Interdisciplinary Space Master

2023 – 2024

(Third Semester)

Scientific Space Project

Generation of Training Images for Lens Flare Removal

AUTHORS:
ATTILA NEMET

Table of Contents

Acronyms 5
1 Introduction 6
1.1 Introduction 6
1.2 Literature Review 6
2 Lens Flare 9
3 Preparing the Software Environment 11
3.1 Installing Blender 3.6 11
3.2 Blender's Bundled Python 13
3.3 Importing Custom Modules into Blender's Python 13
3.4 Blender as a Module 14
3.5 Debugging in Blender's Python 15
3.6 Installing Flares Wizard 16
4 The bpy module 17
5 Setting up the Scene 17
6 Properties of the Flares Wizard 25
7 The program package 29
7.1 Setting up and saving an LF effect with `lf_setup.py` 29
7.2 Mass production of LF effects with `lf_gen.py` 30
7.3 The helper functions in `utils.py` 31
7.3.1 The `get_args()` function 31
7.3.2 The `save()` function 31
7.3.3 The LF class 32
7.3.4 The `rand_lf_origin()` function 35
8 Using the LFW package 35
8.1 Performance 36
9 Postprocessing of LF-only images 38
10 Further improvement possibilities 42
11 Conclusion 43
12 References 44

Abstract

Title: Generation of Training Images for Lens Flare Removal

In the field of Computer Vision (CV), the presence of lens flare (LF) artifacts poses a significant challenge, often degrading the performance of CV models.
This paper introduces the Lens Flare Wizard (LFW), a Python-based toolkit designed to systematically generate and apply realistic lens flare effects on images. The primary aim of the LFW is to enrich training datasets, enabling CV models to better recognize and remove LF artifacts, thereby improving their robustness and effectiveness in real-world applications. Developed using Python and Blender's Flares Wizard add-on, the LFW automates the creation of image pairs—original and lens flare-augmented—for training and testing CV models. Our findings indicate that LFW can efficiently generate lens flare effects, with processing times ranging from 0.2 to 0.8 seconds per image, depending on image resolution and flare complexity. The implications of this work are significant, offering a practical solution for researchers and practitioners in improving CV model performance under challenging visual conditions.

Acknowledgements

I would like to acknowledge and give my warmest thanks to my supervisor Dave van der Meer who made this work possible. His guidance and advice carried me through all the stages of writing my project. I would also like to thank my committee members for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions.

I would also like to give special thanks to my wife Crista and my family as a whole for their continuous support and understanding when undertaking my research and writing my project. Your prayer for me was what sustained me this far.

Acronyms

Parameter	Description
CV	Computer Vision
LF	Lens Flare
LFW	Lens Flare Wizard
SLAM	Simultaneous Location and Mapping
LOD	Limits of Detection
GPU	Graphical Processor Unit
GUI	Graphical User Interface
JSON	Java Script Object Notation
OS	Operating System
RGB	Red, Green, Blue
HSV	Hue, Saturation, Value
SSIM	Structural Similarity Index

1 Introduction

1.1 Introduction

In the realm of Computer Vision (CV), the accuracy and reliability of models are paramount, particularly in applications where precision is crucial, such as in Simultaneous Localization and Mapping (SLAM) algorithms. These algorithms, pivotal in the fields of robotics and autonomous navigation, rely heavily on the integrity of visual input. However, the presence of lens flare artifacts in images can significantly impede their performance by introducing non-existent, false features into the scene, leading to erroneous interpretations and decisions.

The motivation behind this research lies in addressing this challenge by developing a methodology to improve the robustness of SLAM algorithms against the misleading effects of lens flares. By training CV models to identify and mitigate lens flare artifacts, we aim to enhance the accuracy of SLAM algorithms, ensuring more reliable navigation and decision-making in autonomous systems.

To achieve this, we introduced the Lens Flare Wizard (LFW), a Python-based toolkit designed to generate and overlay realistic lens flare effects on images. This toolkit serves a dual purpose: first, to enrich training datasets by introducing a diverse range of lens flare scenarios, and second, to enable the subsequent training of CV models to recognize and disregard these artificial anomalies.

In selecting an appropriate platform for developing the LFW toolkit, several options were considered, including Unity, Unreal, and Blender. Blender emerged as the preferred choice, primarily due to its Python programming interface. This interface facilitates significantly faster development and iteration cycles compared to the more time-consuming C# and C++ languages, which are commonly used in the other platforms. The decision to use Blender aligns with the project's goal of rapid and iterative development, allowing for quick prototyping and testing of the lens flare effects and their impact on CV models.

This paper details the development process of the LFW toolkit, the methodology for applying the generated lens flare effects to training datasets.

1.2 Literature Review

In constructing a comprehensive literature review on producing training sets for computer vision, it's pertinent to explore various methodologies and advancements in the field, especially focusing on the simulation of lens flare effects and their integration into image datasets. A significant contribution to the computer vision domain was made by Fei-Fei Li through the creation of ImageNet, which catalyzed the deep neural network revolution in 2012. ImageNet offered an extensive dataset that significantly enhanced the training capabilities of computer vision models, setting a precedent for the importance of robust and diverse training sets.

The paper "Physically-Based Real-Time Lens Flare Rendering" by (Hullin, Eisemann, Seidel, & Lee, 2011) delves into the complexities of simulating lens flare effects, traditionally considered a challenging aspect due to the intricate light interactions within a lens system. Traditionally viewed as a degrading artifact, lens flare has been recognized for its artistic value and its role in enhancing realism and perceived brightness in imagery. The paper introduces an innovative method for interactively computing physically-plausible lens flare renderings, covering various components essential for realism, such as imperfections, chromatic and geometric lens aberrations, and anti-reflective lens coatings.

Prior to this, interactive techniques relied heavily on approximations, and more accurate simulations were reserved for specific applications, necessitating costly offline processes. The paper highlights the attractiveness of simulating lens flare, as it offers a perceptual cue for humans to interpret the presence of extreme brightness, a feature often utilized in movies and games to enhance realism.

The authors present a rendering scheme that operates interactively to real-time performance, addressing both the simulation of intricate lens flares and various acceleration strategies to balance performance and quality. The underlying model of the optical system is detailed, discussing aspects such as geometry, light propagation, aperture, optical media, dispersion, and the crucial Fresnel equations for reflection and transmission.

Notably, the paper discusses the simulation of diffraction effects, particularly the Fraunhofer and Fresnel approximations, which are pivotal in replicating the intricate patterns observed in real lens flares. The rendering system is meticulously elaborated, outlining the processes of ghost enumeration, ray tracing, rasterization, and shading to achieve the desired lens flare effects.

Furthermore, the paper addresses various strategies to enhance the efficiency and quality of the rendering, such as ray bounding, adaptive resolution, intensity LOD, aperture culling, symmetries in the optical system, and spectral rendering. The GPU implementation of the algorithm is also detailed, ensuring the practical applicability of the technique in real-time applications.

In addition to the technical aspects, the paper explores artistic control, allowing for creative manipulation of optical elements and the addition of realism through imperfections, thus offering a balance between technical accuracy and artistic expression.

The results section affirms the effectiveness of the proposed method, highlighting its suitability for various applications, including computer games, image and video processing, and lens-system design. It also acknowledges the limitations of the approach, particularly in handling light sources and the need for precomputing resolution for the ray bounding precomputation.

The paper "Practical Real-Time Lens-Flare Rendering" (Lee & Eisemann, 2013) introduces a real-time approach for rendering lens flare effects, employing a first-order approximation of ray transfer in an optical system, allowing lens flare-producing light rays to map directly to the sensor. This method is faster and produces physically-plausible images at high framerates on standard graphics hardware.

Recognizing the limitations of previous methods that relied on costly ray tracing or complex polynomial expressions, this paper presents a more streamlined approach. The proposed method is based on a first-order approximation of the ray transfer in an optical system, which significantly simplifies the process by enabling the direct mapping of lens flare-producing light rays to the sensor. This innovation not only makes the approach easy to implement but also ensures that it can produce physically-plausible images at high frame rates on standard off-the-shelf graphics hardware. The resulting technique addresses the need for a balance between realism and computational efficiency, making it a valuable contribution to real-time graphics applications where lens flare effects enhance visual realism and depth without compromising performance.

The paper "Physically based lens flare rendering in 'The Lego Movie 2'" (Pekkarinen & Balzer, 2019) presents an approach for incorporating realistic lens flare rendering in production renderers. The paper describes the approximations, sampling techniques, and artist controls used for efficient lens flare rendering, with "The Lego Movie 2: The Second Part" as a case study.

This work builds upon previously established physically based lens simulation techniques and introduces a series of approximations and sampling methods that streamline the rendering process, making it efficient for use in production environments.

The authors focus on achieving a balance between physical accuracy and computational efficiency, providing flexible controls and workflows that allow artists to manipulate lens flare effects creatively. The techniques developed and demonstrated in this paper showcase the practicality and effectiveness of the approach, contributing to the visually stunning and artistically rich imagery of the movie.

The paper "Lens flare prediction based on measurements with real-time visualization" (Walch, 2018) presents a workflow for generating physically plausible renderings of lens flare phenomena by analyzing lens flares captured with a camera, allowing for predicting lens flares for given light setups.

The authors present a novel workflow for producing physically plausible renderings of lens flare effects by analyzing real lens flares captured with a camera. This process involves identifying and measuring key parameters of the lens system, with a particular focus on aspects such as anti-reflection coatings, which are critical to the appearance of lens flares. By accurately modeling these elements, the workflow allows for the prediction of lens flares under various lighting setups, which is especially useful in planning stages to avoid undesirable flare effects or to enhance visual storytelling.

Crucially, the proposed method is designed for real-time applications, leveraging a tightly controlled parameter set and GPU-based rendering. This enables immediate feedback and efficient integration into visual projects, providing a valuable tool for artists and engineers aiming to manage lens flare in a dynamic, creative environment.

2 Lens Flare

Lens flare is a phenomenon in photography and videography where light is scattered or flared in a lens system, often producing a phenomenon of one or more circles of light or hexagons of light, as well as other shapes. This scattering of light is caused by imperfections in the lens system, where incoming light is refracted, reflected, or scattered by material imperfections, dust, or scratches on the lens surfaces. While sometimes used deliberately for artistic effect, lens flare is generally regarded as undesirable because it degrades the quality of the image.

