# Mitsuba3 Util
# User manual

Version 0.1.0

Andriani Stamou

July 11, 2024

# Contents

# 1 Introduction

Mitsuba3 util is an integration of Mitsuba3 and its purpose is to simplify the usage of Mitsuba3 and automate some of its functionality in a more user-friendly way. To use the util only a configuration `json` file needs to be created that allows the user to set his scene and camera based on several options. In the following sections, it is described in detail how such a configuration file can be created and its expected outcome. Also, in the final section, some advanced applications are presented, to demonstrate the usage range of this util.

# 2 Objects

To add objects to the scene the `objects` array must be filled. There are two options for adding objects to the scene.

- setting the parameters of each object separately by providing a path to the object file for the `filename` field and setting its material options
- setting the parameters of multiple objects at once (in case the objects you want to add to the scene share the same material) by providing a path to a folder with object files for the `filename` field. If a material that supports color is selected for the group of objects, a `.txt` file with RGB colors for each file can be optionally provided for the `colors_filename` field to set different colors for each object.

The objects are placed on the scene according to the coordinates set that is specified in the object file.

## 2.1 Types of supported objects

Along with the filename (or folder), the object (or group of objects) type must be also specified. The following types of objects can be selected by setting the `type` field:

- `obj`

  If the object file (or files inside the folder) are in `OBJ` format, `"type": "obj"` must be added to the object element.

- `ply`

  If the object file (or files inside the folder) are in `PLY` format, `"type": "ply"` must be added to the object element.

- `linearcurve` / `bsplinecurve`

  If the file is `txt` with curves formatted in the way mitsuba3 requires (see Curves), `"type": "linearcurve"` or `"type": "bsplinecurve"` must be added to the object element.

  The util also provides a `Python` and a `MATLAB` script for creating such curves files. More information on this functionality can be found on

Section 6.1.

## 2.2 Materials

All materials (BSDFs) that are documented here are supported. If the guidelines of mitsuba3 documentation are followed, all the required fields of the material can be placed inside the `material` field and the object will be rendered accordingly.

In advance, the util provides some materials and their basic properties for simplifying their use. These materials are described in the following subsections.

### 2.2.1 Glass

For rendering a glass object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "material": {
        "type": "glass",
        "color": [0.5, 0.27, 0.36]
    }
}
```

Listing 1: Glass util object

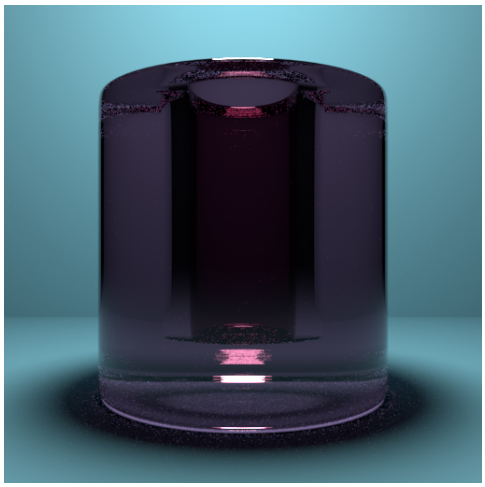
The `color` field is optional. When added a colored glass object will be created, otherwise a simple glass object.

When `glass` is selected as material `type`, the following mitsuba3 BSDF is created internally:

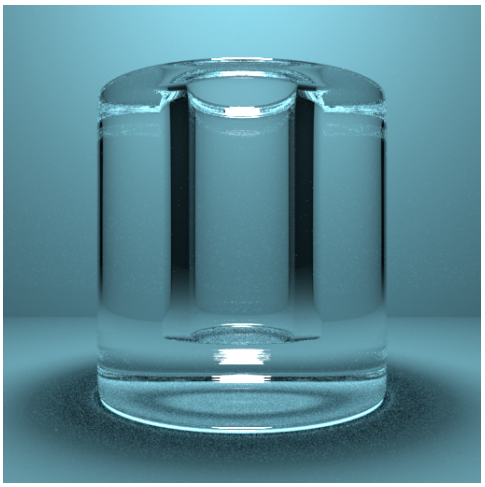
```
{
    "type": "dielectric",
    "int_ior": "bk7",
    "ext_ior": "air",
    "specular_transmittance": {
        "type": "RGB",
        "value": material_color
    }
}
```

Listing 2: Mitsuba3 BSDF when `"type":"glass"`

Note that `specular_transmittance` field will be present only in the case that `color` field is provided in the util json.



Colored glass – `color` field in `material` present



Glass – `color` field in `material` missing

Figure 1: Glass examples

### 2.2.2 Rough Glass

For rendering a rough glass object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "material": {
        "type": "roughglass",
        "alpha": 0.1,
        "color": [0.5, 0.27, 0.36]
    }
}
```

Listing 3: Rough glass util object

The `color` field is optional. When added a colored glass object will be created, otherwise a simple glass object.

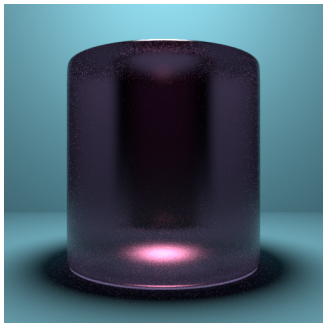
The `alpha` field determines the roughness of the glass material and is also optional. If not provided it takes a default value of 0.1.

When `roughglass` is selected as material `type`, the following mitsuba3 BSDF is created internally:

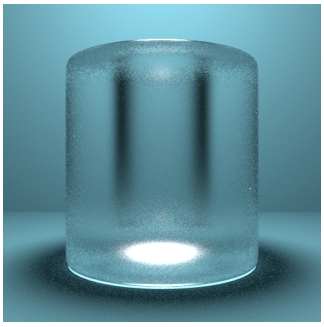
```
{
    "type": "roughdielectric",
    "distribution": "beckmann",
    "alpha": material_alpha,
    "int_ior": "bk7",
    "ext_ior": "air",
    "specular_transmittance": {
        "type": "RGB",
        "value": material_color
    }
}
```

Listing 4: Mitsuba3 BSDF when `"type":"roughglass"`

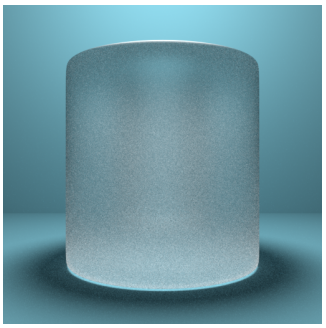
Note that `specular_transmittance` field will be present only in the case that `color` field is provided in the util json.



