Starfish

Kevin Black

Sep 11, 2020

CONTENTS

Starfish is a Python library for automatically creating synthetic, labeled image data using Blender.

This library it extends Blender's powerful Python scripting ability, providing utilities that make it easy to generate long sequences of synthetic images with intuitive parameters. It was designed for people who, like me, know Python much better than they know Blender.

These sequences can be smoothly interpolated from waypoint to waypoint, much like a traditional keyframe-based animation. They can also be exhaustive or random, containing images with various object poses, backgrounds, and lighting conditions. The intended use for these sequences is the generation of training and evaluation data for machine learning tasks where annotated real data may be difficult to obtain.

Contents

CHAPTER

ONE

API DOCUMENTATION

1.1 Core (Frame and Sequence)

class starfish.**Frame**(*, *position=0, 0, 0*, *distance=100*, *pose=Quaternion(1.0, 0.0, 0.0, 0.0)*, *lighting=Quaternion(1.0, 0.0, 0.0, 0.0)*, *offset=0.5, 0.5*, *background=Quaternion(1.0, 0.0, 0.0, 0.0)*)

Represents a single picture of an object with certain parameters.

There are 6 parameters that independently define a picture:

- Object position: the absolute 3D position of the object in the global coordinate system (i.e. where it is in the scene).
- Camera distance: the distance of the camera from the object.
- Object pose: the pose of the object relative to the camera (i.e. how it will appear to be oriented in the picture).
- Lighting: the angle from which the sun's rays will hit the object in the picture (e.g. from above, from the right, from behind the camera, etc.).
- Object offset: the 2D translational offset of the object from the center of the picture frame.
- Background/camera orientation: the orientation of the camera and object relative to the global coordinate system. This affects only what part of the scene appears in the background directly behind the object.

A note on coordinate systems: the representations for `pose` and *`translation`* were carefully chosen to match those of OpenCV rather than using Blender's default coordinate system. This means that OpenCV camera projection functions such as `projectPoints` and `solvePnP` should produce correct results when `pose` and *`translation`* are treated as the `rvec` and `tvec`, respectively.

__init__(*, *position=0, 0, 0*, *distance=100*, *pose=Quaternion(1.0, 0.0, 0.0, 0.0)*, *lighting=Quaternion(1.0, 0.0, 0.0, 0.0)*, *offset=0.5, 0.5*, *background=Quaternion(1.0, 0.0, 0.0, 0.0)*)

Initializes a picture with all of the parameters it needs.

A type of "rotation" means a `mathutils.Quaternion` object or any object with a to_quaternion() method (which includes `mathutils.Euler`, `mathutils.Matrix`, and *`starfish.rotations.Spherical`*).

Parameters

- **position** – (seq, len 3): the (x, y, z) absolute position of the object in the scene's global coordinate system (default: (0, 0, 0))
- **distance** – (float or int): the distance of the camera from the object in blender units (default: 100)

- **pose** – (rotation): the orientation of the object relative to the camera's coordinate system (default: the identity quaternion (aka zero rotation), which corresponds to the camera looking directly in the object's +Z direction with the object's +X direction pointing to the right and +Y pointing down)
- **lighting** – (rotation): the angle of the sun's lighting relative to the camera's coordinate system (default: the identity quaternion (aka zero rotation), which corresponds to the light coming from directly behind the camera)
- **offset** – (seq of float, len 2): the (vertical, horizontal) translational offset of the object from the center of the picture frame. Expressed as a fraction of the distance from edge to edge: e.g., for horizontal offset, 0.0 is the left edge, 0.5 is the center, and 1.0 is the right edge. Same for vertical, but 0.0 is the top edge and 1.0 is the bottom. (default: (0.5, 0.5))
- **background** – (rotation): Imagine a ray starting at the camera and passing through the object. This parameter determines the orientation of this ray in the global coordinate system. For example, if you have a world background image that encircles your entire scene, two degrees of freedom of this parameter will determine the point in the background image that will appear directly behind the object, and the third degree of freedom will determine the rotation of this background image (i.e. which way is 'up'). (default: the identity quaternion (aka zero rotation), which corresponds to the camera->object ray pointing directly in the -Z direction with the +X direction pointing to the right and +Y pointing up)

dumps()
Converts all of the frame's attributes to a JSON object. By default, this will be the 6 frame parameters, plus *translation* if *setup* has already been called. Any additional metadata can be added by just setting it as an attribute: e.g. `frame.sequence_name = '20k_square_earth_background'; metadata = frame.dumps()`

setup(*scene*, *obj*, *camera*, *sun*)
Sets up a camera, object, and sun into the picture-taking position. Also computes and stores the translation vector of the object.