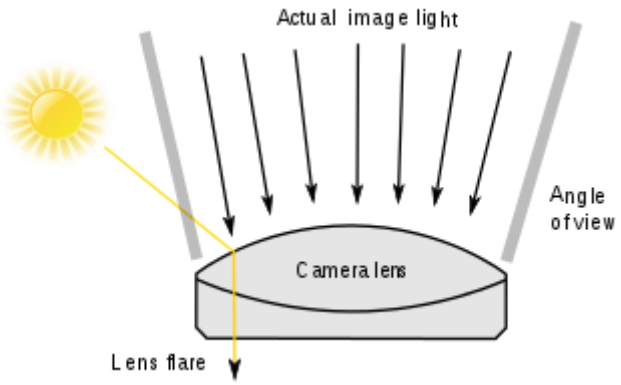

Figure 1 - Scheme of lens flare

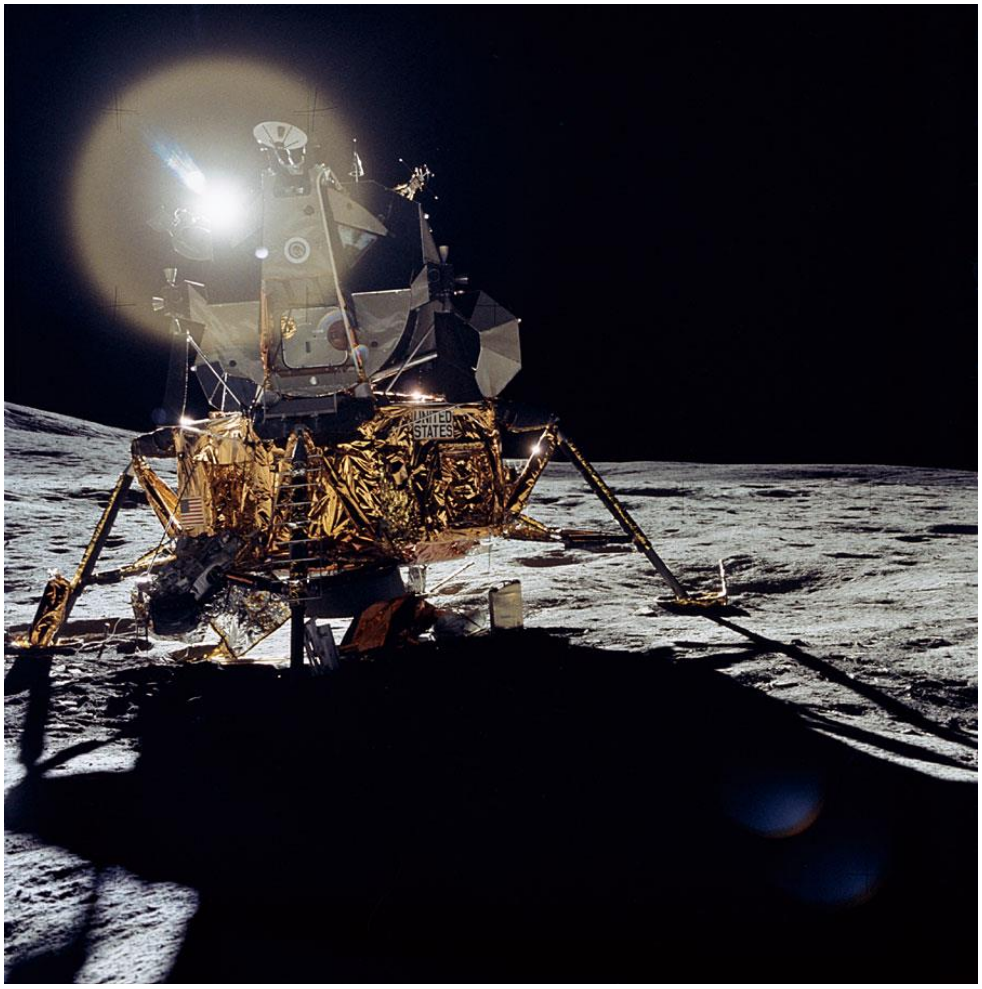

Figure 2 - Photograph of an Apollo Lunar Module containing lens flare.

In the context of computer vision and image and feature recognition, lens flare can be particularly problematic for several reasons:

- Lens flare introduces noise and distortion into an image, obscuring true features and details. This can lead to misinterpretation of the scene by computer vision algorithms, as the flare can mask important features or introduce artifacts that are mistaken for actual objects or shapes.
- Lens flare often reduces the contrast of the image by introducing a wash of light over it. This reduction in contrast can make it difficult for computer vision algorithms to detect edges and textures, which are critical for identifying objects and understanding the scene.
- The scattering of light can also alter the colors in an image, potentially misleading algorithms that rely on color for feature recognition or scene understanding.
- The appearance of lens flares is highly variable, depending on the angle and intensity of the incoming light, as well as the particular lens system. This inconsistency can be challenging for computer vision systems, which rely on stable and consistent features for accurate image recognition and interpretation.

Because of these issues, lens flare is generally considered an obstacle in computer vision and image processing. Mitigating or removing lens flares is often a necessary preprocessing step to improve the performance of computer vision algorithms in tasks like object detection, image classification, and scene reconstruction.

This study is centered around the generation of authentic-appearing lens flares, which are subsequently superimposed onto a pre-existing dataset of images. These modified images will serve as the training data for a computer vision (CV) model. The primary objective of the CV model is to identify and recognize the presence of lens flares, eliminate these lens flares, and try to restore the original image, trying to reproduce the image prior to the introduction of the lens flare.

3 Preparing the Software Environment

The scientific project has been executed on the windows platform (Win11 Pro). The Blender version 3.6.5 has been used.

3.1 Installing Blender 3.6

The results should be reproducible on different platforms. The corresponding Blender can be downloaded from the Blender 3.6 LTS site.

After installing Blender, it is recommended to add the path to blender.exe to the system path to be able to start blender from command line without prefixing the whole path. Starting Blender from command line has the following advantages:

- Command line arguments can be added to specify the python script to be executed or background (headless) execution.
- The script error traceback is printed in full to the terminal.
- Output of the Python `print()` commands can be observed.
- If script accidentally enters an infinite loop, it can be stopped with Ctrl-C

Adding the path to blender on Windows can be done by clicking the Start button or pressing the Windows key and in the search box start typing “edit the system environment variables”, click the best match from the Control panel categories:

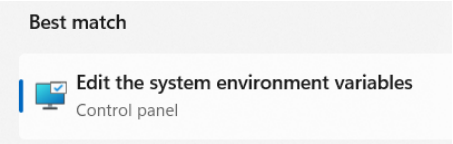

Figure 3 – Windows search result best match

on the Advanced tab click Environment Variables…

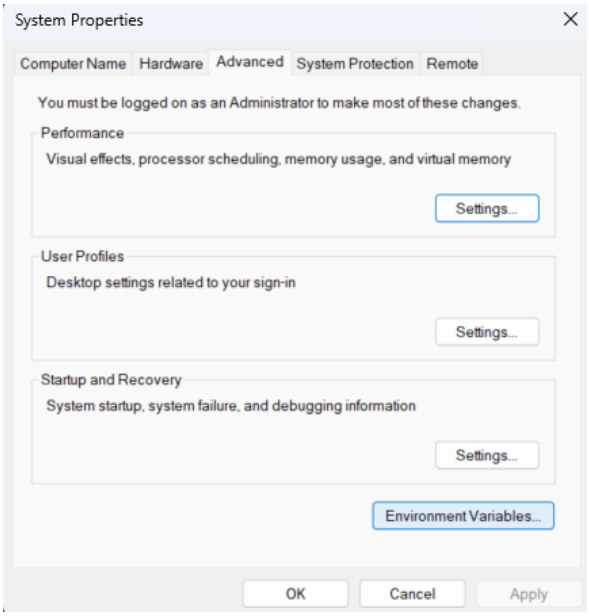

Figure 4 - Windows System Properties

Double-click the System variables Path variable:

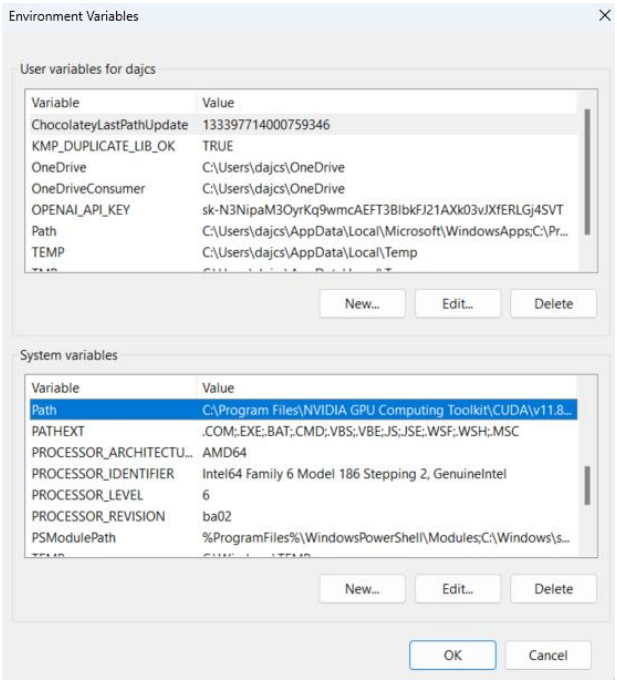

Figure 5 - Environment variables

Click New and enter the path: `c:\Program Files\Blender Foundation\Blender 3.6\`

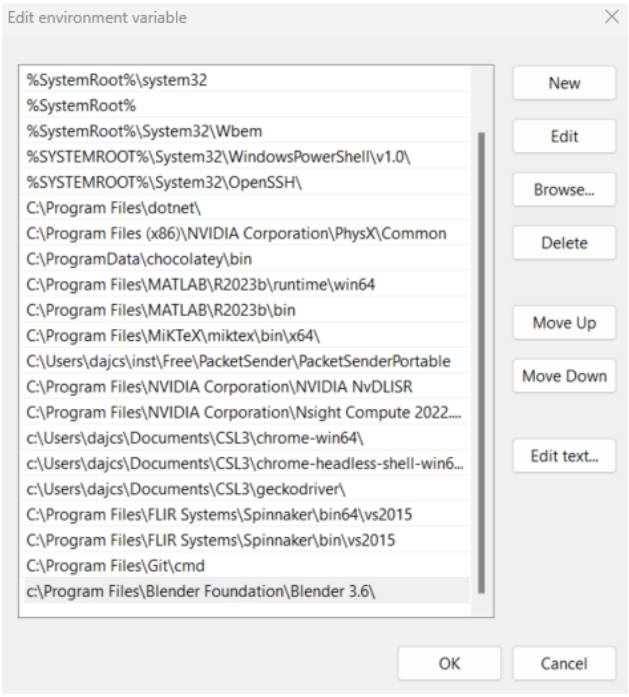

Figure 6 - Add path variable

Press OK all the way. After setting done, opening a new command window should have set the path variable and it should be possible to start Blender by entering `blender`.

The Blender documentation contains additional hints to launch the program from command line for different operating systems.

