integer"
            },
            {
                "type": "integer"
            }
        ],
        "title": "Resolution",
        "type": "array"
    },
    "render_passes": {
        "description": "Render passes of the render job.",
        "items": {
            "$ref": "#/$defs/RenderPass"
        },
        "title": "Render Passes",
        "type": "array"
    },
    "render_engine": {
        "allOf": [
            {
                "$ref": "#/$defs/RenderEngineEnumBlender"
            }
        ],
        "description": "Render engine of the render job."
    },
    "render_samples": {
        "description": "Render samples of the render job.",
        "minimum": 1,
        "title": "Render Samples",
        "type": "integer"
    },
    "transparent_background": {
        "default": false,
        "description": "Whether to render with transparent background.",
        "title": "Transparent Background",
        "type": "boolean"
```


(continues on next page)

(continued from previous page)

```
      },
      "arrange_file_structure": {
         "default": true,
         "description": "Whether to arrange file structure of the output images of␣
↪the render job.",
         "title": "Arrange File Structure",
         "type": "boolean"
      }
   },
   "$defs": {
      "ImageFileFormatEnum": {
         "description": "Image file format enum.",
         "enum": [
            "PNG",
            "BMP",
            "JPEG",
            "JPEG",
            "OPEN_EXR"
         ],
         "title": "ImageFileFormatEnum",
         "type": "string"
      },
      "RenderEngineEnumBlender": {
         "description": "Render engine enum of Blender.",
         "enum": [
            "CYCLES",
            "BLENDER_EEVEE",
            "BLENDER_WORKBENCH"
         ],
         "title": "RenderEngineEnumBlender",
         "type": "string"
      },
      "RenderOutputEnumBlender": {
         "description": "Render layer enum of Blender.",
         "enum": [
            "Image",
            "IndexOB",
            "Depth",
            "Denoising Depth",
            "Vector",
            "Normal",
            "DiffCol",
            "actor_infos.json",
            "camera_params"
         ],
         "title": "RenderOutputEnumBlender",
         "type": "string"
      },
      "RenderOutputEnumUnreal": {
         "description": "Render layer enum of Unreal.",
         "enum": [
            "img",
```

(continues on next page)

(continued from previous page)

```
            "mask",
            "depth",
            "flow",
            "normal",
            "diffuse",
            "metallic",
            "roughness",
            "specular",
            "tangent",
            "basecolor",
            "lineart",
            "vertices",
            "skeleton",
            "actor_infos",
            "camera_params",
            "Audio"
         ],
         "title": "RenderOutputEnumUnreal",
         "type": "string"
      },
      "RenderPass": {
         "description": "Render pass model contains render layer and image format.\
↪n\nSupported render layer and image format:\n\n    .. tabs::\n        .. tab::␣
↪RenderLayerBlender\n\n            :class:`~xrfeitoria.data_structure.constants.
↪RenderOutputEnumBlender`:\n\n                - img\n                - mask\n     ␣
↪          - depth\n                - flow\n                - normal\n           ␣
↪    - diffuse\n                - denoising_depth\n\n        .. tab::␣
↪RenderLayerUnreal\n\n            :class:`~xrfeitoria.data_structure.constants.
↪RenderOutputEnumUnreal`:\n\n                - img\n                - mask\n      ␣
↪          - depth\n                - flow\n                - normal\n           ␣
↪   - diffuse\n                - metallic\n                - roughness\n           ␣
↪    - specular\n                - tangent\n                - basecolor\n\n       ␣
↪.. tab:: ImageFormat\n\n            :class:`~xrfeitoria.data_structure.constants.
↪ImageFileFormatEnum`:\n\n                - png\n                - bmp\n          ␣
↪     - jpg\n                - exr\n\nUsed in:\n    - :class:`~xrfeitoria.data_
↪structure.models.RenderJobBlender`\n    - :class:`~xrfeitoria.data_structure.
↪models.RenderJobUnreal`\n    - :meth:`Renderer.add_job <xrfeitoria.renderer.
↪renderer_base.RendererBase.add_job>`\n    - :meth:`Sequence.add_to_renderer
↪<xrfeitoria.sequence.sequence_base.SequenceBase.add_to_renderer>`\n    - ...\n\
↪nExamples:\n\n    .. tabs::\n        .. tab:: define\n\n            .. code-block:
↪: python\n                :linenos:\n\n                from xrfeitoria.data_
↪structure.models import RenderPass\n                RenderPass('img', 'png')\n   ␣
↪            RenderPass('mask', 'exr')\n                RenderPass('normal', 'jpg
↪')\n                ...\n\n        .. tab:: RenderJobBlender\n\n            ..␣
↪code-block:: python\n                :linenos:\n                :emphasize-lines:␣
↪6\n\n                from xrfeitoria.data_structure.models import␣
↪RenderJobBlender, RenderPass\n                RenderJobBlender(\n                ␣
↪   sequence_name=...,\n                    output_path=...,\n                ␣
↪resolution=...,\n                    render_passes=[RenderPass('img', 'png')],\n ␣
↪               )\n\n        .. tab:: RenderJobUnreal\n\n            .. code-block::
↪ python\n                :linenos:\n                :emphasize-lines: 7\n\n      ␣
↪         from xrfeitoria.data_structure.models import RenderJobUnreal, RenderPass\
```

(continues on next page)

(continued from previous page)