Parameters

- **scene** – (BlendDataObject): the scene to use for aspect ratio calculations. Note that this should be the scene that you intend to perform the final render in, not necessarily the one that your objects exist in. If you render in a scene that has an output resolution with a different aspect ratio than the output resolution of this scene, then the offset of the object may be incorrect.
- **obj** – (BlendDataObject): the object that will be the subject of the picture
- **camera** – (BlendDataObject): the camera to take the picture with
- **sun** – (BlendDataObject): the sun lamp that is providing the lighting

translation = None
The object's position relative to the camera represented by a single translation vector (in Blender units). This value isn't computed until setup time, and will be `None` beforehand. If you need this translation vector as part of your metadata, make sure to call *setup* first before calling *dumps*.

class starfish.**Sequence**(*frames*)
Represents a sequence of frames.

This class acts exactly like a list of starfish.Frame objects, with a few utility methods tacked on.

Most of the power of this class comes from the classmethod constructors, which can be used to create different types of sequences in a more convenient, expressive way.

The bake method is useful for previewing sequences in Blender while they are being tweaked and configured, in order to get an idea of what they will look like before rendering. However, it is not recommended to use bake at render time. Instead, the sequence object can be iterated over frame-by-frame, like so:

```
for frame in sequence:
    frame.setup(...)
    bpy.ops.render.render(...)
```

__init__(*frames*)
Initializes a sequence from a list of frames.

bake(*scene*, *obj*, *camera*, *sun*, *num=None*)
Creates keyframes representing this sequence, so that it can be played as a preview animation. Keyframes will be adjacent to each other, so no interpolation will be done. This is just a means to get an idea of what frames are in the sequence. If `len(frames)` is greater than `num`, only every `(len(frames) / num)` frames will be displayed.

This should not be called with large values of `num` (>5000), as it is quite slow and may crash Blender.

Parameters

- **scene** – (BlendDataObject): the scene to set up the animation in
- **obj** – (BlendDataObject): the object that will be the subject of the picture
- **camera** – (BlendDataObject): the camera to take the picture with
- **sun** – (BlendDataObject): the sun lamp that is providing the lighting
- **num** – (int): The number of keyframes to generate. Defaults to `min(100, len(frames))`

Returns A *Sequence* object.

classmethod exhaustive(***kwargs*)
Creates a sequence that includes every possible combination of the parameters given.

The arguments to this constructor are the same as those to the *Frame* constructor, except instead of a single value, each argument may also be a list of values. For example, while `position` is normally an iterable of length 3 representing a 3D vector, it could instead be a list of 3D vectors (i.e. an array of shape (n, 3)).

This constructor then takes the lists of values for each parameter and generates frames out of their cartesian product. For example, if 10 distances, 10 poses, and 10 offsets are provided, the generated sequence will be 10*10*10 = 10,000 frames long, including every possible combination of given distances, poses, and offsets.

Returns A *Sequence* object.

classmethod interpolated(*waypoints*, *counts*)
Creates a sequence interpolated from a list of waypoints.

Parameters

- **waypoints** – (seq): A starfish.Sequence object (or just a list of starfish.Frame objects) representing the waypoints to interpolate between. (tip: use the starfish.Sequence.standard constructor to create this.)
- **counts** – (int or seq): The number of frames to generate between each pair of waypoints. There will be counts[i] frames in between waypoints[i] (inclusive) and waypoints[i+1] (exclusive). The total number of frames in the sequence will be sum(counts) + 1. The length of counts should be 1 less than the length of waypoints. (If count is an integer, then there should only be 2 waypoints.)

Returns A *Sequence* object.

classmethod standard(***kwargs*)

Creates a sequence from parameters that are lists.

The arguments to this constructor are the same as those to the *Frame* constructor, except instead of a single value, each argument may also be a list of values. For example, while `position` is normally an iterable of length 3 representing a 3D vector, it could instead be a list of 3D vectors (i.e. an array of shape (n, 3)).

This constructor then generates a list of frames where the parameters for each frame come from these lists, zipped together.

Each list of parameters must be either the same length as all the others, or be list with a single value. If a single value is provided for a parameter, then that value is broadcasted across all the frames, i.e. every frame gets that value for that parameter. (The same thing happens if a parameter is omitted: every frame gets the default value for that parameter).