3.2 Blender's Bundled Python

The Blender program suite includes a complete Python installation.
Blender's `\Help\Save System Info` menu saves a file with relevant system information of our installation, and this shows us the installed python version and python paths. Here is an excerpt from the `system-info.txt` file:

```
Blender:
========================================
version: 3.6.5, branch: blender-v3.6-release, commit date: 2023-10-16 14:30, hash:
cf1e1ed46b7e, type: release

Python:
========================================
version: 3.10.13 (main, Oct 10 2023, 08:34:31) [MSC v.1928 64 bit (AMD64)]
file system encoding: utf-8:surrogatepass
paths:
    ...
    'C:\Program Files\Blender Foundation\Blender 3.6\3.6\python'
    'C:\Program Files\Blender Foundation\Blender3.6\3.6\
                                                python\lib\site-packages'
    'C:\Users\dajcs\AppData\Roaming\Blender Foundation\Blender\3.6\
                                                        scripts\addons'

Python (External Binary):
========================================

binary path: 'C:\Program Files\Blender Foundation\Blender
3.6\3.6\python\bin\python.exe'
version: Python 3.10.13
```

Blender's Python package includes the basic Python modules, but won't have all the extensions installed in the system's Python environment. Namely this package is missing the basic image processing modules like OpenCV, matplotlib or pillow. Opening a python console in Blender we can check the basic modules and we can try to import the image processing modules:

```
PYTHON INTERACTIVE CONSOLE 3.10.13 (main, Oct 10 2023, 08:34:31) [MSC v.1928 64 bit
(AMD64)]

Builtin Modules:       bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context,
bpy.utils, bgl, gpu, blf, mathutils
Convenience Imports:   from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

>>> import cv2
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
ModuleNotFoundError: No module named 'cv2'
```

3.3 Importing Custom Modules into Blender's Python

The Blender API documentation tips and tricks has several workarounds to overcome the missing module problem:

1.) Remove Blender Python subdirectory, this way Blender will then fallback on the system's Python. Depending on platform it might be needed to reference explicitly the Python installation, e.g. on Linux:
`PYTHONPATH=/usr/lib/python3.10 ./blender --python-use-system-env`
Note: The Python (major, minor) version must match the one that Blender comes with.

2.) Copy or link the extensions into Blender's Python subdirectory, so Blender can access them.

We are not going to remove the Blender's Python subdirectory, since that seems a very risky operation. We can try the 2nd option to copy modules into Blender's Python subdirectory.

First, we create a new Python 3.10 environment called `blender` in our conda distribution with the required packages:

```
conda create -n blender python=3.10 numpy pandas matplotlib opencv pillow ipython
```

Activate the new environment, start ipython, import sys and modules of interest and finally check the installation path of the modules:

```
(base) C:\Users\dajcs>conda activate blender

(blender) C:\Users\dajcs>ipython
Python 3.10.13 | packaged by Anaconda, Inc. | (main, Sep 11 2023, 13:24:38) [MSC
v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.15.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import sys, cv2, PIL

In [2]: sys.modules['PIL']
Out[2]: <module 'PIL' from 'C:\\Users\\dajcs\\anaconda3\\envs\\blender\\lib\\site-
packages\\PIL\\__init__.py'>

In [3]: sys.modules['cv2']
Out[3]: <module 'cv2' from 'C:\\Users\\dajcs\\anaconda3\\envs\\blender\\lib\\site-
packages\\cv2.cp310-win_amd64.pyd'>
```

We can notice the differences between the Pillow and OpenCV modules. The Pillow package has two directories: `PIL\` and `Pillow-10.0.1.dist-info\` while the OpenCV module points to a single `cv2.cp310-win_amd64.pyd` file (Python Dynamic library – similar to Windows DLLs).

After copying `PIL\` and `Pillow-10.0.1.dist-info\` into Blender's `c:\Program Files\Blender Foundation\Blender 3.6\3.6\python\lib\site-packages\` directory allows us to import PIL in Blender's Python console, however copying the `cv2.cp310-win_amd64.pyd` file is not enough to import the cv2 module. Similarly copying matplotlib's directories wasn't enough, there were plenty of dependencies required by the package.

We consider our experiment of importing custom modules into Blender's Python a partial success and to keep it simple and easily reproducible we are going to build our scripts considering the vanilla Blender installation.

3.4 Blender as a Module

There is another way – trying to import Blender's `bpy` module into the system's Python package. This could have the advantages:

- External IDEs can be used.
- IDEs can autocomplete Blender modules and variables.
- Scripts can import Blender APIs without having to be run inside of Blender.

Running Blender as a Python module requires a special build option described on the Building Blender as a Python Module wiki page.
The option to build Blender as a Python module is not officially supported, we are mentioning this method to list all our possibilities.

3.5 Debugging in Blender's Python

Admittedly there is a lack of Python debugging support in Blender, but the documentation gives us several workarounds. As mentioned before, when starting Blender from a terminal we can insert `print()` commands into the python script and the printouts will be displayed in the terminal window.
It is possible to insert into the python script the code snippet below:

```
import code
namespace = globals().copy()
namespace.update(locals())
code.interact(local=namespace)
```

This will pause the script and an interactive interpreter will be launched in the terminal. When finished examining variables and program status, after quitting the interpreter the python script will continue execution. A somewhat cryptic one-line version of the previous code snippet is easier to paste into the script:

```
__import__('code').interact(local=dict(globals(), **locals()))
```

If IPython is installed into Blender's Python, we can use its `embed()` function which uses the current namespace. The IPython console has syntax highlighting and autocompletion, features that are missing from the standard interactive interpreter.

```
import IPython
IPython.embed()
```

Installing IPython can be done with the method described in 3.3.
The following directories and files need to be copied into the Blender's Python `site-packages` directory:

```
IPython\
ipython-8.15.0.dist-info\
traitlets\
traitlets-5.7.1.dist-info\
stack_data\
stack_data-0.2.0.dist-info\
stack_data\
stack_data-0.2.0.dist-info\
asttokens\
asttokens-2.0.5.dist-info\
six-1.16.0.dist-info\
six.py
pure_eval\
pure_eval-0.2.2.dist-info\
pygments\
Pygments-2.15.1.dist-info\
pickleshare-0.7.5.dist-info\
pickleshare.py
backcall\
backcall-0.2.0.dist-info\
exceptiongroup\
exceptiongroup-1.0.4.dist-info\
prompt_toolkit\
prompt_toolkit-3.0.36.dist-info\
wcwidth\
wcwidth-0.2.5.dist-info\
colorama\
colorama-0.4.6.dist-info\
```

Unfortunately, the suggested debugging techniques didn't turn out to be useful for this project. The problem is that the Blender GUI is not rendered until our python script is executed, and this way we can't check visually the immediate effects of real-time debugging.

3.6 Installing Flares Wizard

The Flares Wizard can be procured on the blender indie market site, or it can be downloaded here for evaluation purposes.
To install start Blender 3.6.5.

1.) Open \Edit\Preferences\Add-ons\Install...
2.) Locate the file `FlaresWizard-3.0.zip`
3.) Click Install Add-on
4.) Check the Enable Add-on square.

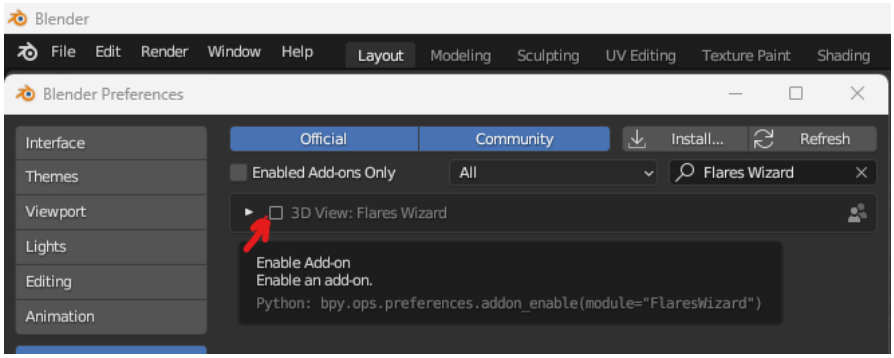

Figure 7 - Installing Flares Wizard add-on

If we are going to use custom presets we need to create and specify the created directory in the `Custom Presets Folder` and check the `Save the preferences` box.

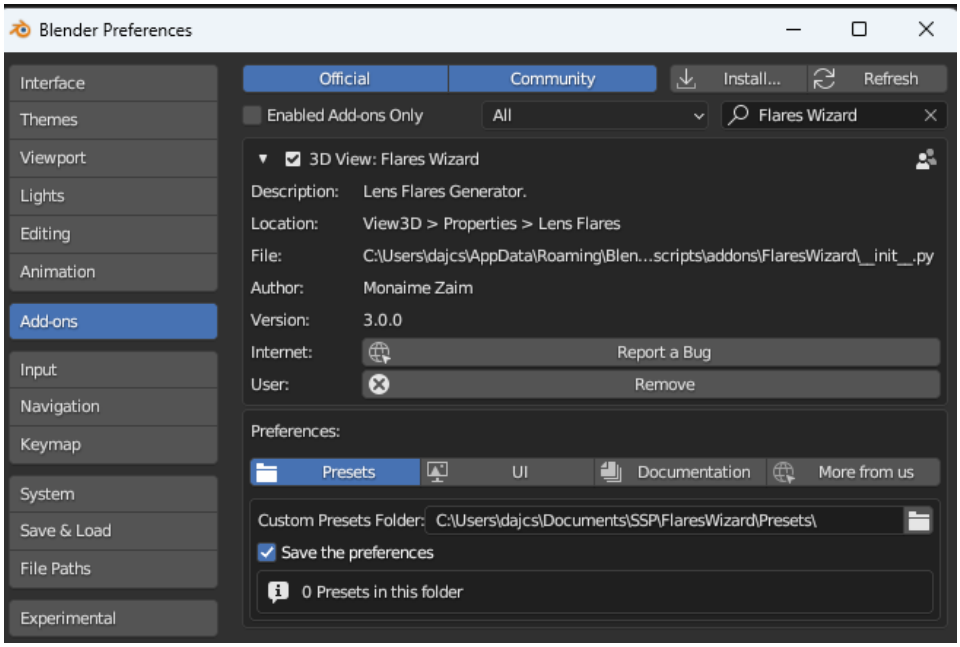

Figure 8 - Setting and saving Custom Presets folder

The add-on will be installed in the directory:

```
c:\Users\dajcs\AppData\Roaming\Blender
Foundation\Blender\3.6\scripts\addons\FlaresWizard\
```

4 The bpy module

In Blender's Python API, the `bpy` module provides different submodules and namespaces for interacting with Blender's data and operations, amongst these the most important are `bpy.data`, `bpy.ops` and `bpy.context`. Understanding where to use and how to use them is crucial for efficient scripting. The general guidelines and common practices of using these interfaces:

- `bpy.data` can be used for direct access to Blender's data blocks. This includes meshes, materials, textures, images, and more. It is generally used when accessing or modifying the properties of objects, materials, or other data elements in a non-contextual manner.
  Example: change and object location.
- `bpy.ops` is used to perform operations or actions, similar to pressing buttons or choosing menu items in the Blender UI. These operations are context-dependent, meaning they depend on the current context (what's selected, the active object, etc.). They are generally higher-level compared to `bpy.data` and can trigger updates in the UI or invoke complex functionalities (like modifiers, render operations, etc.).
  Example: Adding a new object (`bpy.ops.mesh.primitive_cube_add()`).
- `bpy.context` is used to access the current context of the Blender scene. The context contains information about what's currently selected, the active object, the active scene, the active area, etc. It's often used in conjunction with `bpy.ops` to ensure that the right context is set before an operation is performed.
  Example: Accessing the active object (`bpy.context.active_object`).