```
↪n                RenderJobUnreal(\n                    map_path=...,\n          ␣
↪        sequence_path=...,\n                  output_path=...,\n              ␣
↪    resolution=...,\n                  render_passes=[RenderPass('img', 'png')],
↪\n                )\n\n        .. tab:: seq.add_to_renderer\n\n            ..␣
↪code-block:: python\n                :linenos:\n                :emphasize-lines:␣
↪9\n\n                import xrfeitoria as xf\n                from xrfeitoria.
↪data_structure.models import RenderPass\n\n                with xf.init_blender()␣
↪as xf_runner:\n                    seq = xf_runner.Sequence.new(seq_name='test'):\
↪n                        seq.add_to_renderer(\n                            output_
↪path=...,\n                            resolution=...,\n                          ␣
↪  render_passes=[RenderPass('img', 'png')],\n                        )\n\n       ␣
↪            xf_runner.render()",
        "properties": {
           "render_layer": {
              "anyOf": [
                 {
                    "$ref": "#/$defs/RenderOutputEnumBlender"
                 },
                 {
                    "$ref": "#/$defs/RenderOutputEnumUnreal"
                 }
              ],
              "description": "Render layer of the render pass.",
              "title": "Render Layer"
           },
           "image_format": {
              "allOf": [
                 {
                    "$ref": "#/$defs/ImageFileFormatEnum"
                 }
              ],
              "description": "Image format of the render pass."
           }
        },
        "required": [
           "render_layer",
           "image_format"
        ],
        "title": "RenderPass",
        "type": "object"
     }
  },
  "required": [
     "sequence_name",
     "output_path",
     "resolution",
     "render_passes",
     "render_engine",
     "render_samples"
  ]
}
```

Fields

- *arrange_file_structure (bool)*
- *output_path (str | pathlib.Path)*
- *render_engine (xrfeitoria.data_structure.constants.RenderEngineEnumBlender)*
- *render_passes (List[xrfeitoria.data_structure.models.RenderPass])*
- *render_samples (int)*
- *resolution (Tuple[int, int])*
- *sequence_name (str)*
- *transparent_background (bool)*

field arrange_file_structure: bool = True

Whether to arrange file structure of the output images of the render job.

field output_path: str | Path [Required]

Output path of the render job.

field render_engine: *RenderEngineEnumBlender* [Required]

Render engine of the render job.

field render_passes: List[*RenderPass*] [Required]

Render passes of the render job.

field render_samples: int [Required]

Render samples of the render job.

Constraints

- **ge** = 1

field resolution: Tuple[int, int] [Required]

Resolution of the images rendered by the render job.

field sequence_name: str [Required]

Name of the sequence of the render job.

field transparent_background: bool = False

Whether to render with transparent background.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

RenderJobUnreal

pydantic model xrfeitoria.data_structure.models.**RenderJobUnreal**

Render job model for Unreal.


```
{
   "title": "RenderJobUnreal",
   "description": "Render job model for Unreal.",
   "type": "object",
```


(continues on next page)

(continued from previous page)


```
  "properties": {
     "map_path": {
        "description": "Map path of the render job.",
        "title": "Map Path",
        "type": "string"
     },
     "sequence_path": {
        "description": "Sequence path of the render job.",
        "title": "Sequence Path",
        "type": "string"
     },
     "output_path": {
        "anyOf": [
           {
              "type": "string"
           },
           {
              "format": "path",
              "type": "string"
           }
        ],
        "description": "Output path of the render job.",
        "title": "Output Path"
     },
     "resolution": {
        "description": "Resolution of the images rendered by the render job.",
        "maxItems": 2,
        "minItems": 2,
        "prefixItems": [
           {
              "type": "integer"
           },
           {
              "type": "integer"
           }
        ],
        "title": "Resolution",
        "type": "array"
     },
     "render_passes": {
        "description": "Render passes of the render job.",
        "items": {
           "$ref": "#/$defs/RenderPass"
        },
        "title": "Render Passes",
        "type": "array"
     },
     "file_name_format": {
        "default": "{sequence_name}/{render_pass}/{camera_name}/{frame_number}",
        "description": "File name format of the render job.",
        "title": "File Name Format",
        "type": "string"
```


(continues on next page)

(continued from previous page)

```
      },
      "console_variables": {
         "additionalProperties": {
            "type": "number"
         },
         "default": {
            "r.MotionBlurQuality": 0
         },
         "description": "Additional console variables of the render job. Ref to :
↪ref:`FAQ-console-variables` for details.",
         "title": "Console Variables",
         "type": "object"
      },
      "anti_aliasing": {
         "allOf": [
            {
               "$ref": "#/$defs/AntiAliasSetting"
            }
         ],
         "default": {
            "enable": false,
            "override_anti_aliasing": false,
            "spatial_samples": 8,
            "temporal_samples": 8,
            "warmup_frames": 0,
            "render_warmup_frame": false
         },
         "description": "Anti aliasing setting of the render job."
      },
      "export_audio": {
         "default": false,
         "description": "Whether to export audio of the render job.",
         "title": "Export Audio",
         "type": "boolean"
      }
   },
   "$defs": {
      "AntiAliasSetting": {
         "properties": {
            "enable": {
               "default": false,
               "description": "Whether to enable anti aliasing.",
               "title": "Enable",
               "type": "boolean"
            },
            "override_anti_aliasing": {
               "default": false,
               "description": "Whether to override anti aliasing.",
               "title": "Override Anti Aliasing",
               "type": "boolean"
            },
            "spatial_samples": {
```

(continues on next page)

(continued from previous page)

```
                "default": 8,
                "description": "Spatial samples of the anti aliasing.",
                "minimum": 1,
                "title": "Spatial Samples",
                "type": "integer"
            },
            "temporal_samples": {
                "default": 8,
                "description": "Temporal samples of the anti aliasing.",
                "minimum": 1,
                "title": "Temporal Samples",
                "type": "integer"
            },
            "warmup_frames": {
                "default": 0,
                "description": "Warmup frames in engine which would rendered before␣
↪actual frame range. This is crucial when particle system (Niagara) is used.",
                "minimum": 0,
                "title": "Warmup Frames",
                "type": "integer"
            },
            "render_warmup_frame": {
                "default": false,
                "description": "Whether to render warmup frame of the anti aliasing.
↪",
                "title": "Render Warmup Frame",
                "type": "boolean"
            }
        },
        "title": "AntiAliasSetting",
        "type": "object"
    },
    "ImageFileFormatEnum": {
        "description": "Image file format enum.",
        "enum": [
            "PNG",
            "BMP",
            "JPEG",
            "JPEG",
            "OPEN_EXR"
        ],
        "title": "ImageFileFormatEnum",
        "type": "string"
    },
    "RenderOutputEnumBlender": {
        "description": "Render layer enum of Blender.",
        "enum": [
            "Image",
            "IndexOB",
            "Depth",
            "Denoising Depth",
            "Vector",
```

(continues on next page)

(continued from previous page)