For example: `Sequence(distance=[100, 200, 300])` will generate a sequence of 3 frames where the distances are 100, 200, and 300, while all other parameters are the default. `Sequence(position=[(1, 1, 1)], distance=[100, 200, 300])` will generate a sequence of 3 frames where the distances are 100, 200, and 300, while the positions are (1, 1, 1) and all other parameters are the default.

Returns A *Sequence* object.

1.2 Utils

starfish.utils.**cartesian**(**arrays*)

Returns the cartesian product of multiple 1D arrays. For example, `cartesian([0], [1, 2], [3, 4, 5])` returns:

```
array([[0, 1, 3],
       [0, 1, 4],
       [0, 1, 5],
       [0, 2, 3],
       [0, 2, 4],
       [0, 2, 5]])
```

Works with arbitrary objects.

starfish.utils.**jsonify**(*obj*)

Serializes an object's attributes into a JSON string with support for mathutils objects.

All rotation objects are converted to a 4-element list representing wxyz quaternion form. All vectors are converted to a 3-element list.

starfish.utils.**random_rotations**(*n*)

Generates n rotations sampled uniformly from the group of all 3D rotations, SO(3).

Parameters **n** – (int): number of rotations to generate

Returns List of `mathutils.Quaternion` objects.

starfish.utils.**uniform_sphere**(*n*, *random=None*)

Generates n points on the surface of a sphere that are "evenly spaced" using the golden spiral method. Based on https://stackoverflow.com/a/44164075.

Parameters

- **n** – (int): number of points to generate over the surface of the sphere
- **random** – (int): if None, return all generated points. Otherwise, randomly sample this many points from the generated ones (default: None)

Returns A tuple of the form (theta, phi), where theta and phi are each numpy arrays of length n. theta is the azimuthal angle, and phi is the polar angle.

1.3 Annotation

`starfish.annotation.`**`generate_keypoints`**(*obj*, *num*, *stop=1*, *oversample=10*, *seed=0*)

Generates evenly spaced 3D keypoints on the surface of an object.

This function implements the Sample Elimination algorithm from this paper to generate points on the surface of the object that follow a Poisson Disk distribution. The Poisson Disk distribution guarantees that no two points are within a certain distance of each other in 3D space, ensuring that the keypoints are more spread out.

The way Sample Elimination works is by first generating `num * oversample` points at random, and then eliminating points in a certain order until there are `num` left. Thus, a higher value of `oversample` will give more evenly spaced points.

This also has the nice property that every intermediary set of points also follows a Poisson Disk distribution. By default, this function will keep running sample elimination until there is 1 point left, and then return the points in reverse order of elimination so that the first `n` points are also evenly spaced out for any `1 <= n <= num`. The point at which Sample Elimination stops can be controlled with the `stop` parameter.

Parameters

- **obj** – (BlendDataObject): Blender object to operate on
- **num** – (int): number of points to generate
- **stop** – (int): an integer between 1 and `num` (inclusive) at which sample elimination will stop, default 1
- **oversample** – (float): amount of oversampling to do (see above), default 10
- **seed** – (int): seed for the initial random point generation

Returns A list of length `num` containing 3-tuples representing the coordinates of the keypoints in object space. The first `n` elements of the list will also be evenly spaced out for any `stop <= n <= num`.

`starfish.annotation.`**`project_keypoints_onto_image`**(*keypoints*, *scene*, *obj*, *camera*)

Converts 3D keypoints of an object into their corresponding 2D coordinates on the image.

This function takes a list of keypoints represented as 3D coordinates in object space, and then projects them onto the camera to get their corresponding 2D coordinates on the image. It uses the current location and orientation of the input Blender objects. Typical usage would be to call this function after *`Frame.setup`* and then store the 2D locations as metadata for that frame:

```
frame.setup(scene, obj, camera, sun)
frame.keypoints = project_keypoints_onto_image(keypoints, scene, obj, camera)
with open('meta...', 'w') as f:
    f.write(frame.dumps())
```

Parameters

- **keypoints** – a list of 3D coordinates corresponding to the locations of the keypoints in the object space, e.g. the output of *`generate_keypoints`*

- **scene** – (BlendDataObject): the scene to use for aspect ratio calculations. Note that this should be the scene that you intend to perform the final render in, not necessarily the one that your objects exist in. If you render in a scene that has an output resolution with a different aspect ratio than the output resolution of this scene, then the results may be incorrect.
- **obj** – (BlendDataObject): the object to use
- **camera** – (BlendDataObject): the camera to use

Returns a list of (y, x) coordinates in the same order as `keypoints` where (0, 0) is the top left corner of the image and (1, 1) is the bottom right

`starfish.annotation.`**`normalize_mask_colors`**(*mask*, *colors*, *color_variation_cutoff=6*)

Normalizes the colors of a mask image.