It is a good idea to enable the “User Tooltips” and the “Python tooltips” in the \ Edit \ Preferences \ Interface \ Display \ Tooltips menu. This will reveal the object or property location within the `bpy.data` structure, and this location can be copied to the clipboard with shortcut Ctrl-Shift-Alt-C. When manipulating through `bpy.data` is not possible, then operation can be performed through the GUI and in the “Info” window the corresponding `bpy.ops` command can be observed.

5 Setting up the Scene

Blender starts with the opening scene: a 2x2x2 cube centered a coordinate (0,0,0), a camera and a light.

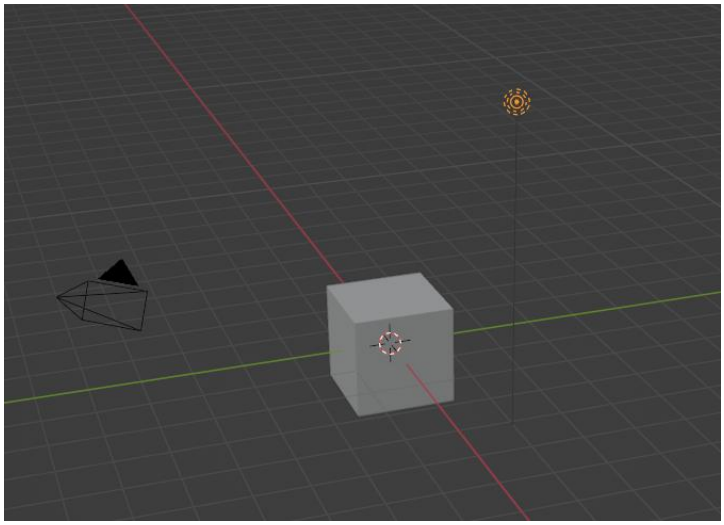

Figure 9 - Blender's opening scene

Pressing the N shortcut will reveal the Lens Flares menu in the top right below the Iten, Tool and View menu. When starting a new project, the Load Lens Flares properties should be clicked.

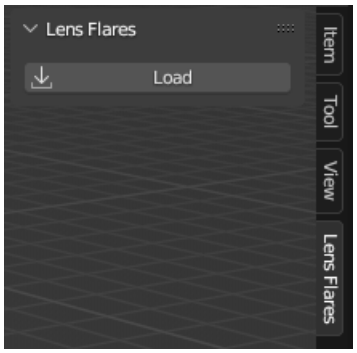

Figure 10 - Load Lens Flares properties

Loading of the Flares wizard through the python interface can be done by the command:

```
# load flares wizard
bpy.ops.flares_wizard.load_props()
```

The add-on uses the drivers in Blender, so “Auto Run Python Scripts” must be enabled in Blender’s preferences, if this isn’t the current setting it can be set by clicking the “Enable” button on the add-on’s tab.

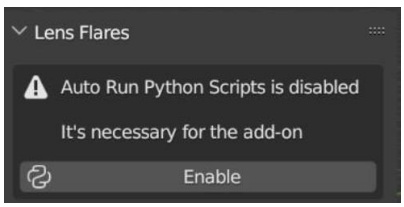

Figure 11 - Enabling Auto Run Python Scripts

The next step is to delete the “Cube” object. This can be done in the GUI by selecting it, pressing the “x” shortcut and clicking “ok”. In the python interface it can be deleted by the commands:

```
# Ensure we are in object mode
bpy.ops.object.mode_set(mode='OBJECT')

# Delete the 'Cube'
####################
if "Cube" in bpy.data.objects:
    # Deselect all objects, select 'Cube' and delete
    bpy.ops.object.select_all(action='DESELECT')
    bpy.data.objects['Cube'].select_set(True)
    bpy.ops.object.delete()
```

The next step is to place the “Camera” at the coordinate `0, 0, z_cam` with orientation towards the World origin. The height `z_cam = +13.88888931274414` has been set at this value so that later when we insert the background plane that will be situated at height 0, corresponding to the X, Y plane.

On the GUI this can be achieved by selecting the “Camera” and on the “N” Item menu setting the Location (0, 0, 13.889) and Rotation (0, 0, 0) as shown on Figure 12.

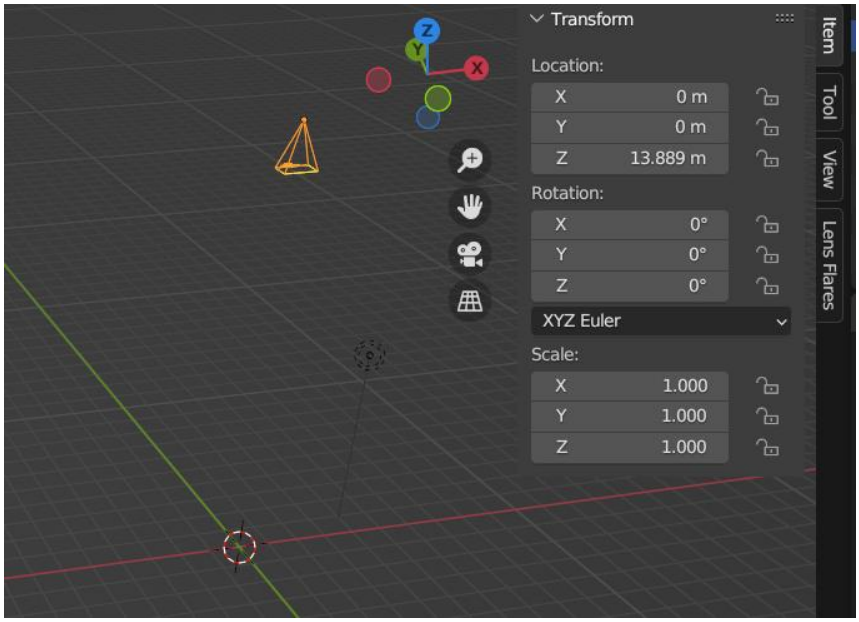

Figure 12 - Placing the "Camera" at Location (0, 0, +13.889), Rotation (0, 0, 0)

Expert advice: when placing the mouse cursor over any value and pressing the “Backspace” key this will reset the field values to the default – in our case both Location and Rotation have default values of (0, 0, 0).

The python code to achieve this setting:

```
# Move 'Camera' to (0,0,z_cam), looking down
############################################
# camera z coordinate
z_cam = +13.88888931274414
if "Camera" in bpy.data.objects:
    camera = bpy.data.objects['Camera']
```

```
camera.location = (0, 0, z_cam)
camera.rotation_euler = (0, 0, 0) # in radians
```

The following step is to add a Background Plane. This can be done on the GUI Lens Flare \ Extra Options \ Add Background Plane menu.

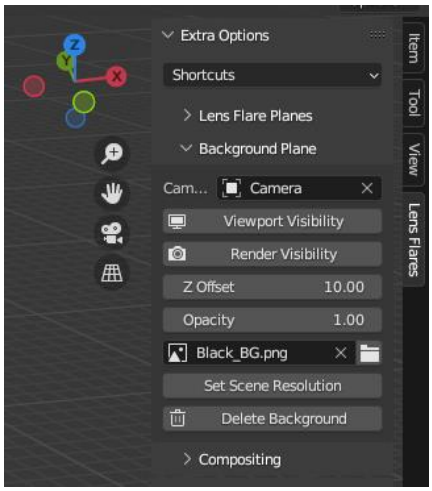

Figure 13 - Add Background Plane

By default, a black background is added with the image "Black_BG.png" but it is possible to replace this with a custom image.
This is the python code to set the variable `bg_filepath` to the file given in the arguments or to set it to `Black_BG.png` if not specified in the arguments.

```
# Add background image
######################
# background image fname with full path or "Black_BG.png"
bg_filepath = args['ref_image']
if bg_filepath and os.path.exists(bg_filepath):
    bg_filepath = os.path.abspath(bg_filepath)
else:
    bg_filepath = 'Black_BG.png'  # Black_BG.png
```

The Flares Wizard is going to insert the Background Plane at a distance which will provide a field of view with a width of 10 meters.

To find the distance at which the subject should be to achieve a field of view width of 10 meters with the default Blender camera which has a sensor width of 36mm and focal length of 50mm, we can use the concept of similar triangles. The field of view can be understood as the width of the scene that the camera can capture at a certain distance. The sensor width to focal length ratio should be equivalent to the field of view width (10 meters in this case) to the distance from the subject (what we're trying to find).

$$\frac{SensorWidth}{FocalLength} = \frac{FieldofViewWidth}{DistancetoSubject}$$

We'll rearrange it to solve for the distance to the subject:

$$DistancetoSubject = \frac{FieldofViewWidth * FocalLength}{SensorWidth}$$

Plugging in the numbers:

$$DistancetoSubject = \frac{10m * 50mm}{36mm} = 13.889\,m$$

The python commands to add the Background Plane:

```
# add Flares Wizard background plane
bpy.ops.flares_wizard.add_background()
# add background image
bpy.ops.flares_wizard.open_image(type="BG", filepath=bg_filepath)
# match scene resolution to image resolution
bpy.ops.flares_wizard.set_scene_resolution()
```

The last command will set the resolution to the given reference image resolution if this has been specified.

When checking on the GUI the properties of the inserted Background Plane we can observe that the Location x, y, z and the Scale x and y coordinates are displayed on magenta background as seen on Figure 14.

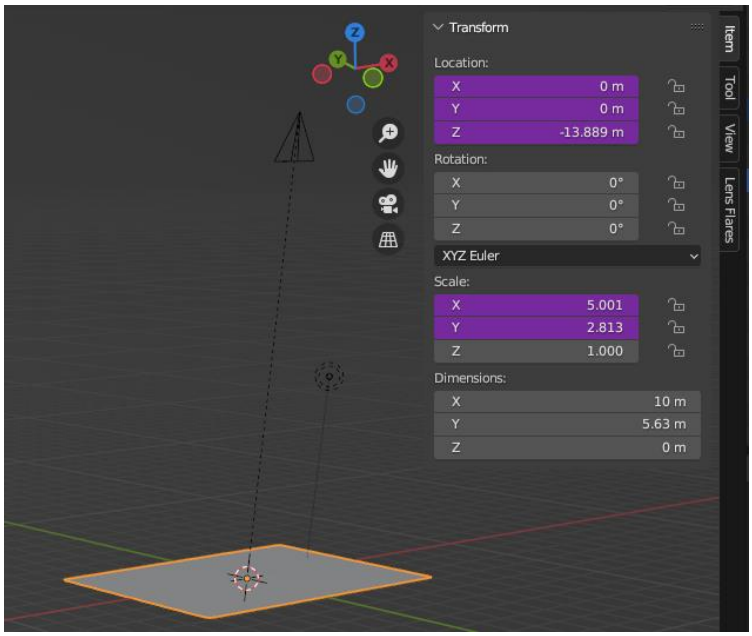

Figure 14 - Background Plane Location and Scale

The reason for this is that the Location coordinates and the Scale values are computed based on other input variables.