Colored glass – `color` field in `material` present and default `alpha`



Rough glass – `color` field in `material` missing and default `alpha` value



Rough Glass – `color` field in `material` missing and `alpha` value `0.5`

Figure 2: Rough glass examples

### 2.2.3 Thin Glass

For rendering a thin glass object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "material": {
        "type": "thinglass",
        "color": [0.5, 0.27, 0.36]
    }
}
```

Listing 5: Thin glass util object

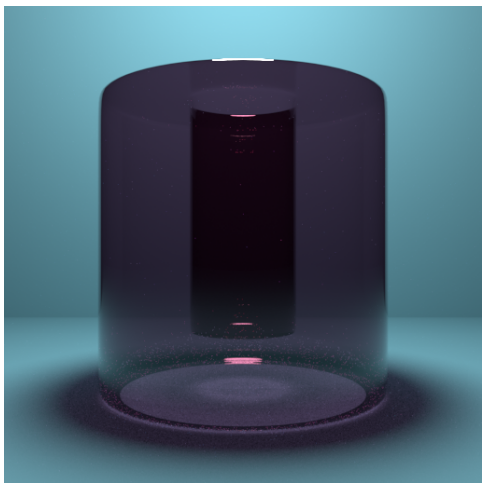
The `color` field is optional. When added a colored glass object will be created, otherwise a simple glass object.

When `thinglass` is selected as material `type`, the following mitsuba3 BSDF is created internally:

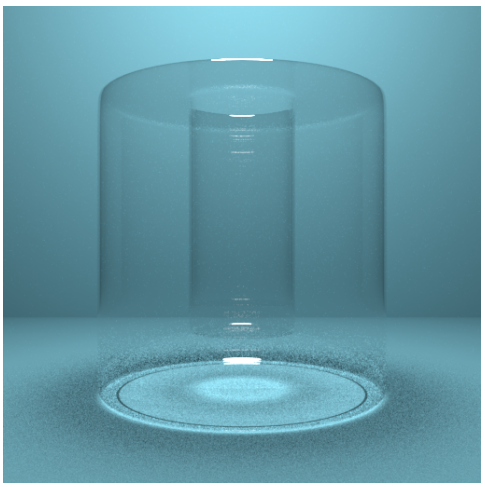
```
{
    "type": "thindielectric",
    "int_ior": "bk7",
    "ext_ior": "air",
    "specular_transmittance": {
        "type": "RGB",
        "value": material_color
    }
}
```

Listing 6: Mitsuba3 BSDF when `"type":"thinglass"`

Note that `specular_transmittance` field will be present only in the case that `color` field is provided in the util json.



Colored thin glass – `color` field in `material` present



Thin glass – `color` field in `material` missing

Figure 3: Thin glass examples

### 2.2.4 Plastic

For rendering a plastic object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "material": {
        "type": "plastic",
        "color": [0.5, 0.27, 0.36]
    }
}
```

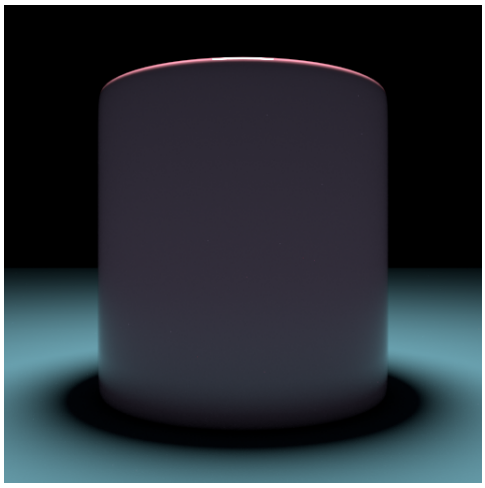
Listing 7: Plastic util object

The `color` field is optional. When added a colored plastic object will be created, otherwise a plastic object of a default color.

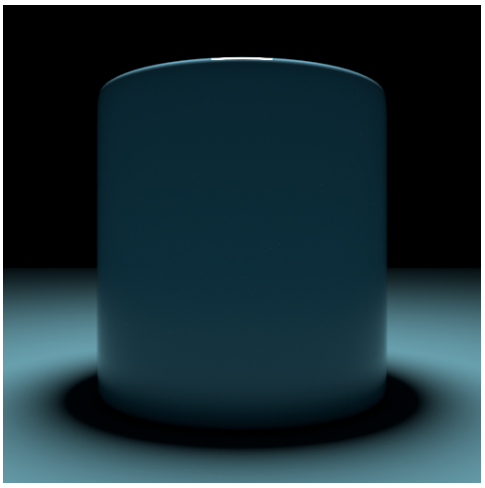
When `plastic` is selected as material `type`, the following mitsuba3 BSDF is created internally:

```
{
    "type": "plastic",
    "diffuse_reflectance": {
        "type": "RGB",
        "value": material_color
    }
}
```

Listing 8: Mitsuba3 BSDF when `"type":"plastic"`



Colored plastic – `color` field in `material` present



Plastic (default) – `color` field in `material` missing

Figure 4: Plastic examples

### 2.2.5 Rough Plastic

For rendering a plastic object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "material": {
        "type": "roughplastic",
        "alpha": 0.1,
        "color": [0.5, 0.27, 0.36]
    }
}
```

Listing 9: Rough plastic util object

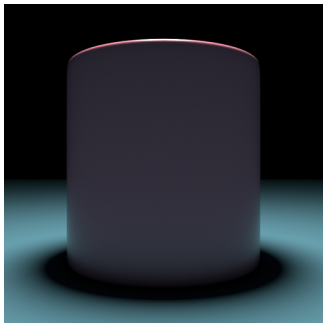
The `color` field is optional. When added a colored plastic object will be created, otherwise a plastic object of a default color.

The alpha field determines the roughness of the plastic material and is also optional. If not provided it takes a default value of 0.1.

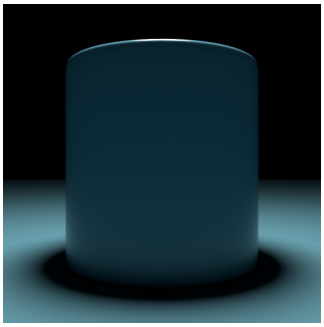
When `roughplastic` is selected as material `type`, the following mitsuba3 BSDF is created internally:

```
{
    "type": "roughplastic",
    "distribution": "beckmann",
    "alpha": material_alpha
    "diffuse_reflectance": {
        "type": "RGB",
        "value": material_color
    }
}
```

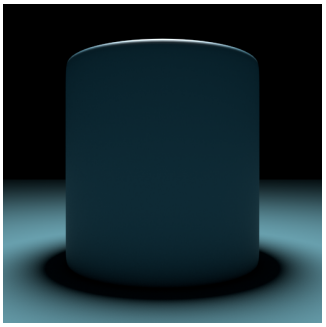
Listing 10: Mitsuba3 BSDF when `"type":"roughplastic"`



Colored rough plastic – `color` field in `material` present and default `alpha` value



Default rough plastic – `color` field in `material` missing and default `alpha` value



Default rough plastic – `color` field in `material` missing and `alpha` value `0.5`

Figure 5: Rough plastic examples

### 2.2.6 Metal

For rendering a metal object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "material": {
        "type": "metal",
        "metal_type": "Al"
    }
}
```

Listing 11: Metal util object

The `metal_type` field is optional. When added a metal object of the specified type will be created, otherwise a gold (Au) metal object.

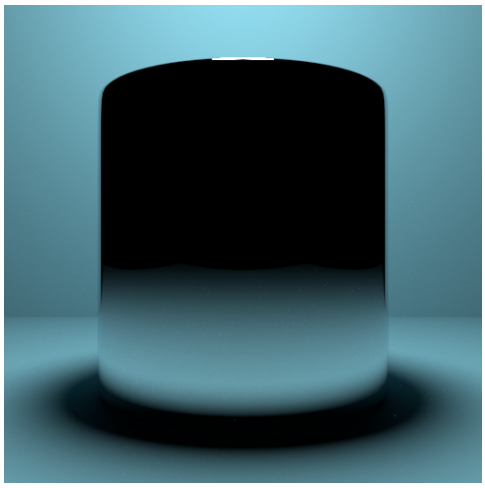
The following table lists all supported values for `material_type` that can be passed into the `metal` and `roughmetal` material types.

| Value | Description | Value | Description |
|---|---|---|---|
| a-C | Amorphous carbon | Na_palik | Sodium |
| Ag | Silver | Nb | Niobium |
| Al | Aluminium | Ni_palik | Nickel |
| AlAs | Cubic aluminium arsenide | Rh | Rhodium |
| AlSb | Cubic aluminium antimonide | Se | Selenium |
| Au | Gold | SiC | Hexagonal silicon carbide |
| Be | Polycrystalline beryllium | SnTe | Tin telluride |
| Cr | Chromium | Ta | Tantalum |
| CsI | Cubic caesium iodide | Te | Trigonal tellurium |
| Cu | Copper | ThF4 | Polycryst. thorium (IV) fluoride |
| Cu2O | Copper (I) oxide | TiC | Polycrystalline titanium carbide |
| CuO | Copper (II) oxide | TiN | Titanium nitride |
| d-C | Cubic diamond | TiO2 | Tetragonal titan. dioxide |
| Hg | Mercury | VC | Vanadium carbide |
| HgTe | Mercury telluride | V_palik | Vanadium |
| Ir | Iridium | VN | Vanadium nitride |
| K | Polycrystalline potassium | W | Tungsten |
| Li | Lithium | MgO | Magnesium oxide |
| Mo | Molybdenum | none | 100% reflecting mirror |

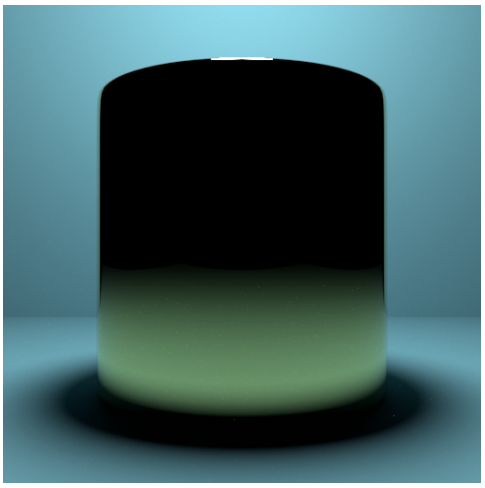
When `metal` is selected as material `type`, the following mitsuba3 BSDF is created internally:

```
{
    "type": "conductor",
    "material": material_metal_type
}
```

Listing 12: Mitsuba3 BSDF when `"type":"metal"`



Aluminium object – `metal_type` field in `material` `Al`



Metal (default) – `metal_type` field in `material` missing

Figure 6: Metal examples

### 2.2.7 Rough Metal

For rendering a rough metal object from a file "object.obj" an array element as the one below must be added in `objects` array.

```
{
    "filename": "object.obj",
    "type": "obj",
    "alpha": 0.1,
    "material": {
        "type": "roughmetal",
        "metal_type": "Al"
    }
}
```

Listing 13: Metal util object

The `metal_type` field is optional. When added a rough metal object of the specified type will be created, otherwise a gold (Au) rough metal object.

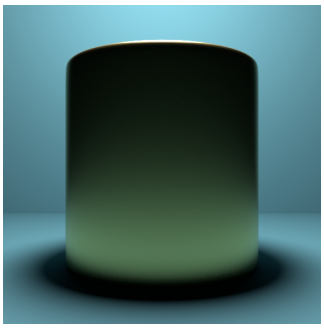
When `roughmetal` is selected as material `type`, the following mitsuba3 BSDF is created internally:

The `alpha` field determines the roughness of the metal material and is also optional. If not provided it takes a default value of `0.1`.