```
            "Normal",
            "DiffCol",
            "actor_infos.json",
            "camera_params"
         ],
         "title": "RenderOutputEnumBlender",
         "type": "string"
      },
      "RenderOutputEnumUnreal": {
         "description": "Render layer enum of Unreal.",
         "enum": [
            "img",
            "mask",
            "depth",
            "flow",
            "normal",
            "diffuse",
            "metallic",
            "roughness",
            "specular",
            "tangent",
            "basecolor",
            "lineart",
            "vertices",
            "skeleton",
            "actor_infos",
            "camera_params",
            "Audio"
         ],
         "title": "RenderOutputEnumUnreal",
         "type": "string"
      },
      "RenderPass": {
         "description": "Render pass model contains render layer and image format.\
↪n\nSupported render layer and image format:\n\n    .. tabs::\n        .. tab::␣
↪RenderLayerBlender\n\n            :class:`~xrfeitoria.data_structure.constants.
↪RenderOutputEnumBlender`:\n\n                - img\n                - mask\n    ␣
↪          - depth\n                - flow\n                - normal\n          ␣
↪    - diffuse\n                - denoising_depth\n\n        .. tab::␣
↪RenderLayerUnreal\n\n            :class:`~xrfeitoria.data_structure.constants.
↪RenderOutputEnumUnreal`:\n\n                - img\n                - mask\n      ␣
↪          - depth\n                - flow\n                - normal\n          ␣
↪  - diffuse\n                - metallic\n                - roughness\n          ␣
↪    - specular\n                - tangent\n                - basecolor\n\n       ␣
↪.. tab:: ImageFormat\n\n            :class:`~xrfeitoria.data_structure.constants.
↪ImageFileFormatEnum`:\n\n                - png\n                - bmp\n          ␣
↪     - jpg\n                - exr\n\nUsed in:\n    - :class:`~xrfeitoria.data_
↪structure.models.RenderJobBlender`\n    - :class:`~xrfeitoria.data_structure.
↪models.RenderJobUnreal`\n    - :meth:`Renderer.add_job <xrfeitoria.renderer.
↪renderer_base.RendererBase.add_job>`\n    - :meth:`Sequence.add_to_renderer
↪<xrfeitoria.sequence.sequence_base.SequenceBase.add_to_renderer>`\n    - ...\n\
↪nExamples:\n\n    .. tabs::\n        .. tab:: define\n\n            .. code-block:
```


(continues on next page)

(continued from previous page)

```
↪: python\n                :linenos:\n\n                from xrfeitoria.data_
↪structure.models import RenderPass\n                RenderPass('img', 'png')\n  ␣
↪            RenderPass('mask', 'exr')\n                RenderPass('normal', 'jpg
↪')\n                ...\n\n        .. tab:: RenderJobBlender\n\n            ..␣
↪code-block:: python\n                :linenos:\n                :emphasize-lines:␣
↪6\n\n                from xrfeitoria.data_structure.models import␣
↪RenderJobBlender, RenderPass\n                RenderJobBlender(\n                ␣
↪   sequence_name=...,\n                    output_path=...,\n                    ␣
↪resolution=...,\n                    render_passes=[RenderPass('img', 'png')],\n ␣
↪              )\n\n        .. tab:: RenderJobUnreal\n\n            .. code-block::
↪ python\n                :linenos:\n                :emphasize-lines: 7\n\n      ␣
↪         from xrfeitoria.data_structure.models import RenderJobUnreal, RenderPass\
↪n                RenderJobUnreal(\n                    map_path=...,\n           ␣
↪        sequence_path=...,\n                    output_path=...,\n               ␣
↪    resolution=...,\n                    render_passes=[RenderPass('img', 'png')],
↪\n                )\n\n        .. tab:: seq.add_to_renderer\n\n            ..␣
↪code-block:: python\n                :linenos:\n                :emphasize-lines:␣
↪9\n\n                import xrfeitoria as xf\n                from xrfeitoria.
↪data_structure.models import RenderPass\n\n                with xf.init_blender()␣
↪as xf_runner:\n                    seq = xf_runner.Sequence.new(seq_name='test'):\
↪n                    seq.add_to_renderer(\n                                output_
↪path=...,\n                                resolution=...,\n                    ␣
↪  render_passes=[RenderPass('img', 'png')],\n                    )\n\n       ␣
↪            xf_runner.render()",
        "properties": {
           "render_layer": {
              "anyOf": [
                 {
                    "$ref": "#/$defs/RenderOutputEnumBlender"
                 },
                 {
                    "$ref": "#/$defs/RenderOutputEnumUnreal"
                 }
              ],
              "description": "Render layer of the render pass.",
              "title": "Render Layer"
           },
           "image_format": {
              "allOf": [
                 {
                    "$ref": "#/$defs/ImageFileFormatEnum"
                 }
              ],
              "description": "Image format of the render pass."
           }
        },
        "required": [
           "render_layer",
           "image_format"
        ],
        "title": "RenderPass",
        "type": "object"
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "required": [
        "map_path",
        "sequence_path",
        "output_path",
        "resolution",
        "render_passes"
    ]
}
```

Fields

- *anti_aliasing (xrfeitoria.data_structure.models.RenderJobUnreal.AntiAliasSetting)*
- *console_variables (Dict[str, float])*
- *export_audio (bool)*
- *file_name_format (str)*
- *map_path (str)*
- *output_path (str | pathlib.Path)*
- *render_passes (List[xrfeitoria.data_structure.models.RenderPass])*
- *resolution (Tuple[int, int])*
- *sequence_path (str)*

field anti_aliasing: *AntiAliasSetting* = AntiAliasSetting(enable=False, override_anti_aliasing=False, spatial_samples=8, temporal_samples=8, warmup_frames=0, render_warmup_frame=False)

Anti aliasing setting of the render job.

field console_variables: Dict[str, float] = {'r.MotionBlurQuality': 0}

Additional console variables of the render job. Ref to *What is console_variables* for details.

field export_audio: bool = False

Whether to export audio of the render job.

field file_name_format: str = '{sequence_name}/{render_pass}/{camera_name}/{frame_number}'

File name format of the render job.

field map_path: str [Required]

Map path of the render job.

field output_path: str | Path [Required]

Output path of the render job.

field render_passes: List[*RenderPass*] [Required]

Render passes of the render job.

field resolution: Tuple[int, int] [Required]

Resolution of the images rendered by the render job.

field sequence_path: str [Required]

Sequence path of the render job.

pydantic model AntiAliasSetting