The Location formulas:

X	shift_x
Y	shift_y

Z	(-lens / sensor_width) * z_offs

The Scale formulas:

X	(clamp((res_x * aspect_x) / (res_y * aspect_y)) * z_offs / 2) + 0.001
Y	(clamp((res_y * aspect_y) / (res_x * aspect_x)) * z_offs / 2) + 0.001

Where:

- shift_x is the Camera horizontal shift (0 by default)
- shift_y is the Camera vertical shift (0 by default)
- lens is the focal length (50 mm by default)
- sensor_width is the camera sensor width (36 mm by default)
- res_x is the number of horizontal pixels in the rendered image (1920 by default)
- res_y is the number of vertical pixels in the rendered image (1080 by default)
- aspect_x is horizontal aspect ratio for non-square pixel output (1 by default)
- aspect_y is the vertical aspect ratio for non-square pixel output (1 by default)
- z_offs is the Z Offset set in the Flares Wizard add-on (set to 10 meter)

The next step is to place the light in the Background Plane at arbitrary chosen x = 2 and y = 2 coordinates. The Background Plane is in the XY plane which implies the third coordinate z = 0. The Rotation of the lamp will be set to (0, 0, 0). We are moving the lamp to the Background Plane because later we are going to attach the LF effects to the lamp and this will simplify the LF origin computations. Moving the lamp can be done in the usual way on the GUI as shown on Figure 15.

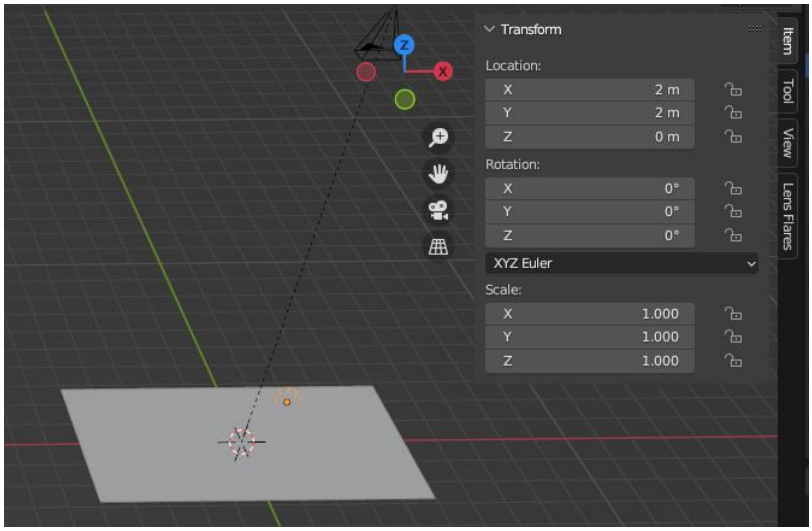

Figure 15 - Moving the lamp to Location (2, 2, 0) and Rotation (0, 0, 0)

The corresponding python code:

```
# get 'FW_BG_Plane' location and scale, move light to (2, 2, z_bg + z_cam)
#######################################################################
if "FW_BG_Plane" in bpy.data.objects:
    bg_plane = bpy.data.objects['FW_BG_Plane']
    z_bg = bg_plane.location.z    # -13.889  (appears at z=0)
    # bg_dim = D.objects['FW_BG_Plane'].dimensions  # Vector((10.0, 7.5, 0.0))

if "Light" in bpy.data.objects:
```

```
light = bpy.data.objects['Light']
light.location = (2, 2, z_bg + z_cam)            # (2, 2, 0)
light.rotation_euler = (0, 0, 0) # in radians
```

The next step is to switch to camera view and to rendered shading. On GUI this is simply clicking the "camera" and the "rendered shading" buttons as shown on Figure 16.

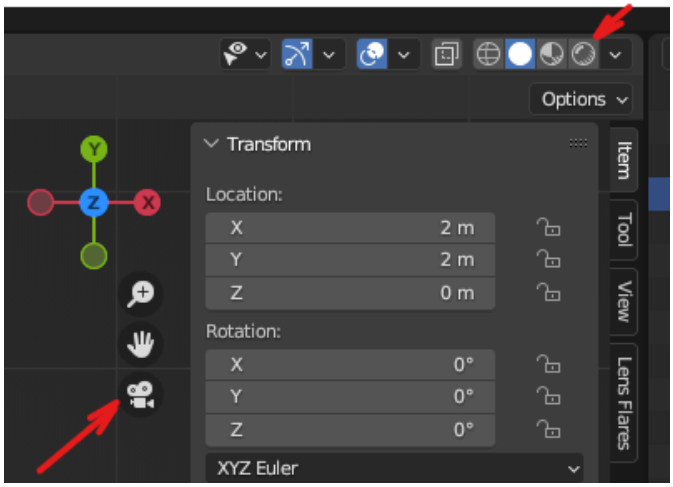

Figure 16 - Turning on Camera view and Rendered shading

The corresponding python code:

```
# Switch to Camera View & Rendered shading
##########################################
bpy.context.scene.camera = bpy.data.objects['Camera']
# Find a 3D view area to change context
for area in bpy.context.screen.areas:
    if area.type == 'VIEW_3D':
        # Set the view to the active camera and shading to 'rendered'
        area.spaces.active.region_3d.view_perspective = 'CAMERA'
        area.spaces.active.shading.type = 'RENDERED'
        break
```

In Blender the World surface has an illumination and we are going to turn off this to not interfere with our LF effects. This can be done on the GUI by selecting the World \ Surface and then Color setting to black and Strength to 0, as shown in Figure 17.

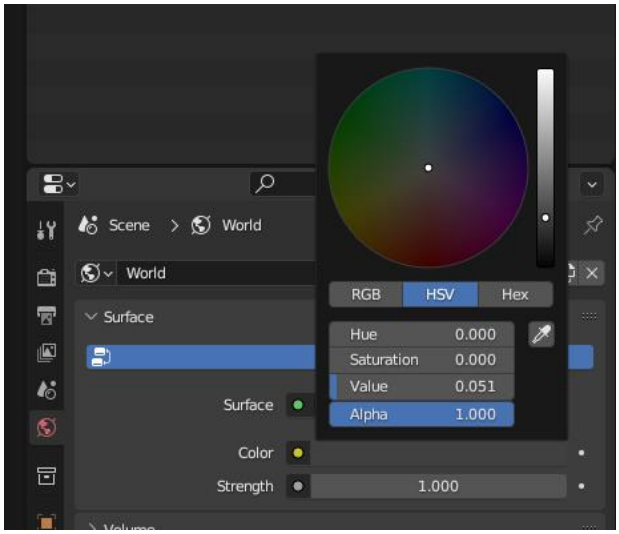

Figure 17 - Setting World Surface Color to black and Strength to 0

The corresponding python code:

```
    # set world background illumination to 0
    # set Color to Black
    bpy.data.worlds["World"].node_tree.nodes["Background"].inputs[0].default_value
= (0, 0, 0, 1)
    # set Strength to 0
    bpy.data.worlds["World"].node_tree.nodes["Background"].inputs[1].default_value
= 0
```

The next step is to set the Eevee rendering engine \ Color Management \ View Transform from “Filmic” to “Standard” which can be done on the GUI by selecting it from the dropdown menu.

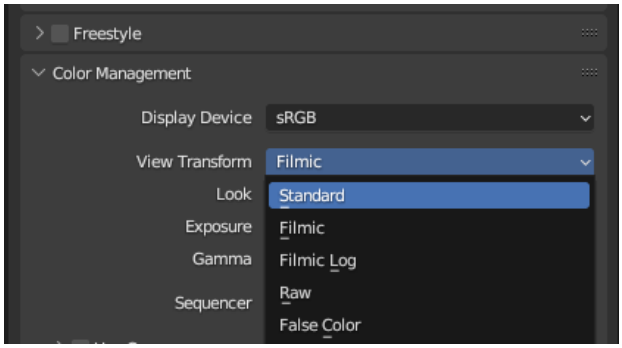

Figure 18 - Setting Eevee rendering Color Management View Transform to "Standard"

This can be achieved with the python code:

```
    # Set Eevee renderig Color Management \ View Transform from 'Filmic' to
'Standard'
    bpy.context.scene.view_settings.view_transform = 'Standard'
```

We are going to use the Eevee rendering engine because it is two orders of magnitude faster than the Cycles rendering engine and in our use case the LF effects seem more realistic. The “Standard” View Transform is better at preserving the colors generated by the Flares Wizard.

We are going to set the output file format to “JPEG”. We were considering to use the “PNG” format together with the alpha channel, but it turned out that the alpha channel is not a solution to combine the LF only images with other target images, as presented in chapter 9, Postprocessing of LF-only images.

The output format is selected on GUI from the dropdown menu:

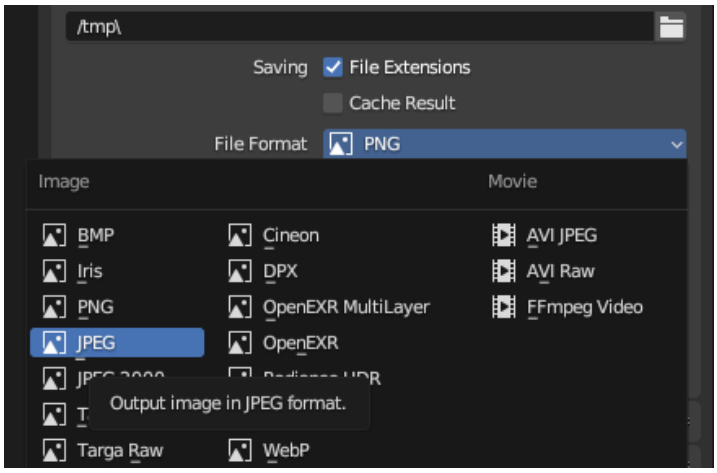

Figure 19 - Setting the output format to JPEG

This can be achieved with the python code:

```
# Set output format
bpy.context.scene.render.image_settings.file_format = 'JPEG'
```

The last step of our scene preparation is to select the “Light”, make it active and attach to this a blank LF.

```
# select 'Light' and make it active
#################################
bpy.ops.object.select_all(action='DESELECT')
obj = bpy.data.objects['Light']
obj.select_set(True)
bpy.context.view_layer.objects.active = obj

# Add Blank Lens Flare
# bpy.ops.flares_wizard.presets_browser()
bpy.ops.flares_wizard.add_lens_flare()
```

The properties of the Flares Wizard we are going to examine in more detail in the next chapter.

6 Properties of the Flares Wizard

The Flares Wizard add-on can be accessed via the “N” shortcut and clicking the “Lens Flares” menu as already presented at the beginning of Setting up the Scene.

We can add a new LF by pressing the “+” button as shown in Figure 20. We should make sure that the “Light” object is elected because we are going to attach the LF to the Light.

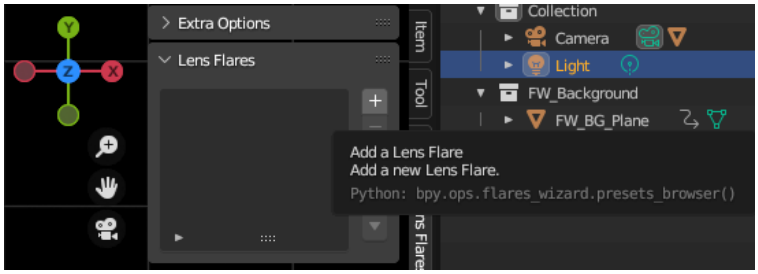

Figure 20 - Adding a LF effect

After pressing the "+" a new pop-up window appears where we have the possibility to choose from 50 different LF presets.