Blender has a bug where the colors in a mask image vary slightly (e.g. instead of the background being solid rgb(0, 0, 0) black, it will actually be a random mix of rgb(0, 0, 1), rgb(1, 1, 0), etc…). This function takes a mask as well as a map of what the colors are supposed to be, then eliminates this variation.

This function accepts either the path to the mask (str) or the mask itself represented as a numpy array. If a path is provided, then the function will return the normalized mask as well as overwrite the original mask on disk. If a numpy array is provided, then the function will just return the normalized mask.

Parameters

- **mask** – path to mask image (str) or numpy array of mask image (RGB)
- **colors** – a list of what the label colors are supposed to be, each in [R, G, B] format
- **color_variation_cutoff** – colors will be allowed to differ from a color in the label map by a cityblock distance of no more than this value. The default value is 6, or equivalently 2 in each RGB channel. I chose this value because, in my experience with Blender 2.8, the color variation is no more than 1 in each channel, a number I then doubled to be safe.

Returns the normalized mask as a numpy array

`starfish.annotation.`**`get_bounding_boxes_from_mask`**(*mask*, *label_map*)

Gets bounding boxes from instance masks.

Parameters

- **mask** – path to mask image (str) or numpy array of mask image (RGB)
- **label_map** – dictionary mapping classes (str) to their corresponding color(s). Each class can correspond to a single color (e.g. `{"cygnus": (0, 0, 206)}`) or multiple colors (e.g. `{"cygnus": [(0, 0, 206), (206, 0, 0)]}`)

Returns a dictionary mapping classes (str) to their corresponding bboxes (a dictionary with the keys ‘xmin’, ‘xmax’, ‘ymin’, ‘ymax’). If a class does not appear in the image, then it will not appear in the keys of the returned dictionary.

`starfish.annotation.`**`get_centroids_from_mask`**(*mask*, *label_map*)

Gets centroids from instance masks.

Parameters

- **mask** – path to mask image (str) or numpy array of mask image (RGB)
- **label_map** – dictionary mapping classes (str) to their corresponding color(s). Each class can correspond to a single color (e.g. `{"cygnus": (0, 0, 206)}`) or multiple colors (e.g. `{"cygnus": [(0, 0, 206), (206, 0, 0)]}`)

Returns a dictionary mapping classes (str) to their corresponding centroids (y, x). If a class does not appear in the image, then it will not appear in the keys of the returned dictionary.

1.4 Rotations

This module is for alternative 3D rotation formalisms besides the Quaternion, Matrix, and Euler representations provided by the `mathutils` library. They must implement the `to_quaternion` method, which returns a `mathutils.Quaternion` instance, in order to be compatible with the rest of this library. A `from_other` classmethod may also be useful, in order to convert from a mathutils representation to the alternative representation.

class starfish.rotations.**Spherical**(*theta*, *phi*, *roll*)

An alternative 3-value representation of a rotation based on spherical coordinates.

Imagine a unit sphere centered about an object. Two spherical coordinates (an azimuthal angle, henceforth theta, and a polar angle, henceforth phi) define a point on the surface of the sphere, and a corresponding unit vector from the center of the sphere (the object) to the point on the surface of the sphere.

First, the +Z axis of the object is rotated to this unit vector, while the XY plane of the object is aligned such that the +X axis points in the +phi direction and the +Y axis points in the +theta direction. It may be helpful to visualize this rotation as such: imagine that the +Z axis of the object is a metal rod attached rigidly to the object, extending out through the surface of the sphere. Now grab the rod and use it to rotate the object such that the rod is passing through the point on the sphere defined by theta and phi. Finally, twist the rod such that the original "right" direction of the object (its +X axis) is pointing towards the south pole of the sphere, along the longitude line defined by theta. Correspondingly, this should mean that the original "up" direction of the object (its +Y axis) is pointing eastward along the latitude line defined by phi.

Next, perform a right-hand rotation of the object about the same unit vector by a third angle (henceforth called the roll angle). In the previous analogy, this is equivalent to then twisting the metal rod counter-clockwise by the roll angle. This configuration is the final result of the rotation.