```
{
   "title": "AntiAliasSetting",
   "type": "object",
   "properties": {
      "enable": {
         "default": false,
         "description": "Whether to enable anti aliasing.",
         "title": "Enable",
         "type": "boolean"
      },
      "override_anti_aliasing": {
         "default": false,
         "description": "Whether to override anti aliasing.",
         "title": "Override Anti Aliasing",
         "type": "boolean"
      },
      "spatial_samples": {
         "default": 8,
         "description": "Spatial samples of the anti aliasing.",
         "minimum": 1,
         "title": "Spatial Samples",
         "type": "integer"
      },
      "temporal_samples": {
         "default": 8,
         "description": "Temporal samples of the anti aliasing.",
         "minimum": 1,
         "title": "Temporal Samples",
         "type": "integer"
      },
      "warmup_frames": {
         "default": 0,
         "description": "Warmup frames in engine which would rendered before␣
↪actual frame range. This is crucial when particle system (Niagara) is used.",
         "minimum": 0,
         "title": "Warmup Frames",
         "type": "integer"
      },
      "render_warmup_frame": {
         "default": false,
         "description": "Whether to render warmup frame of the anti aliasing.",
         "title": "Render Warmup Frame",
         "type": "boolean"
      }
   }
}
```


Fields

- enable (bool)
- override_anti_aliasing (bool)
- render_warmup_frame (bool)
- spatial_samples (int)
- temporal_samples (int)
- warmup_frames (int)

field enable: bool = False

Whether to enable anti aliasing.

field override_anti_aliasing: bool = False

Whether to override anti aliasing.

field render_warmup_frame: bool = False

Whether to render warmup frame of the anti aliasing.

field spatial_samples: int = 8

Spatial samples of the anti aliasing.

Constraints

- **ge** = 1

field temporal_samples: int = 8

Temporal samples of the anti aliasing.

Constraints

- **ge** = 1

field warmup_frames: int = 0

Warmup frames in engine which would rendered before actual frame range. This is crucial when particle system (Niagara) is used.

Constraints

- **ge** = 0

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

SequenceTransformKey

pydantic model xrfeitoria.data_structure.models.**SequenceTransformKey**

Transform key model for object (actor/camera) in sequence.

Examples

define

```python
from xrfeitoria.data_structure.models import SequenceTransformKey as SeqTransKey
key = SeqTransKey(
    frame=0,
    location=(0, 0, 0),
    rotation=(0, 0, 0),
    scale=(1, 1, 1),
    interpolation='AUTO'
)
```

Used in methods with suffix `_with_keys`, such as

- *Sequence.spawn_camera_with_keys*
- *Sequence.spawn_shape_with_keys*
- *SequenceUnreal.spawn_actor_with_keys*
- *SequenceBlender.import_actor_with_keys*
- *ObjectUtils.set_transform_keys*
- ...


```
{
   "title": "SequenceTransformKey",
   "description": "Transform key model for object (actor/camera) in sequence.\n\
↪nExamples:\n\n    .. tabs::\n\n        .. tab:: define\n\n            .. code-
↪block:: python\n                :linenos:\n\n                from xrfeitoria.data_
↪structure.models import SequenceTransformKey as SeqTransKey\n                key␣
↪= SeqTransKey(\n                    frame=0,\n                    location=(0, 0,␣
↪0),\n                    rotation=(0, 0, 0),\n                    scale=(1, 1, 1),
↪\n                    interpolation='AUTO'\n                )\n\nUsed in methods␣
↪with suffix ``_with_keys``, such as\n    - :meth:`Sequence.spawn_camera_with_keys
↪<xrfeitoria.sequence.sequence_base.SequenceBase.spawn_camera_with_keys>`\n    - :
↪meth:`Sequence.spawn_shape_with_keys <xrfeitoria.sequence.sequence_base.
↪SequenceBase.spawn_shape_with_keys>`\n    - :meth:`SequenceUnreal.spawn_actor_
↪with_keys <xrfeitoria.sequence.sequence_unreal.SequenceUnreal.spawn_actor_with_
↪keys>`\n    - :meth:`SequenceBlender.import_actor_with_keys <xrfeitoria.sequence.
↪sequence_blender.SequenceBlender.import_actor_with_keys>`\n    - :func:
↪`ObjectUtils.set_transform_keys <xrfeitoria.object.object_utils.
↪ObjectUtilsBlender.set_transform_keys>`\n    - ...",
   "type": "object",
   "properties": {
      "frame": {
         "description": "Frame number of the transform key, unit: frame.",
         "title": "Frame",
         "type": "integer"
      },
      "location": {
         "anyOf": [
            {
```


(continues on next page)

(continued from previous page)

```
                "maxItems": 3,
                "minItems": 3,
                "prefixItems": [
                   {
                      "type": "number"
                   },
                   {
                      "type": "number"
                   },
                   {
                      "type": "number"
                   }
                ],
                "type": "array"
             },
             {
                "type": "null"
             }
          ],
          "default": null,
          "description": "Location of the object in the transform key, unit: meter.",
          "title": "Location"
       },
       "rotation": {
          "anyOf": [
             {
                "maxItems": 3,
                "minItems": 3,
                "prefixItems": [
                   {
                      "type": "number"
                   },
                   {
                      "type": "number"
                   },
                   {
                      "type": "number"
                   }
                ],
                "type": "array"
             },
             {
                "type": "null"
             }
          ],
          "default": null,
          "description": "Rotation of the object in the transform key, unit: degree.
↪",
          "title": "Rotation"
       },
       "scale": {
          "anyOf": [
```

(continues on next page)

(continued from previous page)