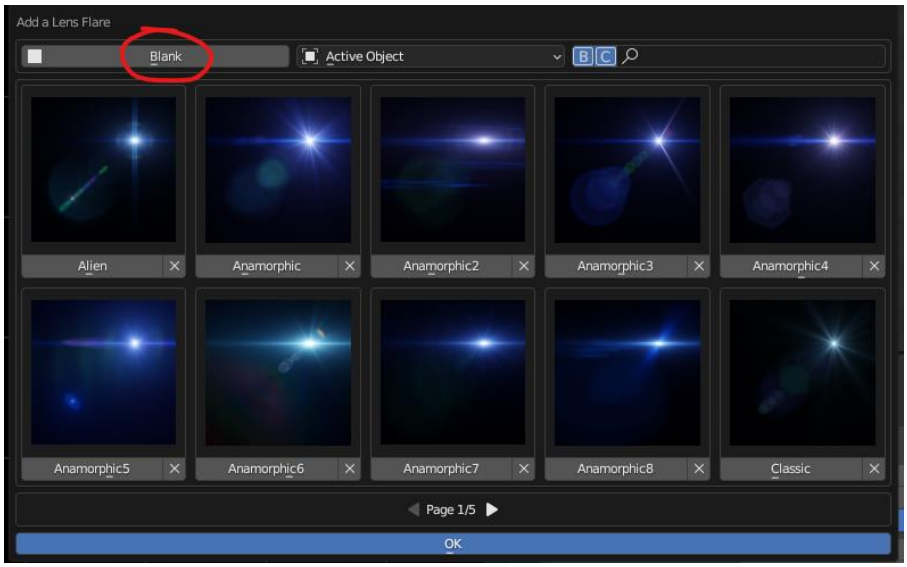

Figure 21 - Choosing the LF effect

The preset LF effects have a great artistic value, but we are going to select a "Blank" LF effect as shown on Figure 21. We have the option to attach the LF effect to the active Object or to the active Collection. We will attach it to the active "Light". There is a possibility to load a preset LF effect, to make modifications and to save it as a custom LF. The "B" and "C" buttons can be selected or deselected whether we want to see the built-in or custom presets, or both.

After adding the LF effect a new window can be opened where we can add Elements to the LF by pressing the "+" button as shown on Figure 22

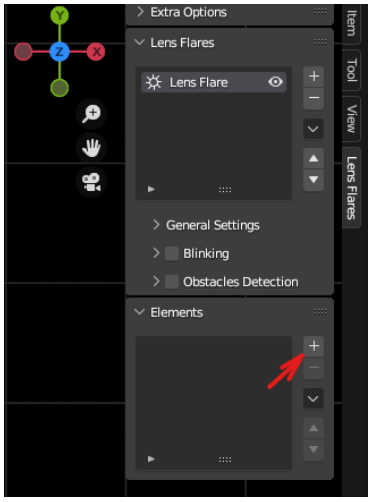

Figure 22 - Adding Element to LF

Now we will have the possibility to choose from 10 different Elements to be added as seen on Figure 23.

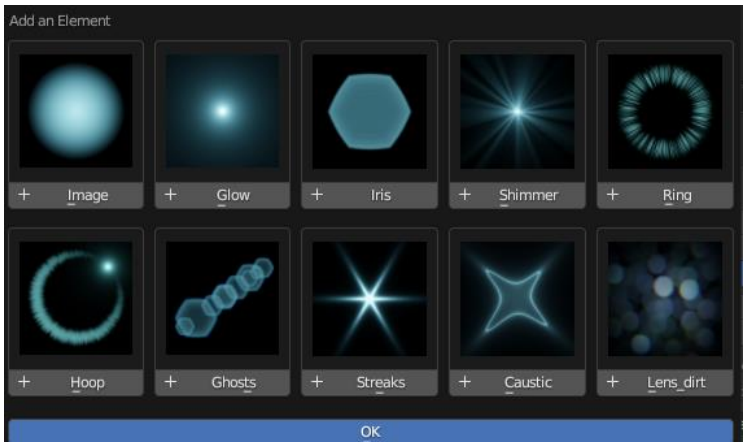

Figure 23 - Elements to be added to LF

It can be added arbitrary number of elements, even of the same type, and any element can be removed by selecting it and pressing the "– " button.

Every element has property group Shader, Transform and Proximity Trigger. The elements can be procedural, or image based. The procedural means that the LF effects are computed based on functions and the image-based elements are manipulating a sample image. The image-based elements are the: Image, Ghosts and Lens_dirt. These can be recognized by the fact that in the Shader group there is an Image property, while the procedural elements have an Interpolation property (Figure 24 and Figure 25)

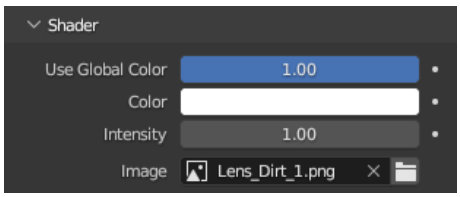

Figure 24 - Shader properties for image-based Element

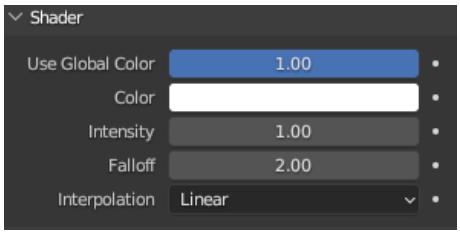

Figure 25 - Shader properties for procedural Element

For both element types we have the Transform group, with the properties: Position, X, Y Location, X, Y Lock, Rotation, X, Y Scale and Track the target (Figure 26)

Figure 26 - The Transform group properties of an Element

The Proximity Trigger allows to change the intensity of the element dynamically depending on the distance between the target (Light) and the (borders & center) of the active camera (Figure 27).

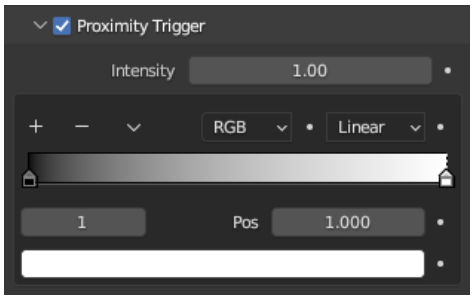

Figure 27 - Proximity Triggers settings of an Element

Every element has Special Options with settings relevant only for that type of Element – e.g. the Streaks element has Count, Random Rotation & Seed and Random Scale & Seed – as seen on Figure 28.

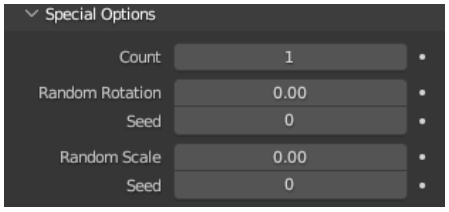

Figure 28 - Special Options for the Streaks Element

Here is an example python code of adding a new Element of type “STREAKS” and setting some of the properties.

```
# add new element of type: STREAKS
bpy.ops.flares_wizard.add_element(type=ele['STREAKS'])
# switch context to current element
bpy.context.scene.fw_group.coll[0].ele_index = 0
# set element properties
bpy.context.scene.fw_group.coll[0].elements[0].scale_x = 8.0
bpy.context.scene.fw_group.coll[0].elements[0].scale_y = 0.4000000059604645
bpy.context.scene.fw_group.coll[0].elements[0].light_falloff = 2.630000114440918
bpy.context.scene.fw_group.coll[0].elements[0].streaks_count = 4
bpy.context.scene.fw_group.coll[0].elements[0].color = [0.06963802129030228,
                                                        1.0,
                                                        0.1308511197566986,
                                                        1.0]
bpy.context.scene.fw_group.coll[0].elements[0].use_global_color = 0.733333379053
bpy.context.scene.fw_group.coll[0].elements[0].intensity = 0.7000000476837158
```

7 The program package

The package is available at the https://github.com/dajcs/LFW link.
Contains 3 python scripts:

- lf_setup.py
- lf_gen.py
- utils.py

The script lf_setup.py helps us to compose an LF effect and save the settings in a JSON file.
The script lf_gen.py is used to mass-produce the LF effects on top of an existing image dataset, or on top of black background.
The script utils.py contains helper functions for the previous two scripts.

7.1 Setting up and saving an LF effect with `lf_setup.py`

The lf_setup.py main program is quite short:

```
def main():
    args = utils.get_args(sys.argv)
    prepare_scene(args)

    with open(args['lf_params']) as f:
        elements = json.load(f)       # load list of lf elements from file
        lf = utils.LF(elements)       # create lf object based on the elements
        lf.apply()                    # apply lf element properties in Blender

    print('''\n\n\n
    Press "N", select "Lens Flares" in the side menu
    Please adjust the Lens Flare effects for your project needs

    When finished adjustments, select Scripting workspace (top menu right)
    Save settings by entering the commands below into Blender's Python console
(left middle window)

    import utils
    utils.save('filename.json')

    ''')
```

These are the main steps:

- getting the arguments with the utils.get_args(sys.argv) function
- preparing the scene with the prepare_scene(args) function. All the steps and their realization through python commands has been presented in Chapter 5
- loading the elements specified from a JSON file
- creating an lf object with the loaded elements
- asking the user to prepare the LF effects on GUI and save the result to a JSON file

After the user saved the JSON file, can close Blender, no need to save the Blender file because we are not going to use it.

7.2 Mass production of LF effects with `lf_gen.py`

The script lf_gen.py starts similar to lf_setup.py: imports modules, reads the arguments, prepares the scene, creates an lf object with elements specified in the JSON file and applies the lf effects.