Note: the particular alignment of the XY plane (+X is +phi and +Y is +theta) was chosen so that "zero rotation" (aka the identity quaternion, or (0, 0, 0) Euler angles) corresponds to (theta, phi, roll) = (0, 0, 0).

Also note that this representation only uses 3 values, and thus it has singularities at the poles where theta and the roll angle are redundant (only their sum matters).

Attributes:

- theta: The azimuthal angle, in radians
- phi: The polar angle, in radians (0 at the north pole, pi at the south pole)
- roll: The roll angle, in radians

classmethod from_other(*obj*)

Constructs a Spherical object from a Quaternion, Euler, or Matrix rotation object from the mathutils library.

to_quaternion()

Returns a `mathutils.Quaternion` representation of the rotation.

CHAPTER

TWO

REQUIREMENTS

Though Starfish has only been tested with Blender 2.82, it should work with any version 2.8+. Please open an issue on GitHub if it doesn't.

CHAPTER

THREE

INSTALLATION

1. Identify the location of your Blender scripts directory. This can be done by opening Blender, clicking on the 'Scripting' tab, and entering `bpy.utils.script_path_user()` in the Python console at the bottom. Generally, on Linux, the default location is `~/.config/blender/[VERSION]/scripts`. From now on, this path will be referred to as `[SCRIPTS_DIR]`.
2. Create the addon modules directory, if it does not exist already: `mkdir -p [SCRIPTS_DIR]/addons/modules`
3. Install the library to Blender: `pip install git+https://github.com/autognc/starfish --no-deps --target [SCRIPTS_DIR]/addons/modules`. Starfish does not require any additional packages besides what is already bundled with Blender, which is why `--no-deps` can be used.

Starfish can also be pip-installed normally without Blender for testing purposes or for independent usage of certain modules.

CHAPTER

FOUR

QUICKSTART

4.1 Recommended Reading

To use Starfish, you're probably going to have to interact with Blender using the Python API. This library also makes heavy use of mathutils, which is an independent math library that comes bundled with Blender.

4.2 Running Scripts in Blender

The easiest way to experiment with the library is by opening Blender, navigating to the Scripting tab, and hitting the plus button to create a new script. You can then import starfish, write some code, and hit `Alt+P` to see what it does.

Once you're ready to execute a more long-running script, you can write it outside Blender and then execute it using `blender file.blend --background --python script.py` (or `blender file.blend -b -P script.py` for short).

4.3 Generating Images

4.3.1 Frames

At the core of Starfish is the *`Frame`* class, which represents a single image of a single object. A frame is defined by 6 parameters:

```
frame = starfish.Frame(
    position=(0, 0, 0),
    distance=10,
    pose=mathutils.Euler([0, math.pi, 0]),
    lighting=mathutils.Euler([0, 0, 0]),
    offset=(0.3, 0.7),
    background=mathutils.Euler([math.pi / 2, 0, 0])
)
```

See the *`starfish.Frame`* documentation for more details about what each parameter means. Once you have a frame object, you use it to 'set up' your scene:

```
frame.setup(
    bpy.data.scenes['MyScene'],
    bpy.data.objects['MyObject'],
    bpy.data.objects['Main_Camera'],
```

(continues on next page)

(continued from previous page)

```
    bpy.data.objects['The_Sun']
)
```

This moves all the objects so that the image that the camera sees matches up with the parameters in the frame object. At this point, you can render the frame using `bpy.ops.render.render`. You can also dump metadata about a frame into JSON format using the *`Frame.dumps`* method.

4.3.2 Sequences

Of course, Starfish wouldn't be very useful without the ability to create multiple frames at once. The *`Sequence`* class is essentially just a list of frames, but with several classmethod constructors for generating these sequences of frames in different ways. For example, *`Sequence.interpolated`* generates 'animated' sequences that smoothly interpolate between keyframes, and *`Sequence.exhaustive`* generates long sequences that contain every possible combination of the parameters given.

Once you've created a sequences, you can iterate through its frames like so:

```
seq = starfish.Sequence...

for frame in seq:
    frame.setup(...)
    bpy.ops.render.render()
```

The *`Sequence.bake`* method also provides an easy way to 'preview' sequences that you're working on in Blender. See *`Sequence`* for more detail.

4.3.3 Utils

The *`utils`* module provides a few more functions that may be useful for core image generation, such as *`random_rotations`* or *`uniform_sphere`*.

4.3.4 Annotation

Starfish also contains an *`annotation`* module that provides utility functions related to annotating generated data.