```
            {
               "maxItems": 3,
               "minItems": 3,
               "prefixItems": [
                  {
                     "type": "number"
                  },
                  {
                     "type": "number"
                  },
                  {
                     "type": "number"
                  }
               ],
               "type": "array"
            },
            {
               "type": "null"
            }
         ],
         "default": null,
         "description": "Scale of the object in the transform key.",
         "title": "Scale"
      },
      "interpolation": {
         "anyOf": [
            {
               "$ref": "#/$defs/InterpolationEnumUnreal"
            },
            {
               "$ref": "#/$defs/InterpolationEnumBlender"
            }
         ],
         "description": "Interpolation type of the transform key.",
         "title": "Interpolation"
      }
   },
   "$defs": {
      "InterpolationEnumBlender": {
         "description": "Keyframe interpolation enum of Blender.",
         "enum": [
            "BEZIER",
            "LINEAR",
            "CONSTANT"
         ],
         "title": "InterpolationEnumBlender",
         "type": "string"
      },
      "InterpolationEnumUnreal": {
         "description": "Keyframe interpolation enum of Unreal.",
         "enum": [
            "AUTO",
```


(continues on next page)

(continued from previous page)

```
            "LINEAR",
            "CONSTANT"
         ],
         "title": "InterpolationEnumUnreal",
         "type": "string"
      }
   },
   "required": [
      "frame",
      "interpolation"
   ]
}
```

Fields

- *frame (int)*
- *interpolation (xrfeitoria.data_structure.constants.InterpolationEnumUnreal | xrfeitoria.data_structure.constants.InterpolationEnumBlender)*
- *location (Tuple[float, float, float] | None)*
- *rotation (Tuple[float, float, float] | None)*
- *scale (Tuple[float, float, float] | None)*

field frame: int [Required]

Frame number of the transform key, unit: frame.

field interpolation: *InterpolationEnumUnreal* | *InterpolationEnumBlender* [Required]

Interpolation type of the transform key.

field location: Tuple[float, float, float] | None = None

Location of the object in the transform key, unit: meter.

field rotation: Tuple[float, float, float] | None = None

Rotation of the object in the transform key, unit: degree.

field scale: Tuple[float, float, float] | None = None

Scale of the object in the transform key.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

Enumerations

xrfeitoria.data_structure.constants.EngineEnum(value)	Render engine enum.
xrfeitoria.data_structure.constants.ImportFileFormatEnum(value)	Import file format enum.
xrfeitoria.data_structure.constants.ImageFileFormatEnum(value)	Image file format enum.
xrfeitoria.data_structure.constants.RenderOutputEnumBlender(value)	Render layer enum of Blender.
xrfeitoria.data_structure.constants.RenderOutputEnumUnreal(value)	Render layer enum of Unreal.
xrfeitoria.data_structure.constants.InterpolationEnumUnreal(value)	Keyframe interpolation enum of Unreal.
xrfeitoria.data_structure.constants.InterpolationEnumBlender(value)	Keyframe interpolation enum of Blender.
xrfeitoria.data_structure.constants.RenderEngineEnumBlender(value)	Render engine enum of Blender.