The next step is to decide whether we are going to apply the LF effects on top of an existing image dataset, or we are going to generate LF effects on top of a black background. The decision is made upon the `'source'` parameter in the code snippet below:

```
if args['source']:
    imgs = glob.glob(os.path.join(args['source'], '*'))    # all files in source
directory
    imgs = [im for im in imgs if os.path.splitext(im)[1] in ['.png', '.jpg',
'.jpeg', '.bmp']]  # keep images
    if not imgs:
        try:
            nr_img = int(args['source'])
        except ValueError:
            print(f"\nThe specified --source {args['source']} has no images and it
can't convert to integer\n")
            sys.exit()
```

First, we consider the `'source'` parameter a directory and we try to read the images in that directory. If there are no images in the directory, then we try to convert the `'source'` parameter to integer, and if that isn't successful then we display error message and exit the program.
If there are images, then this code will be executed:

```
if imgs:
    # add LF to images in --source directory
    for im in imgs:
        # set im as bg_image
        bg_im = os.path.abspath(im)
        # add as a background image
        bpy.ops.flares_wizard.open_image(type="BG", filepath=bg_im)
        # match scene resolution to image resolution
        bpy.ops.flares_wizard.set_scene_resolution()

        # randomize lf origin
        utils.rand_lf_origin(args['outside_image'])

        # if there are ranges to be processed, create and apply new sample
        if lf.delta:
            lf.new_sample()
            lf.apply_delta()

        # create output filename
        base_name_with_ext = os.path.basename(bg_im)  # -> e.g. img003077.jpg
        base_name = os.path.splitext(base_name_with_ext)[0] # ---> img003077
        output_fname = os.path.join(os.path.abspath(args['output']), base_name +
'_lf.jpg')
        # Set output path
```

```
        bpy.context.scene.render.filepath = output_fname
        # Render the scene, save image
        bpy.ops.render.render(write_still = True)
```

For every image in the directory:

- We get the absolute path of the image, and we apply it to the Background Plane. Our script can receive in the `'source'` parameter relative path, but Blender is not aware of the location from where we started the command line and therefore needs the absolute path
- We then set scene resolution to the image resolution
- We randomize the `'Light'` position – therefore the LF origin position with the script `utils.rand_lf_origin()`
- If the LF effects differ from image to image (`lf.delta()` is True) then we create new effect and apply it with `lf.new_sample()` and `lf.apply_delta()`
- We create and set the output filename, we render it, and we save it

When we don't have source images then the process is very similar, the difference is at setting the scene resolution – this is based on the reference image resolution – if available, or we go with the command line parameters or default values.

7.3 The helper functions in `utils.py`

The most important helper functions in the utils.py are:

- get_args()
- save()
- LF class
- ele_sample()
- apply_ele_prop()
- rand_lf_origin()

7.3.1 The `get_args()` function

The `get_args()` function has a vanilla definition with a few exceptions.
We need to distinguish between the command line parameters used by Blender and by our script. These parameters are separated by the '--' token. This is done by the code snippet below:

```
    # Find the index of '--' token
    # parameters before '--' are for Blender
    try:
        idx = argv.index("--")
        my_argv = argv[idx + 1:]  # Arguments after the first '--'
    except ValueError:
        print('\n\nWarning!!!  Couldn\'t find token "--" needed between Blender\'s
arguments and current script\'s arguments.\n')
        my_argv = []
```

Here `my_argv[]` will get the arguments after the token '—' and we are going to parse these.
The `argparse.Argumentparser()` is defined with description/usage/epilog and we are using the `formatter_class=RawDescriptionHelpFormatter` parameter in order to be able to format the descriptions with newline to make it structured and more readable.

7.3.2 The `save()` function

The `save()` function consists of a few lines:

```
    # value like bpy.data.images['picture.png'] -> remove because can't be saved to
json, keep ['picture.png']
    js = str([ ele.to_dict() for ele in
bpy.context.scene.fw_group.coll[0]['elements']]).replace('bpy.data.images','')
```

```
    try:
        with open(fname, 'w') as f:
            json.dump(eval(js), f, indent=4)
        print(f'Lens Flare parameters saved to file "{fname}"\n')
    except Exception as e:
        print(f"An error occurred:\n\n{e}\n\nTry again...")
```

We are getting all elements of an LF and applying the `.to_dict()` method we're getting the elements properties and their values in the form of dictionary. This list of dictionaries could be saved to JSON, the only problem is that the 'image' property values are `bpy.data.images` objects:

```
{'image': bpy.data.images['Poly_circle_smt2.png']}
```

A quick solution for this is to get a string representation of the list of dictionaries, replace the `'bpy.data.images'` with empty string, thus keeping only the list string, and then the resulting list of dictionaries string can be evaluated and saved to a JSON file.

7.3.3 The LF class

The LF class stores the LF elements properties in the form of a list of dictionaries and applies the property settings in Blender. This is the initialization of the class:

```
class LF:
    '''
    stores LF elements properties
    and applies property settings in Blender
    '''
    def __init__(self, elements):
        # raw elements might contain ranges
        self.raw_elements = elements
        self.sample_elements = []     # updated by self.new_sample()
        self.delta_elements = []      # updated by self.new_sample()
        # delta initially False, if there are ranges in raw_elements -> set to True
by self.new_sample()
        self.delta = False
        # get sample_elements by sampling the ranges, delta_elements: contains only
the sampled properties
        self.new_sample()
```

Here we can see that there are `raw_elements`, `sample_elements` and `delta_elements`. The reason to have so many elements is because the user is allowed to edit the generated JSON file which stores the LF settings. For example, a GHOST element in the saved JSON file has this form:

```
    {
        "name": "ELEMENT5",
        "ui_name": "Ghosts",
        "type": "GHOSTS",
        "flare": "LF",
        "scale_x": 0.15000000596046448,
        "scale_y": 0.15000000596046448,
        "position": 1.25,
        "intensity": 0.10000000149011612,
        "random_scale": 0.5,
        "random_scale_seed": 6,
        "ghosts_count": 1,
        "image": [
            "Poly_circle_smt0.png"
        ],
        "hide": 0
    }
```

The user is allowed to replace any float value with a two-element list of floats, any int value with a two-element list of integers, and the list of an image filename can be expanded to a list of arbitrary number of image filenames. A valid edit of the previous GHOST element, where "position", "ghosts_count" and "image" is expanded can be seen here:

```
    {
        "name": "ELEMENT5",
        "ui_name": "Ghosts",
        "type": "GHOSTS",
        "flare": "LF",
        "scale_x": 0.15000000596046448,
        "scale_y": 0.15000000596046448,
        "position": [1.0, 2.0]
        "intensity": 0.10000000149011612,
        "random_scale": 0.5,
        "random_scale_seed": 6,
        "ghosts_count": [1, 5],
        "image": [
            "Poly_circle_smt0.png",
            "Poly_circle_smt1.png",
            "Poly_circle_smt2.png"
        ],
        "hide": 0
    }
```

These edited elements will be read then from the JSON file, and it will be stored in the `self.raw_elements`. The LF class `.new_sample()` method is going to convert the edited two-element lists back to a single random value between the limits of the two elements of the list, or by choosing randomly an image from the expanded image list.

The `new_sample()` method is a short one:

```
    def new_sample(self):
        '''
        Setting of:
        self.sample_elements: list of dict,
                    element properties with ranges sampled to a single value
                    the other properties kept intact
        self.delta_elements : list of dict,
                    only element properties with ranges sampled to a single value
        self.delta: bool, -> True if at least 1 range property has been sampled
        '''
        self.sample_elements = []
        self.delta_elements = []
        for e in self.raw_elements:
            new_e, delta_e = ele_sample(e)
            self.sample_elements.append(new_e)
            self.delta_elements.append(delta_e)
            if delta_e:
                self.delta = True
```

It iterates through all elements in the `self.raw_elements` and calling the `ele_sample()` function it will get back a `new_e` and `delta_e`. The `new_e` contains every element property (with ranges converted to single value) and the `delta_e` contains only properties where there has been a conversion. The `self.delta` will be set to `True` if there has been at least one conversion.

The `apply()` and the `apply_delta()` methods are very similar. They both are going to generate commands to create and set the LF element properties in Blender. The difference between the two methods – as the name suggests – is that `apply()` is creates new elements and sets all the

properties of that element, while the `apply_delta()` is skipping the creation of new elements, and it will manipulate only the `self.delta_elements` properties.
Here is the `apply()` method:

```
    def apply(self):
        '''
        The function is going to set the self.sample_elements properties in Blender
        '''
        for i, ele in enumerate(self.sample_elements):
            bpy.ops.flares_wizard.add_element(type=ele['type'])  # element type:
STREAKS, GHOSTS, SHIMMER,...
            bpy.context.scene.fw_group.coll[0].ele_index = i     # set current
element context
            print(f'\nele = {list(ele.items())[:4]}')
            for prop in ele.keys():
                apply_ele_prop(i, prop, ele[prop])
```

It iterates through all the elements, and within an element it iterates through all the properties calling the function `apply_ele_prop()` with the element index, the property and the property value.

```
def apply_ele_prop(i, prop, val):
    '''
    i : int, element index
    prop : str, property to be set
    val : int/float/list - depending on prop, value to be set
    The function is going to set the LF element properties to val in Blender
    '''
    if prop in ['name', 'ui_name', 'type', 'flare']:
        return
    elif prop == 'image':
        if not val[0] in bpy.data.images:
            # texture_path in Windows, on Linux 'Users' -> 'home' (to be checked)
            texture_path = os.path.join(
                [p for p in sys.path if 'Users' in p and p.endswith('addons')][0],
                                                    'FlaresWizard', 'Textures')
            filepath = os.path.join(texture_path, val[0])
            bpy.ops.flares_wizard.open_image(type="ELEMENT", filepath=filepath)
        print(f'bpy.context.scene.fw_group.coll[0].elements[{i}].{prop} =
bpy.data.images{val}')
        exec(f'bpy.context.scene.fw_group.coll[0].elements[{i}].{prop} =
bpy.data.images{val}')
    else:
        print(f'bpy.context.scene.fw_group.coll[0].elements[{i}].{prop} = {val}')
        try:
            exec(f'bpy.context.scene.fw_group.coll[0].elements[{i}].{prop} =
{val}')
        except TypeError:
            # likely an Enum property, the value should be a string
            print(f'bpy.context.scene.fw_group.coll[0].elements[{i}].{prop} =
"{val}"')
            exec(f'bpy.context.scene.fw_group.coll[0].elements[{i}].{prop} =
"{val}"')
```

The `apply_ele_prop()` function is going to generate the python commands needed to set the LF element property in Blender as presented at the end of chapter 6. The commands will be generated as a string (printed in the terminal window) and the python `exec()` function is converting the string to python command and execute it.
When we have an "`image`" property which contains an image not yet known to `bpy.data.images` then this image should be loaded by referring to it through the absolute path. The absolute path

depends on the OS, currently it is inferred in Windows, and the script should be fixed with the Linux and macOS paths – which will be done once we get the exact location on these OS-s.
There are a few properties where the value type is `enum`, here we're getting a type error when trying to set the property value, and we're repeating the command with the value as a string.

7.3.4 The `rand_lf_origin()` function

This function is straightforward, it randomizes the `'Light'` location in the background plane – therefore randomizing the LF origin.

```
def rand_lf_origin(outside_image_percent):
    '''
    orig_outside_image : int, percentage of the image size
    places 'Light' - therefore LF origin randomly on bg_plane
    '''
    # get bg_plane dimensions
    bg_width, bg_height, _ = bpy.data.objects['FW_BG_Plane'].dimensions
    lf_origin = np.array([bg_width, bg_height]) * (1 + outside_image_percent / 100)
# default +20% outside image allowed
    lf_origin_middle = lf_origin / 2
    lf_origin *= np.random.rand(2)    # randomize
    lf_orig_x, lf_orig_y = lf_origin - lf_origin_middle    # shift to middle
(centered in (0,0))
    # set Light x, y coords
    bpy.data.objects['Light'].location = (lf_orig_x, lf_orig_y, 0)
```

We start by getting the background plane dimensions, we expand this by the `outside_image_percent`, we transform the coordinates corresponding to the fact that the center of the background plane is at x, y coordinates (0, 0) and then we set the `'Light'` location to the computed (x, y, 0) coordinates.