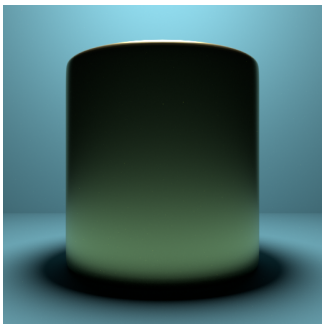
```
{
    "type": "roughconductor",
    "material": material_metal_type,
    "distribution": "ggx",
    "alpha_u": material_alpha,
    "alpha_v": material_alpha
}
```

Listing 14: Mitsuba3 BSDF when `"type":"roughmetal"`

Rough aluminium object – `metal_type` field in `material` `Al` and default `alpha` value



Rough metal (default) – `metal_type` field in `material` missing and default `alpha` value



Rough metal (default) – `metal_type` field in `material` missing and `alpha` value `0.5`

Figure 7: Rough metal examples

## 2.3 Other examples

### 2.3.1 Rendering an object in `.ply` format

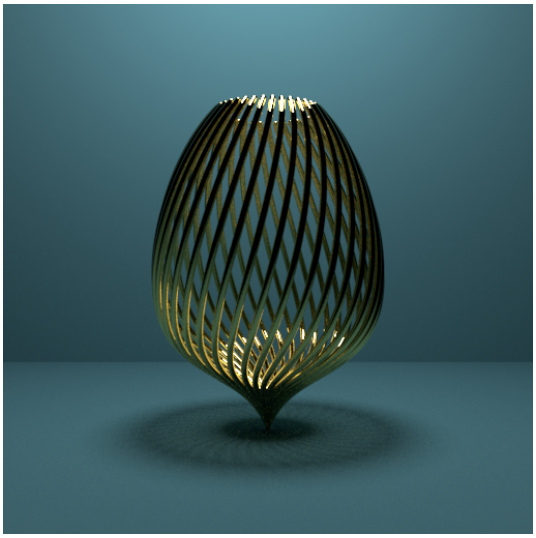
Adding the following object inside the `objects` array, a glass objects saved in a `object.ply` file can be rendered.

```
{
    "filename": "object.ply",
    "type": "ply",
    "material": {
        "type": "glass"
    }
}
```

### 2.3.2 Rendering `linearcurve` and `bsplinecurve` shapes from file

If a `curves.txt` containing control points for curves (see Curves for more information on such files) is available, the following object can be added in `objects` array to render them.

```
{
    "filename": "curves.txt",
    "type": "linearcurve", //or "bsplinecurve"
    "material": {
        "type": "metal"
    }
}
```



Examples curve files can be found in `mitsuba3-util/canning/curves_files`. Also, the mitsuba3 util contains `MATLAB` and `Python` scripts for automatically creating curve files. See Section 6.1 for more details.

### 2.3.3 Rendering multiple objects of different materials in the same scene

`objects` array can contain more than one object element at once to add in the scene. For example, adding the following objects in `objects` will result in adding curves from both files in the scene rendered as the materials specified for each one.

```
{
  "filename": "curves1.txt",
  "type": "bsplinecurve",
  "material": {
    "type": "metal"
  }
},
{
  "filename": "curves2.txt",
  "type": "bsplinecurve",
  "material": {
    "type": "glass",
    "color": [0.8, 0.2, 0.5]
  }
}
```

The same structure would work for `obj` and `ply` files.

**Note:** the coordinates of the objects can't be modified. They must be set correctly in their files.

### 2.3.4 Rendering multiple objects of same material in the same scene

Suppose you have multiple objects in multiple files (of the **same format**) and you want these objects to share the same material. In that case, you can place the files in a folder (**only object files must be inside the folder**) and enter the path in the `filename` field in the object element. Also, the files must follow this naming convention: `001.obj, 002.obj, 003.obj`, etc. (or `.ply` or `.txt` for curves).

In advance, in this case, a `colors_filename` field is supported that can take a `colors.txt` file that includes RGB values for each object in the folder. The file must contain the same number of lines as the objects in the folder and each line must contain 3 space-separated values. Also, the material selected must be one of the mitsuba3 util materials described above for the `colors_filename` field to be supported.

```
{
    "filename": "objects_folder/",
    "colors_filename": "colors.txt",
    "type": "obj",
    "material": {
        "type": "glass"
    }
}
```

Please note that this functionality is **not available** when `output_type:"animation_video"`.

### 2.3.5 Setting the `objects` for an animation video

To create an animation video your configuration file in `objects` field must contain at least one element that its `filename` is a path to a folder containing multiple object files ( `.obj` or `.ply` ) of the same object at different positions. The resulting animation video will have as many frames as the provided positions of the object.

You can still add multiple objects, as described in Section 2.3.3.

For example, having the following `objects` field in your configuration file:

```
"objects": [
    {
      "name": "material",
      "filename": "animation_files/01_cutting/3d_out/material/",
      "type":"obj",
      "material": {
        "type": "metal"
      }
    },
    {
      "name": "tool",
      "filename": "animation_files/01_cutting/3d_out/tool/",
      "type":"obj",
      "material": {
        "type": "roughglass"
      }
    },
    {
      "name": "scene",
      "filename": "animation_files/01_cutting/3d_out/scene/floor.obj",
      "type":"obj",
      "material": {
        "type": "plastic"
      }
    }
]
```

will result in an animation video where in each frame there is an object `animation_files/01_cutting/3d_out/scene/floor.obj` at the same position. The material and tool objects will be in different positions according to the frame.

# 3 Lighting

To add lights to the scene the `lights` array must be filled. Two lighting options/types are available, `area` and `envmap`, and are described in detail in the following sections.