xrfeitoria.data_structure.constants.ShapeTypeEnumBlender(value)	Shape type enum of Blender.
xrfeitoria.data_structure.constants.ShapeTypeEnumUnreal(value)	Shape type enum of Unreal.

EngineEnum

enum xrfeitoria.data_structure.constants.**EngineEnum**(*value*)

Render engine enum.

Valid values are as follows:

unreal = <EngineEnum.unreal: 1>

blender = <EngineEnum.blender: 2>

ImportFileFormatEnum

enum xrfeitoria.data_structure.constants.**ImportFileFormatEnum**(*value*)

Import file format enum.

Member Type
str

Valid values are as follows:

fbx = <ImportFileFormatEnum.fbx: 'fbx'>

obj = <ImportFileFormatEnum.obj: 'obj'>

abc = <ImportFileFormatEnum.abc: 'abc'>

ply = <ImportFileFormatEnum.ply: 'ply'>

stl = <ImportFileFormatEnum.stl: 'stl'>

glb = <ImportFileFormatEnum.glb: 'glb'>

ImageFileFormatEnum

enum xrfeitoria.data_structure.constants.**ImageFileFormatEnum**(*value*)

Image file format enum.

Member Type
str

Valid values are as follows:

png = <ImageFileFormatEnum.png: 'PNG'>

bmp = <ImageFileFormatEnum.bmp: 'BMP'>

jpg = <ImageFileFormatEnum.jpg: 'JPEG'>

exr = <ImageFileFormatEnum.exr: 'OPEN_EXR'>

RenderOutputEnumBlender

enum xrfeitoria.data_structure.constants.**RenderOutputEnumBlender**(*value*)

Render layer enum of Blender.

Member Type
str

Valid values are as follows:

img = <RenderOutputEnumBlender.img: 'Image'>

mask = <RenderOutputEnumBlender.mask: 'IndexOB'>

depth = <RenderOutputEnumBlender.depth: 'Depth'>

denoising_depth = <RenderOutputEnumBlender.denoising_depth: 'Denoising Depth'>

flow = <RenderOutputEnumBlender.flow: 'Vector'>

normal = <RenderOutputEnumBlender.normal: 'Normal'>

diffuse = <RenderOutputEnumBlender.diffuse: 'DiffCol'>

actor_infos = <RenderOutputEnumBlender.actor_infos: 'actor_infos.json'>

camera_params = <RenderOutputEnumBlender.camera_params: 'camera_params'>

RenderOutputEnumUnreal

enum xrfeitoria.data_structure.constants.**RenderOutputEnumUnreal**(*value*)

Render layer enum of Unreal.

Member Type
str

Valid values are as follows:

img = <RenderOutputEnumUnreal.img: 'img'>

mask = <RenderOutputEnumUnreal.mask: 'mask'>

depth = <RenderOutputEnumUnreal.depth: 'depth'>

flow = <RenderOutputEnumUnreal.flow: 'flow'>

normal = <RenderOutputEnumUnreal.normal: 'normal'>

diffuse = <RenderOutputEnumUnreal.diffuse: 'diffuse'>

metallic = <RenderOutputEnumUnreal.metallic: 'metallic'>

roughness = <RenderOutputEnumUnreal.roughness: 'roughness'>

specular = <RenderOutputEnumUnreal.specular: 'specular'>

tangent = <RenderOutputEnumUnreal.tangent: 'tangent'>

basecolor = <RenderOutputEnumUnreal.basecolor: 'basecolor'>

lineart = <RenderOutputEnumUnreal.lineart: 'lineart'>

vertices = <RenderOutputEnumUnreal.vertices: 'vertices'>

skeleton = <RenderOutputEnumUnreal.skeleton: 'skeleton'>

actor_infos = <RenderOutputEnumUnreal.actor_infos: 'actor_infos'>

camera_params = <RenderOutputEnumUnreal.camera_params: 'camera_params'>

audio = <RenderOutputEnumUnreal.audio: 'Audio'>

InterpolationEnumUnreal

enum xrfeitoria.data_structure.constants.**InterpolationEnumUnreal**(*value*)

Keyframe interpolation enum of Unreal.

Member Type
str

Valid values are as follows:

AUTO = <InterpolationEnumUnreal.AUTO: 'AUTO'>

LINEAR = <InterpolationEnumUnreal.LINEAR: 'LINEAR'>