- One common type of annotation generated for computer vision tasks is some sort of segmentation mask (e.g. using the ID Mask Node) where having perfectly uniform colors is important. Unfortunately, I've often encountered an issue in Blender where the output colors differ slightly: for example, instead of the background being solid `rgb(0, 0, 0)` black, it will actually be a random mix of `rgb(0, 0, 1)`, `rgb(1, 1, 0)`, etc. The *`normalize_mask_colors`* function can be used to clean up such images.
- Once a mask has been cleaned up, *`get_bounding_boxes_from_mask`* and *`get_centroids_from_mask`* can be used to get the bounding boxes and centroids of segmented areas, respectively.
- Another common type of annotation is keypoints: e.g. where particular 3D points on the object appear in the 2D image. *`generate_keypoints`* can be used to automatically generate evenly distributed 3D keypoints from an object's mesh; *`project_keypoints_onto_image`* can then take these keypoints and map them to 2D image locations after rendering a particular frame.

4.4 Example Script

All together, here is what an image generation script might look like:

```
"""
IMPORTANT NOTE: This script is just for demonstrating the various capabilities of␣
↪Starfish, and is not meant
to be run as-is. If you try to run this script without modifications, it will␣
↪probably not work, unless you have
your Blend file set up with the exact same scenes, objects, and compositing nodes.␣
↪Even then, it will immediately
start rendering several long sequences and writing files to disk, with the files from␣
↪each sequence overwriting
the files from the previous one.
"""
import time
import bpy
import numpy as np
from mathutils import Euler
from starfish import Sequence
from starfish.utils import random_rotations
from starfish.annotation import normalize_mask_colors, get_centroids_from_mask, get_
↪bounding_boxes_from_mask

# create a standard sequence of random configurations...
seq1 = Sequence.standard(
    pose=random_rotations(100),
    lighting=random_rotations(100),
    background=random_rotations(100),
    distance=np.linspace(10, 50, num=100)
)
# ...or an exhaustive sequence of combinations...
seq2 = Sequence.exhaustive(
    distance=[10, 20, 30],
    offset=[(0.25, 0.25), (0.25, 0.75), (0.75, 0.25), (0.75, 0.75)],
    pose=[Euler((0, 0, 0)), Euler((np.pi, 0, 0))]
)
# ...or an interpolated sequence between keyframes...
seq3 = Sequence.interpolated(
    waypoints=Sequence.standard(distance=[10, 20], pose=[Euler((0, 0, 0)), Euler((0,␣
↪np.pi, np.pi))]),
    counts=[100]
)

for seq in [seq1, seq2, seq3]:
    # render loop
    for i, frame in enumerate(seq):
        # non-starfish Blender stuff: e.g. setting file output paths
        bpy.data.scenes['Real'].node_tree.nodes['File Output'].file_slots[0].path = f
↪'real_{i}.png'
        bpy.data.scenes['Mask'].node_tree.nodes['File Output'].file_slots[0].path = f
↪'mask_{i}.png'

        # set up and render
        scene = bpy.data.scenes['Real']
        frame.setup(scene, bpy.data.objects['MyObject'],
                    bpy.data.objects['MyCamera'], bpy.data.objects['TheSun'])
```

(continues on next page)

(continued from previous page)

```
        bpy.ops.render.render(scene=scene)

        scene = bpy.data.scenes['Mask']
        frame.setup(scene, bpy.data.objects['MyObject'],
                    bpy.data.objects['MyCamera'], bpy.data.objects['TheSun'])
        bpy.ops.render.render(scene=scene)

        # postprocessing
        label_map = {'object': (255, 255, 255), 'background': (0, 0, 0)}
        clean_mask = normalize_mask_colors(f'mask_{i}.png', label_map.values())
        del label_map['background']
        bboxes = get_bounding_boxes_from_mask(clean_mask, label_map)
        centroids = get_centroids_from_mask(clean_mask, label_map)

        # add some extra metadata
        frame.timestamp = int(time.time() * 1000)
        frame.sequence_name = '1000 random poses'
        frame.tags = ['front_view', 'left_view', 'right_view']
        frame.bboxes = bboxes
        frame.centroids = centroids

        # save metadata to JSON
        with open(f'meta_{i}.json', 'w') as f:
            f.write(frame.dumps())
```

PYTHON MODULE INDEX

S

INDEX

Symbols

B

C

D

E

F

G

I

J

M

N

P

R

S

T

U