**Note:** The two types can be combined by adding multiple elements in `lights` array.

## 3.1 Add lighting using an environment map

To use an environment map `my_map.hdr` for lighting your scene, an object of the following format can be added in the `lights` array:

```
{
  "emitter_type": "envmap",
  "filename": "my_map.hdr",
  "rotation_axis": [1,0,0],
  "rotation_degrees": 90,
  "scale_factor": 0.5
}
```

If you want to rotate your environment map, the optional fields `rotation_axis` and `rotation_degrees` can be used. If these fields are missing, the original orientation of your map will be preserved.

`rotation_axis:[1,0,0]`, `rotation_degrees:180`

`rotation_axis:[1,0,0]`, `rotation_degrees:0`

Figure 12: Envmap rotation examples

The optional field `scale_factor` is used for adjusting the radiance of the lighting in the scene. For example, to double the radiance of the original map, a value of `2` must be used. The default value for this field is `1`.

`scale_factor:4`

`scale_factor:0.5`

Figure 13: Envmap radiance examples

## 3.2 Add lighting using an emitter object

The second lighting option is using a sphere or rectangle object as an emitter. To add one of those types of emitters to the scene, an object of the following format can be added in the `lights` array:

```
{
  "emitter_type": "area",
  "emitter_shape": "sphere",
  "position": "top-center",
  "rotation_axis": [0,1,0],
  "rotation_degrees": -110,
  "size": "small",
  "distance_from_object": "small",
  "radiance": 10
}
```

The `emitter_shape` field sets the shape of the light and it takes the following two values: `sphere`, `rectangle`.

The `position` field sets the position of the light in the scene. Some preset positions are available for the sphere and rectangle lights to be easily positioned on the scene according to the center of the bounding box containing all the objects of the scene.

For the `rectangle` light, the `bottom-center-back` value can be used to place the rectangle light as a background for the object/objects.

For the `sphere` light, the following preset positions are available: `top-center`, `top-left`, `top-right`, `bottom-left`, `bottom-right`, `top-center-front`, `top-left-front`, `top-right-front`, `bottom-center-front`, `bottom-left-front`, `bottom-right-front`, `top-center-back`, `top-left-back`, `top-right-back`, `bottom-center-back`, `bottom-left-back`, `bottom-right-back`.

If none of the preset positions are used, the `position` field can take 3d coordinates (e.g. `"position": [0,0,0]`).

The `rotation_axis` and `rotation_degrees` can be used to rotate the

light on `rotation_axis` axis `rotation_degrees` degrees. Note that these fields are relevant only if 3d coordinates are provided for `position` field. Otherwise, they are ignored.

The `size` field sets the size of the light. It can take a scale vector for the light object to be resized or one of the preset sizes: `small`, `medium`, `large`. For these preset values, the scale vector is computed internally according to the size of the bounding box.

The `distance_from_object` sets the light's distance from the bounding box containing all the objects added to the scene. This field is ignored if `position` field contains 3d coordinates. The field can take one of the preset values `small`, `medium`, `large`, or a vector representing the distance from the object/objects for each axis. (e.g. `"distance_from_object": [0,10,0]`). For these preset values, the scale vector is computed internally according to the size of the bounding box.

Finally, `radiance` sets the light's radiance and it can take any real number as its value.

# 4 Output

The fields that can modify the output are placed inside `output` object of the configuration file and are displayed in Listing **??**.

```
{
    "use_gpu": ...,
    "output": {
        "type": "image",
        "results_folder": "",
        "rotation_degrees": 0,
        "rotation_step": 1,
        "width": 512,
        "height": 512,
        "fps": 5,
        "target": "auto",
        "distance": "auto",
        "seed": 0,
        "fov": 45,
        "samples_per_pixel": 1024,
        "up_axis": [0,0,1],
        "camera_axis": [0,1,0],
        "rotation_axis": [0,0,1],
        "add_floor": true,
        "floor_type": "diffuse",
        "floor_color": [0.1, 0.25, 0.3],
        "add_background": true,
        "background_type": "diffuse",
        "background_color": [0.1, 0.25, 0.3]
    },
    "objects": ...,
    "lights": ...
}
```

In the following sections, each non-self-explanatory field of `output` is explained.

## 4.1 Adding helper objects in the scene

From `output` field, the mitsuba3 util supports automatically adding floor or a background wall for your object.

### 4.1.1 Adding floor

The relevant `output` fields for adding floor to your scene are:

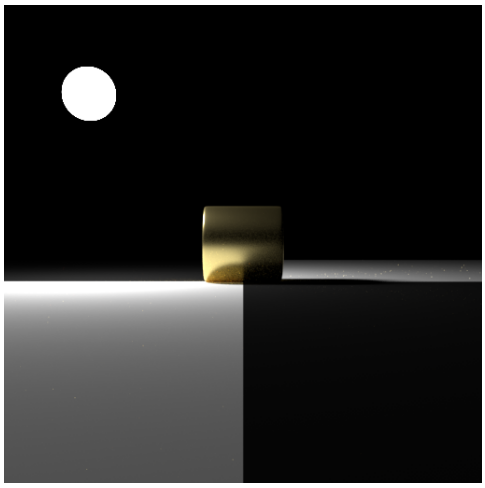
```
"output": {
    "add_floor": true,
    "floor_type": "diffuse",
    "floor_color": [0.1, 0.25, 0.3],
}
```

By setting the above fields, the util will automatically place a rectangle object as the floor. The position of the floor is computed based on your camera position, the axis that is considered the up axis, and the center of the bounding box that contains all the objects that you added to the scene. Note that by rotating the camera, the floor position will not be affected.

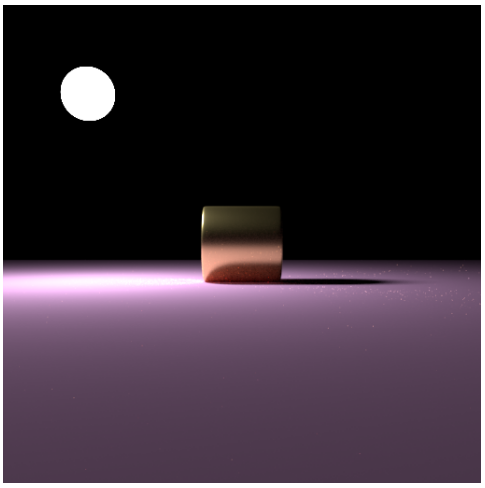
The `add_floor` field is optional and its default value is `True`. Meaning that if you don't explicitly set it as `False`, a floor object will be added to your scene.