CONSTANT = <InterpolationEnumUnreal.CONSTANT: 'CONSTANT'>

InterpolationEnumBlender

enum xrfeitoria.data_structure.constants.**InterpolationEnumBlender**(*value*)

Keyframe interpolation enum of Blender.

Member Type
str

Valid values are as follows:

AUTO = <InterpolationEnumBlender.AUTO: 'BEZIER'>

LINEAR = <InterpolationEnumBlender.LINEAR: 'LINEAR'>

CONSTANT = <InterpolationEnumBlender.CONSTANT: 'CONSTANT'>

RenderEngineEnumBlender

enum xrfeitoria.data_structure.constants.**RenderEngineEnumBlender**(*value*)

Render engine enum of Blender.

Member Type
str

Valid values are as follows:

cycles = <RenderEngineEnumBlender.cycles: 'CYCLES'>

eevee = <RenderEngineEnumBlender.eevee: 'BLENDER_EEVEE'>

workbench = <RenderEngineEnumBlender.workbench: 'BLENDER_WORKBENCH'>

ShapeTypeEnumBlender

enum xrfeitoria.data_structure.constants.**ShapeTypeEnumBlender**(*value*)

Shape type enum of Blender.

Member Type
str

Valid values are as follows:

cone = <ShapeTypeEnumBlender.cone: 'cone'>

cube = <ShapeTypeEnumBlender.cube: 'cube'>

cylinder = <ShapeTypeEnumBlender.cylinder: 'cylinder'>

plane = <ShapeTypeEnumBlender.plane: 'plane'>

sphere = <ShapeTypeEnumBlender.sphere: 'uv_sphere'>

ico_sphere = <ShapeTypeEnumBlender.ico_sphere: 'ico_sphere'>

ShapeTypeEnumUnreal

enum xrfeitoria.data_structure.constants.**ShapeTypeEnumUnreal**(*value*)

Shape type enum of Unreal.

Member Type
str

Valid values are as follows:

cone = <ShapeTypeEnumUnreal.cone: 'cone'>

cube = <ShapeTypeEnumUnreal.cube: 'cube'>

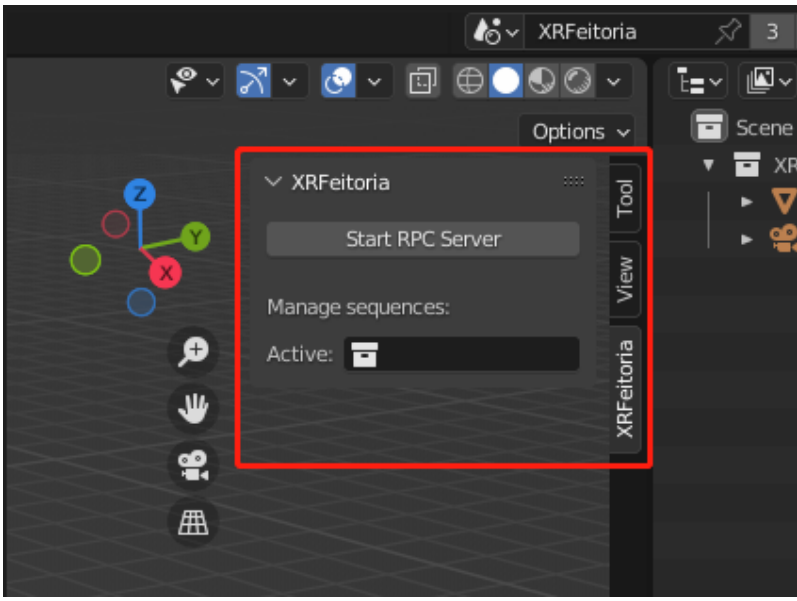
cylinder = <ShapeTypeEnumUnreal.cylinder: 'cylinder'>

plane = <ShapeTypeEnumUnreal.plane: 'plane'>

sphere = <ShapeTypeEnumUnreal.sphere: 'sphere'>

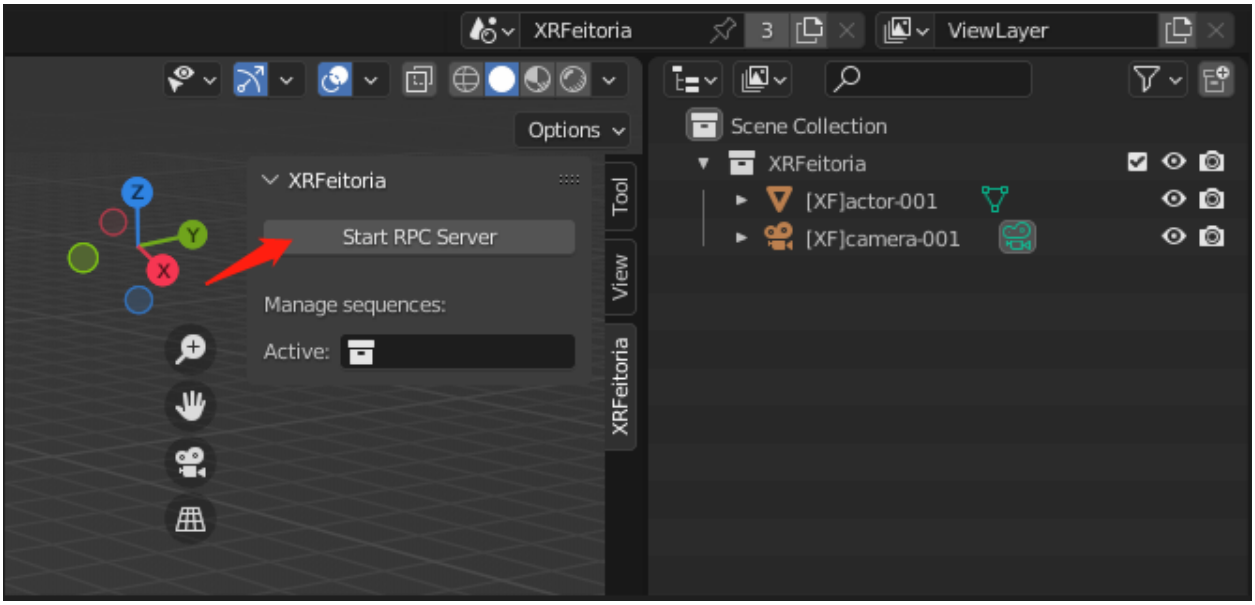
2.1.12 XRFeitoria Blender Addon

By running `xf.init_blender`, a XRFeitoria addon will be installed in your Blender. The source code of the addon is in `src/XRFeitoriaBpy`. It enables you to start RPC server and manage your sequences.

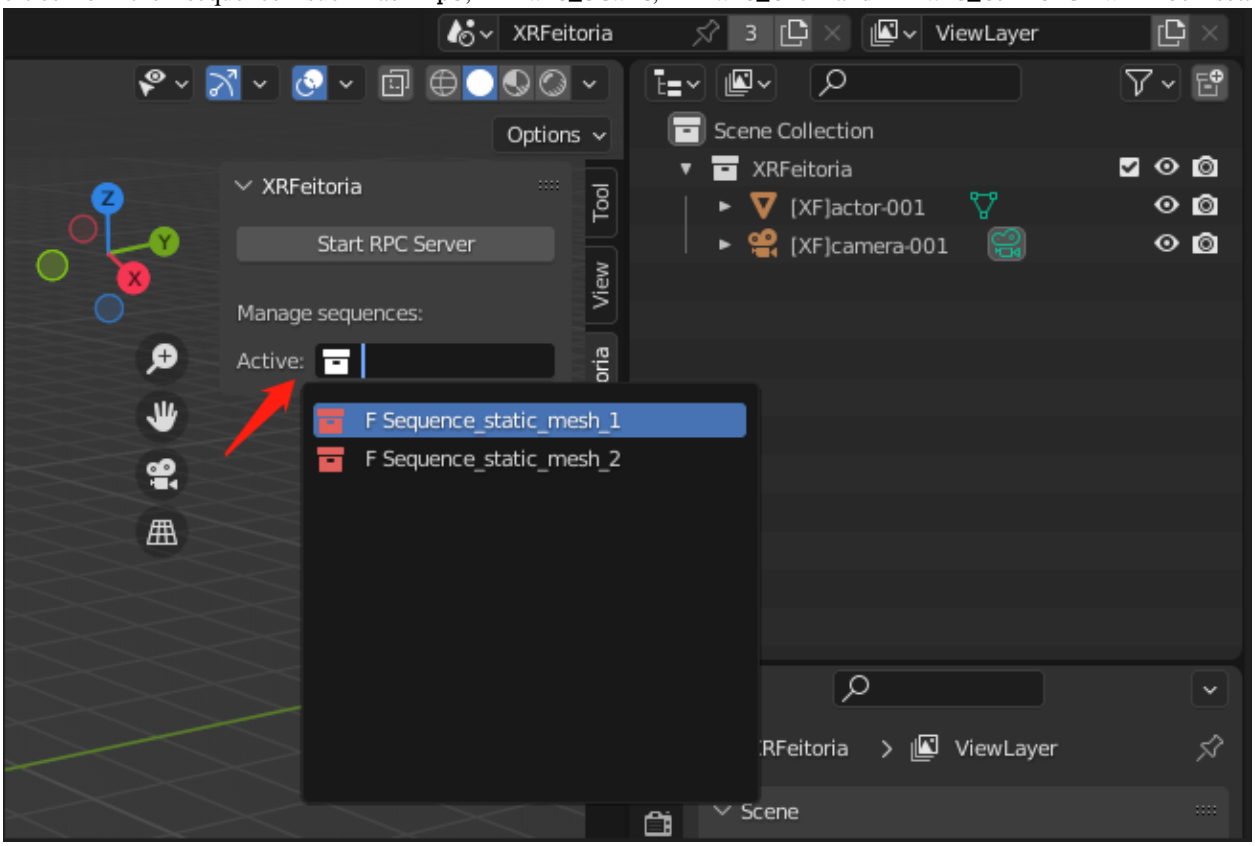


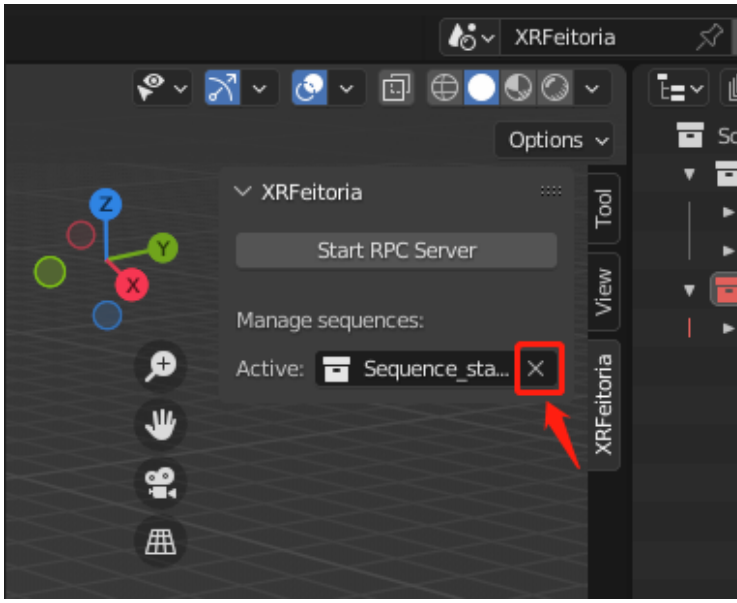
The addon panel is in the top-right corner of the 3D Viewport.

Click the button `Start RPC Server` to start the RPC server. The server will listen on port 9997 to receive commands from XRFeitoria and execute them in Blender.



Click the drop down box `Active` in the `Manage sequences` section to open a sequence belonging to the current level. The objects in the sequence will be loaded to the window, and the properties of the sequence such as `fps`, `frame_start`, `frame_end` and `frame_current` will be set.





Click the 'x' button beside the drop down box to close the sequence.

2.1.13 XRFeitoria Unreal Plugin

TBD

2.1.14 Frequently Asked Questions

We list some common troubles faced by many users and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them. If the contents here do not cover your issue, do not hesitate to create an issue!

API

How to use the plugin of Blender/Unreal under development

First you should clone the repo of XRFeitoria, and maybe modify the code of the plugin under `src/XRFeitoriaBlender` or `src/XRFeitoriaUnreal`. Then you can use the plugin under development by setting `dev_plugin=True` in *`init_blender`* or *`init_unreal`*.

You can install the plugin by:

Blender

Unreal