8 Using the LFW package

The recommended way of using the LF generating scripts is by installing Blender, adding it to the path and setting up the desired lens flare with command:

```
blender --python lf_setup.py -- [options]
```

Mass producing of the LF effects can be done with command:

```
blender --python lf_gen.py -- [options]
```

If the user doesn't read the instructions and just starts one of the LFW scripts without Blender, it will get displayed a basic help message:

```
python lf_setup.py

  Usage:
    blender --python lf_setup.py -- [options]
    blender --python lf_gen.py -- [options]
  More help:
    blender --python lf_setup.py -- --help
    blender --python lf_gen.py -- --help
```

If the user follows the advice and enters the suggested command for more help, it will get displayed a comprehensive help message:

```
blender --python lf_setup.py -- --help

usage:
blender [-b] --python lf_[setup|gen].py -- [LFW options]

LFW scripts: lf_setup.py and lf_gen.py is used to generate Lens Flare effects using Blender with Flares Wizard add-on.
It is recommended to add Blender program location to the path and start Blender via cmd line from a terminal window, as shown.
The parameters before token "--" are interpreted by Blender's python; parameters after token "--" are used by the LFW scripts.
This help message can be displayed by command:
blender --python lf_setup.py -- --help

options:
  -h, --help            show this help message and exit
  -b, --background      background (headless) blender execution (parameter handled by Blender)
  -P PYTHON, --python PYTHON
                        path to python script executed by blender (parameter handled by Blender). Use "--" to separate Blender and LFW script
                        parameters.After "--" all the parameters will be parsed by the LFW scripts
  -ri REF_IMAGE, --ref_image REF_IMAGE
                        path to image used as a background for lens flares setup or getting resolution by lf_gen
  -lf LF_PARAMS, --lf_params LF_PARAMS
                        path to json file storing lens flares settings
  -s SOURCE, --source SOURCE
                        path to source directory of original images, or int representing nr of images to generate LF on black background
  -o OUTPUT, --output OUTPUT
                        path to output directory for images with LF, it will be created if it doesn't exist
  -rx RES_X, --res_x RES_X
                        resolution X (width, default 1920) of the output images, considered when no ref_image and source is a number (generating LF on
                        black background)
  -ry RES_Y, --res_y RES_Y
                        resolution Y (height, default 1080) of the output images, considered when no ref_image and source is a number (generating LF on
                        black background)
  -oi OUTSIDE_IMAGE, --outside_image OUTSIDE_IMAGE
                        percentage of how far can move the LF effect origin outside the image borders. Default 20 percent of the image size.

Example:
blender --python lf_setup.py -- --ref_image LTV.png
can be used to prepare the json file containing the lens flare effects.
This starts with displaying the ref_image LTV.png, on top of it a basic LF effect.
Add and/or adjust LF elements and save LF to a json file - as described in the terminal window.
The saved json file can be further edited by replacing individual values with two-element lists,
and lf_gen.py will take a random sample between the new list's 2 elements.
Example:
blender --python lf_gen.py -- --lf_params my_lf_params.json --source images --output outimages
is used to add LF effects specified in my_lf_params.json to images in "images" directory and save them to "outimages" directory.
Github:
https://github.com/dajcs/LFW
```

8.1 Performance

According to our measurements there was a 0.2 to 0.8 second processing time per image – depending on LF complexity and image resolution – when the Eevee rendering engine was used.

Figure 29 - LF effect rendered with Eevee

We tried the Cycles rendering engine as well. This had a processing time of about 1.5 minutes per image while visually there was no improvement in the output. The images rendered by Eevee and Cycles can be seen on Figure 29 and Figure 30.
Initially we were planning to make the rendering engine an option but given these results the Cycles rendering engine is not a viable option for mass production of LF effects.

Figure 30 - LF effect rendered with Cycles

9 Postprocessing of LF-only images

We made a case study where the user is producing an LF-only image and it tries to apply this on top of a target image. In our study the LF-only image is the first built-in preset from the Flares Wizard seen on Figure 31.

Figure 31 - LF-only image

The target image can be seen on Figure 32.

Figure 32 - Target image

We rendered with Blender the LF effect applied over the target image, and we tried to replicate this with our own composition schema.

Figure 33 - LF over target image composited in Blender

We made several experiments using the alpha channel, that is after rendering an LF-only image we saved it as a .png file with RGBA channels. Unfortunately, we couldn't get an alpha channel to mimic the LF effect shape, we could get only fully opaque alpha channel (when we had a black background plane) or a fully transparent alpha channel (when we deleted the background plane).
We made an experiment with a custom semi-transparent alpha channel, but the end result was a dark image (Figure 34)

Figure 34 - Composition with semi-transparent alpha channel

The next idea was to create our own alpha channel based on the pixel intensity of the LF-only image. This had the best results amongst all the alpha channel experiments, as it can be seen on Figure 35.

Figure 35 - Compositing with alpha channel based on overlay image pixel intensity

We made then an experiment without alpha channel, we just added together the pixels of the target image and the overlay LF-only image. This introduced fake colors as seen on Figure 36

Figure 36 - Summing target image and overlay image pixels

The next idea was to make a summation by taking the maximum value for each pixel RGB channel this way we won't introduce new colors in the image. The result can be seen on Figure 37.

Figure 37 – Pixel-wise maximum of target image and overlay image RGB value

The pixel-wise maximum is the most alike to the reference image rendered by Blender, but just to make things sure let's compare it numerically.

To compare how much one image is alike another, we can calculate the Structural Similarity Index (SSIM). The SSIM measures the similarity between two images; it considers changes in texture, luminance, and contrast between the two images. The SSIM index ranges from -1 to 1, where 1 indicates perfect similarity. This is the code and the result of SSIM comparison:

```
from skimage.metrics import structural_similarity as ssim
import cv2

# Function to calculate SSIM between two images
def calculate_ssim(imageA_path, imageB_path):
    # Load the two images
    imageA = cv2.imread(imageA_path)
    imageB = cv2.imread(imageB_path)

    # Convert the images to grayscale
    imageA_gray = cv2.cvtColor(imageA, cv2.COLOR_BGR2GRAY)
    imageB_gray = cv2.cvtColor(imageB, cv2.COLOR_BGR2GRAY)

    # Compute SSIM between two images
    score, _ = ssim(imageA_gray, imageB_gray, full=True)
    return score

# Paths to the images
reference_path = 'LF_on_LTV_by_Blender.jpg'
composite_image_path = 'composite_image.png'
maxed_image_path = 'maxed_image.jpg'

# Calculate SSIM for each pair of images
ssim_composite_reference = calculate_ssim(composite_image_path, reference_path)
ssim_maxed_reference = calculate_ssim(maxed_image_path, reference_path)

print(f'SSIM alpha composite vs Blender reference: {ssim_composite_reference}\n'+
      f'SSIM pixel-wise maximum vs Blender reference: {ssim_maxed_reference}\n')
```

```
SSIM alpha composite vs Blender reference: 0.9344458473211983
SSIM pixel-wise maximum vs Blender reference: 0.9656581505718361
```

The previous code compares the images on grayscale versions of the images.

The SSIM can be applied to each RGB color channel as well and we can take the average:

```
# Function to calculate SSIM between two color images
def color_ssim(imageA_path, imageB_path):
    # Load the two images
    imageA = cv2.imread(imageA_path)
    imageB = cv2.imread(imageB_path)

    # Compute SSIM over the 3 channels (color) and average them
    score = (ssim(imageA[:, :, 0], imageB[:, :, 0]) +
             ssim(imageA[:, :, 1], imageB[:, :, 1]) +
             ssim(imageA[:, :, 2], imageB[:, :, 2])) / 3
    return score

# Calculate SSIM for each pair of color images
color_ssim_composite = color_ssim(composite_image_path, reference_path)
color_ssim_maxed = color_ssim(maxed_image_path, reference_path)

print(f'color SSIM composite vs Blender reference: {color_ssim_composite}\n'+
      f'color SSIM pixel-wise max vs Blender reference: {color_ssim_maxed}\n')
```

```
color SSIM composite vs Blender reference: 0.9205018374621772
color SSIM pixel-wise max vs Blender reference: 0.9567464762704648
```

Considering these results we recommend saving the LF-only images in .jpg files and then applying this pixel-wise maximum operation on target and overlay images.

Our experiment results and the python code used is available on the GitHub repository notebook directory.

10 Further improvement possibilities

These are the most important improvements that can be added to the LFW toolkit:

- Adding a GUI
  Instead of asking the user to enter commands in Blender python console and instead of parametrization of the command line we could develop a Blender menu for these functions. This is of less importance, because the users of these scripts are presumably familiar with handling the command line.
- Automate the range editing
  This could be automated, so after the user composed and saved the LF properties into a JSON file, it can continue with modifying the LF effect and subsequent saving would automatically generate the ranges in the JSON file.
- Color space range
  The colors are currently stored in the RGB space, and this can be problematic e.g. if the user wants to have an LF effect in the range of green and blue then the whole range of RGB values should be specified, and the random selection of RGB values will generate unwanted colors. Handling the color values in the HSV space could be a solution for this problem.

11 Conclusion

The development and implementation of the Lens Flare Wizard (LFW) toolkit represents a significant advancement in the field of Computer Vision (CV). By addressing the challenge of lens flare (LF) artifacts, which frequently degrade the quality of images and hinder the performance of CV models, the LFW toolkit offers a robust and efficient solution. This Python-based toolkit facilitates the automated generation and application of realistic LF effects on images. The key achievement of LFW lies in its ability to enrich training datasets, enabling CV models to not only recognize but also effectively remove LF artifacts. This enhancement is crucial for the practical application of CV across various sectors, such as autonomous vehicles, surveillance, where visual accuracy is paramount.

Our findings indicate that LFW can produce lens flare effects with impressive efficiency, taking only 0.2 to 0.8 seconds per image, depending on factors like image resolution and flare complexity. This capability is a testament to the toolkit's potential in scaling and improving the training process of CV models. The LFW's contribution extends beyond mere artifact removal; it serves as a critical tool in advancing the reliability and applicability of CV technologies in real-world conditions, where unpredictable visual anomalies are common. This research not only addresses a specific technical challenge in image processing but also opens avenues for further innovations in enhancing the robustness of CV systems against a variety of visual distortions. The introduction of LFW marks a significant stride in the ongoing effort to refine computer vision technology, ensuring its readiness and reliability in diverse and challenging operational environments.

12 References

Hullin, M., Eisemann, E., Seidel, H.-P., & Lee, S. (2011). Physically-Based Real-Time Lens Flare Rendering. *ACM SIGGRAPH 2011 papers*. doi:10.1145/1964921.1965003

Lee, S., & Eisemann, E. (2013). Practical Real-Time Lens-Flare Rendering. *Computer Graphics Forum*. doi:10.1111/cgf.12145

Pekkarinen, E., & Balzer, M. (2019). Physically based lens flare rendering in "The Lego Movie 2". *Proceedings of the 2019 Digital Production Symposium.* doi:10.1145/3329715.3338881

Walch, A. (2018). Lens flare prediction based on measurements with real-time visualization. *The Visual Computer*. doi:10.1007/s00371-018-1552-4