The `floor_type` can either be `diffuse` or `checkerboard`.

If the `floor_type` is `diffuse`, you can set its color by setting the `floor_color` field with an RGB value.



Automatic checkerboard floor



Automatic colored floor

Figure 14: Floor examples

### 4.1.2 Adding background wall

The relevant `output` fields for adding a background wall to your scene are:

```
"output": {
    "add_background": true,
    "background_type": "diffuse",
    "background_color": [0.1, 0.25, 0.3],
}
```

By setting the above fields, the util will automatically place a rectangle object as the background wall. The position of the background is computed based on your camera position, the axis that is considered the up axis, and the center of the bounding box that contains all the objects that you added to the scene. Note that by rotating the camera, the background wall position will not be affected.

The `add_background` field is optional and its default value is `False`. This means that if you don't explicitly set it as `True`, a background wall will not be added to your scene.

The `background_type` can either be `diffuse` or `checkerboard`.

If the `background_type` is `diffuse`, you can set its color by setting the `background_color` field with an RGB value.

## 4.2 Setting the camera

### 4.2.1 Camera position

The relevant `output` fields for positioning the camera to your preference are the following:

```
"output": {
    "target": [10,10,5],
    "distance": "auto",
    "up_axis": [0,1,0],
    "camera_axis": [1,0,0]
}
```

By setting the above fields you are positioning your camera on the `camera_axis` at distance `distance` from `target` and looking at `target` with up axis `up_axis`.

If `camera_axis` and `up_axis` are not set they take their default values: `camera_axis: [0,1,0]` `up_axis:[0,0,1]`

Not including the `target` and `distance` fields in `output` or setting them to `auto`, results in being computed automatically. In that case, `target` will be the center of the bounding box containing all the objects added in the scene and `distance` will be double the maximum size among the 3 coordinates of the bounding box.

### 4.2.2 Camera rotation

The relevant `output` fields for rotating the camera are:

`rotation_axis` is used to specify the axis around which the camera will rotate.

```
"output": {
    "rotation_degrees": 0,
    "rotation_step": 1,
    "rotation_axis": [1,0,0]
}
```

`rotation_degrees` when `image` or `animation_video` output type is selected is used to specify the angle of the object that the camera will look at.

When `rotation_video` output type is selected it is used to define the total degrees of the rotation that will be performed (e.g. if `rotation_degrees = 360` the resulting rotation video will be a full rotation around the object).

`rotation_step` is relevant only when `rotation_video` output type is selected and is used to specify the number of frames that will be produced for the video (i.e. if `rotation_degrees = 360` and `rotation_step = 1`, 360 frames will be produced for the rotation video).

Some examples of the above fields (and also `up_axis_field`) in the context of image output are presented in Figure **??**:

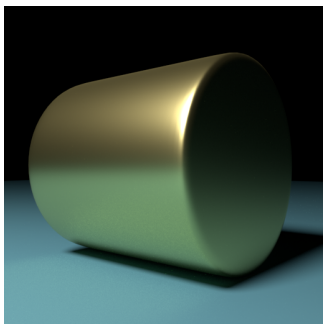
## 4.3 Setting options for the output file

### 4.3.1 Output type

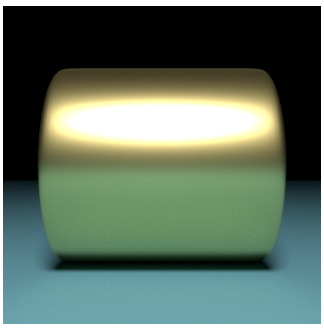
Mitsuba3 Util supports 3 types of outputs:

- Image (.png)
- Rotation Video (.avi)
- Animation Video (.avi)

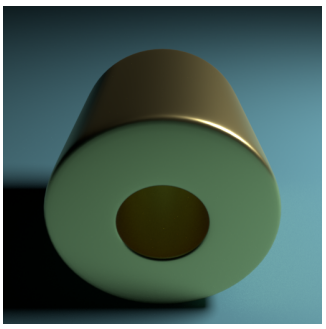
and can be set through `type` field by selecting one of the following values: `image`, `rotation_video`, `animation_video`.



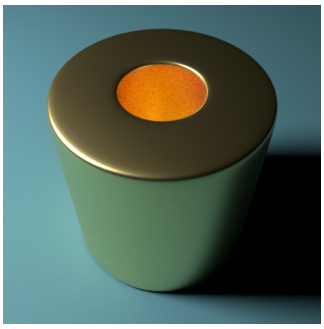
"rotation_axis": [0,1,0], "rotation_degrees": 130, "up_axis": [0,1,0]



"rotation_axis": [0,1,0], "rotation_degrees": 90, "up_axis": [0,1,0]



"rotation_axis": [1,0,0], "rotation_degrees": -30, "up_axis": [0,1,0]



(a) "rotation_axis": [1,0,0], "rotation_degrees": 50, "up_axis": [0,0,1]

Figure 15: Axes and rotations examples

### 4.3.2 Output size

The relevant `output` fields for defining the output size are:

```
"output": {
    "width": 512,
    "height": 512
}
```

For example, the images or frames will be 512x512 pixels using the above values.

### 4.3.3 Output quality

The relevant `output` fields for defining the output quality are:

```
"output": {
    "samples_per_pixel": 224,
    "seed": 0
}
```

The field `samples_per_pixel` determines how many samples will be (randomly) taken for each pixel. The higher the value of this field is, the better the quality of the image.

The field `seed` field is to set the seed for the random sampling. This can be useful when you need to produce many images and combine them to get a better quality result if rendering the image at once exceeds the limits of your resources. Different samples will be used each time by changing the `seed` value for each image.

### 4.3.4 Output file location

The relevant `output` field for choosing where the result file will be saved is:

```
"output": {
    "results_folder": "path/to/results/folder"
}
```

This field is optional and if it is not included the result file will be saved in the current directory the util runs.