```
git clone https://github.com/openxrlab/xrfeitoria.git
cd xrfeitoria
pip install -e .
python -c "import xrfeitoria as xf; xf.init_blender(replace_plugin=True, dev_plugin=True)
↪"
```

(continues on next page)

(continued from previous page)

```

# or through the code in tests
python -m tests.blender.init --dev [-b]
```

```
git clone https://github.com/openxrlab/xrfeitoria.git
cd xrfeitoria
pip install -e .
python -c "import xrfeitoria as xf; xf.init_unreal(replace_plugin=True, dev_plugin=True)"

# or through the code in tests
python -m tests.unreal.init --dev [-b]
```

Build plugins

If you want to publish plugins of your own, you can use the following command:

```
# install xrfeitoria first
cd xrfeitoria
pip install .

# for instance, build plugins for Blender, UE 5.1, UE 5.2, and UE 5.3 on Windows.
# using powershell where backtick(`) is the line continuation character.
python -m xrfeitoria.utils.publish_plugins `
  -u "C:/Program Files/Epic Games/UE_5.1/Engine/Binaries/Win64/UnrealEditor-Cmd.exe" `
  -u "C:/Program Files/Epic Games/UE_5.2/Engine/Binaries/Win64/UnrealEditor-Cmd.exe" `
  -u "C:/Program Files/Epic Games/UE_5.3/Engine/Binaries/Win64/UnrealEditor-Cmd.exe"
```

Please check the path `./src/dist` for the generated plugins. `XRFeitoriaBlender` will be archived by default, and `XRFeitoriaUnreal` will only be built when you specify the Unreal editor path. Make sure you have installed the corresponding Unreal Engine and Visual Studio before building the unreal plugin.

Find out the plugin version in `./xrfeitoria/version.py`. Or by:

```
>>> python -c "import xrfeitoria; print(xrfeitoria.__version__)"
0.6.1.dev10+gd12997e.d20240122
```

You can set the environment variable `XRFEITORIA__DIST_ROOT` and `XRFEITORIA__VERSION` to change the plugins used by XRFeitoria. Run your code `xxx.py` like:

UNIX

Windows

```
XRFEITORIA__VERSION=$(python -c "import xrfeitoria; print(xrfeitoria.__version__)") \
XRFEITORIA__DIST_ROOT=src/dist \
    python xxx.py
```

```
$env:XRFEITORIA__VERSION=$(python -c "import xrfeitoria; print(xrfeitoria.__version__)")
$env:XRFEITORIA__DIST_ROOT="src/dist"; `
    python xxx.py
```

What is `stencil_value`

`stencil_value` is to distinguish different actors in the sequence when rendering segmentation masks. The RGB mask color of actors will be saved in the `{output_path}/actor_infos.json` of the render job.

In:

- *ActorBase.import_from_file*
- *SequenceUnreal.spawn_actor*
- ...

What is `console_variables`

`console_variables` is a dictionary of console variables for configure detailed rendering settings. Please refer to the official documentation for more details: Unreal-MRQ-Doc.

example:

```
>>> console_variables = {'r.MotionBlurQuality': 0}  # disable motion blur
```

In:

- *RenderJobUnreal.console_variables*
- ...

RPC Port

The RPC port is used for communication between python and engine (blender/unreal). If the default port is occupied, or you want to use multiple engines at the same time, you can set the environment variable `BLENDER_PORT` or `UNREAL_PORT` to change it.

UNIX

Windows

```
BLENDER_PORT=50051 python xxx.py
```

```
$env:BLENDER_PORT=50051; python xxx.py
```

PYTHON MODULE INDEX

X

INDEX

Symbols

A

B

C

G

H

I

J

L

M

N

O

P

R

S