# 5 Running mitsuba3 Util on CPU/GPU

Mitsuba3 provides a set of different system “variants” that change aspects of the simulation — including different computational backends (e.g., GPU, CPU).

Each variant follows the naming convention presented in Figure 16.

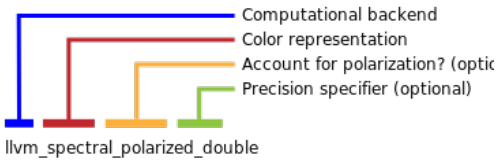



Figure 16: Variants’ naming

Mitsuba3 Util supports two such variants, one for computation on the CPU and one for the GPU. Both variants belong to the subset of variants installed when installing Mitsuba3 on your system with `pip`.

More details on Mitsuba3 variants can be found in Mitsuba’s documentation: Choosing variants

## 5.1 Running on CPU

By setting the json field of the configuration file `use_gpu` to `false` the computations are performed on CPU and `llvm_ad_rgb` variant is used.

```
{
    "use_gpu": false,
    "output": ...,
    "objects": ...,
    "lights": ...
}
```

## 5.2 Running on GPU

By setting the json field of the configuration file `use_gpu` to `true` the computations are performed on GPU and `cuda_ad_rgb` variant is used.

```
{
    "use_gpu": true,
    "output": ...,
    "objects": ...,
    "lights": ...
}
```

**Note:** Mitsuba3 requires `NVidia driver >= 495.89` for computation on the GPU

# 6 Applications

## 6.1 Curves files creation

Mitsuba3 util provides two ways of creating curve files automatically.

### 6.1.1 Cylinder orderings

In folder `cylinder_orderings` a `Python` script `generate_control_points.py` is located that automatically creates a curve file `curves.txt`.

To create the curves of your preference you can edit the variables on the top of the file.

- `r` variables: `r1`, `r2`, etc represent the radius of the curve at each bending level/control point.
- `level_dist` list: It must be the same size as the number of `r` variables and represent the height of each control point.
- `m` variable: Represents the number of cylinders your ordering will have. These cylinders will be placed around a circle.

The resulting file will be saved inside `cylinder_orderings` folder. In `cylinder_orderingscurves_f` some example files can be found.

### 6.1.2 Caning

The second option provided creates curve files inspired by the caning technique.

In `caning` folder a `MATLAB` script named `runme.m` can be found that creates two curve files `curves_twist_1.txt` and `curves_twist_2.txt` that each one contains every second cylinder/cane so that consecutive canes can be rendered as different materials when used from general util.

To modify the canes produced by this script you can change the variables on the top of the script:

- `ncanes` variable: Sets the number of canes
- `x_offset` variable: This is used to augment the radius of the final caning object. By setting it, the points in `pts` will be moved on the x-axis at the selected value.

- `pts` array: Sets the points of the profile of the final object that will be created.

## 6.2 Filigree

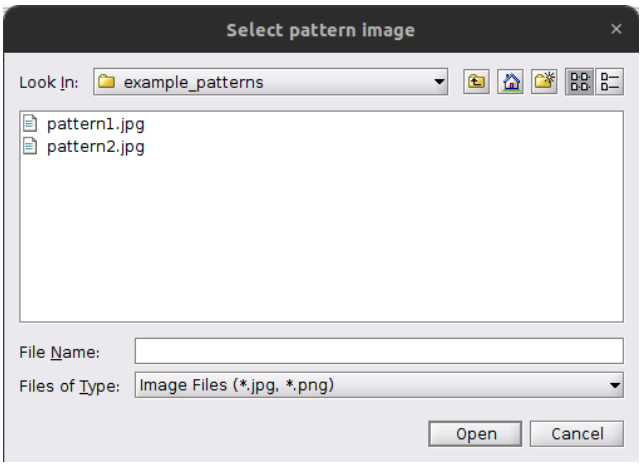
Mitsuba3 util provides a script to create a filigree object from an image and a modification of the general util to render it inside a lighted room. This application is inspired by the relief sculptural method.

### 6.2.1 Creating the filigree object file

In folder, `filigree` a `MATLAB` script `runme.m` can be found that creates a surface object with a pattern, provided by an image, engraved on it.

Running the script will result in the creation of an object `image_surface.obj` the size of the pattern image with the pattern engraved on it. The `.obj` file is saved inside `filigree` folder.

Once you run the `MATLAB` script a dialog box will appear for selecting the desired pattern from your filesystem:




In folder `example_patterns` example images of patterns can be found.

### 6.2.2 Placing the object in the center of a room

To render the filigree object created by the `MATLAB` script in a room, from `filigree` folder you can run:

```
python3 filigree_room.py config.json
```

where `config.json` is a configuration file similar to the one used for the general util. For this configuration file, the following json fields are available:

```
{
  "use_gpu": true,
  "filigree_path": "image_surface.obj",
  "lights_radiance": 10,
  "output": {
    "fov": 45,
    "results_folder": "",
    "width": 512,
    "height": 512,
    "samples_per_pixel": 1024,
    "seed": 0
  }
}
```

Listing 15: Filigree configuration file

The purpose of `use_gpu` and the fields inside `output` is the same as described in the general util.

Additionally, a required `filigree_path` must be set with the path to the surface `.obj` file.

Also, an optional `lights_radiance` field is provided to set the radiance of the lights inside the room. If omitted, its default value is 10.

Note that this script is specifically for filigree objects created by the `MATLAB` script described in the previous section. It is mostly automated with a small subset of the general util's parameters being functional. If you need to render your own `.obj` filigree object or more freedom setting the scene, like adding additional objects or more lights, the general util must be used.

## 6.3 Stained glass

Mitsuba3 util provides scripts to create stained glass from an image. There are two options for creating the stained glass tiles: randomly segmenting the image in a specified number of tiles or segmenting the image based on the color regions it contains and creating the tiles accordingly. In addition to creating the tiles, the scripts automatically generate a skeleton for the stained glass arrangement.

Also, a modification of the general util is provided to render the stained glass in a room as a window.

In folder `example_images`, example images that can be used as input can be found.

In folder `example_files`, example files that the `MATLAB` scripts produced can be found.

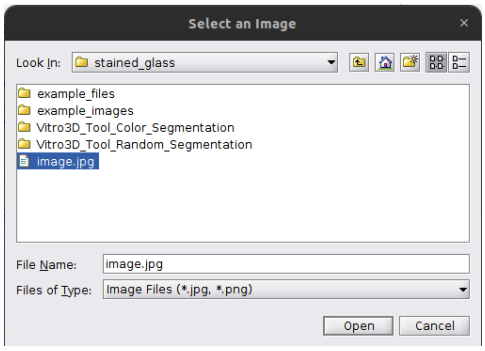
### 6.3.1 Creating the stained glass object files from an image

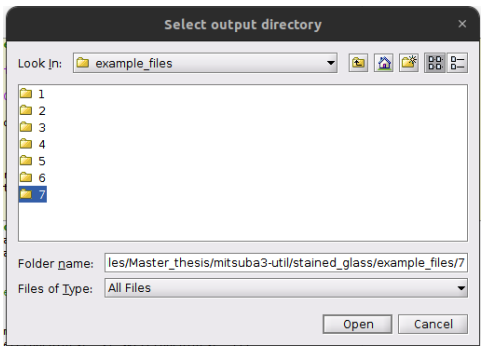
#### 6.3.1.1 Random segmentation

In folder `stained_glass/Vitro3D_Tool_Random_Segmentation` a `MATLAB` script `runme.m` can be found that creates the tiles and skeleton of stained glass based on an image.

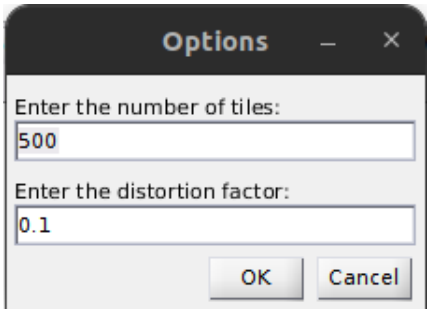
Running the script will result in the creation of the skeleton `skeleton.obj`, tiles `.obj` files in folder `tiles`, and a `colors.txt` file containing the RGB value of each tile, obtained from the image provided.

Once you run the `MATLAB` script two dialog boxes will appear: one for selecting an input image and one for selecting the output folder. The script will place the generated files for the input image inside the selected output folder:



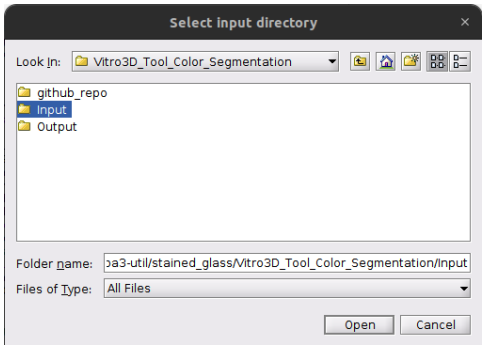
Also, a dialog box will appear for setting the number of random tiles you want the script to generate and also the distortion factor for these tiles. Higher values for the distortion factor will result in rougher surfaces for the tiles.

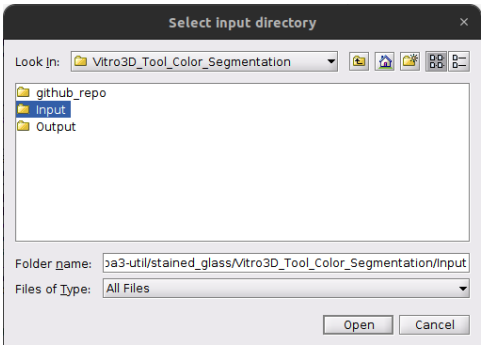



#### 6.3.1.2 Color segmentation

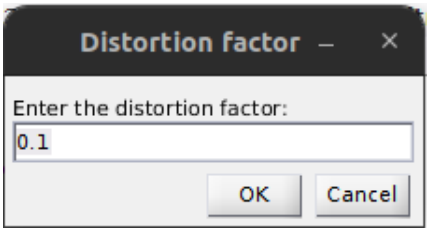
In folder `stained_glass/Vitro3D_Tool_Color_Segmentation` a `MATLAB` script `Vitro3D_Tool.m` can be found that creates the tiles and skeleton of stained glass based on the color regions of an image.

Once you run the `MATLAB` script two dialog boxes will appear: one for selecting an input folder and one for selecting the output folder. The script will create a sub-folder inside the selected output folder for each image file ( `.jpg` or `.png` ) found in the selected input folder. The generated files mentioned above will be placed inside this sub-folder.




Also, a dialog box will appear for setting the distortion factor for the tiles. Higher values for the distortion factor will result in rougher surfaces for the tiles.




#### 6.3.1.3 Placing the stained glass in a room as a window

To render the stained glass created by either of the two `MATLAB` scripts in a room as a window, from `stained_glass` folder you can run:

```
python3 stained_glass_room.py config.json
```

where `config.json` is a configuration file similar to the one used for the general util. For this configuration file, the following json fields are available:

```
{
  "use_gpu": true,
  "stained_glass_files": {
    "tiles_path": "example_files/1/tiles/",
    "skeleton_path": "example_files/1/skeleton.obj",
    "colors_path": "example_files/1/colors.txt"
  },
  "lights_radiance": 10,
  "output": {
    "fov": 45,
    "results_folder": "",
    "width": 512,
    "height": 512,
    "samples_per_pixel": 224,
    "seed": 0
  }
}
```

Listing 16: Stained glass configuration file

The purpose of `use_gpu` and the fields inside `output` is the same as described in the general util.

Additionally, all the fields of `stained_glass_files` object must be set with the path to the files created by the `MATLAB` script for the stained glass.

Also, an optional `lights_radiance` field is provided to set the radiance of the lights inside the room. If omitted, its default value is 10.

Note that this script is specifically for stained glass files created by the `MATLAB` script described in the previous section. It is mostly automated with a small subset of the general util's parameters being functional. If you need more freedom setting the scene, like adding additional objects or more lights, the general util must be used.