Faculty of
Computer Sciences

Marin Philippe

A Tool to Render Them All: Facilitating the Creation of New Dataset Elements for Space Object Detection and Pose Estimation

Supervisors:

Prof. Mathieu Salzmann (EPFL)

PhD. Andrew Price Lawrence (EPFL)

Laboratory:

Computer Vision Laboratory (CVLAB)

05.01.2024

Contents

Chapter 1

Introduction

The main concern in today's machine-learning era is data availability. When we want to create a computer vision model, we desperately need images. However, the issue with pose estimation of space objects in motion is that simple space object photos are hardly available in open-source without mentioning unprocessed images. The challenge the previous group answered was bridging the gap between 3D models and realistic photos and motions. The issue we then had was the amount of work each model took from the 3D model download to the creation of the first rendering of an image, making the dataset at the time difficultly scalable. My main objective was to reduce the work and time needed on models to allow any person with 3D models and a minimum of knowledge of Blender to import and use its objects. I approached this challenge by improving its usability to allow myself to understand the issues in the implementation and then understand the importing process, restraining it to the minimum needed. Then I made the code as soft-coded as possible so that each object added and processed has its own generated information automatically for the following steps. And finally, I concentrated on optimising the processes needed to allow true scalability and fast testing.

Chapter 2

Usability

Usability was the primary focus of the project, as it would be easier to test quickly and without errors for the rest of the project. The structure and execution of scripts were organised, so no input from the user was needed in scripts. This was brought about by the formatting of certain crucial files to allow the rest of the scripts to read and extract the information needed

The first challenge was having a coherent architecture assembled from three different projects that came from this:

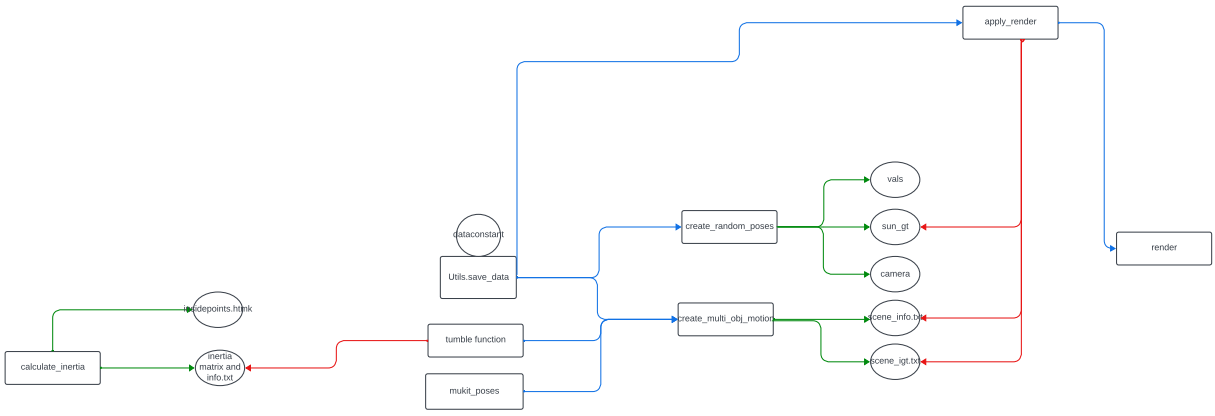

Figure 2.1: Flowchart start

The biggest issue with this structure is that most scripts had hardcoded inputs that would have been very time-consuming to change in each file. The work then consisted of taking away all of those hard-coded inputs and reading from the intended files directly, with the biggest addition being the .sh files, which allow easy command-line inputs of not more than 2 to minimise possible errors from users and frustration. The new structure is made so that it is easy for each script to find the needed file and information. Here is the broad file structure:

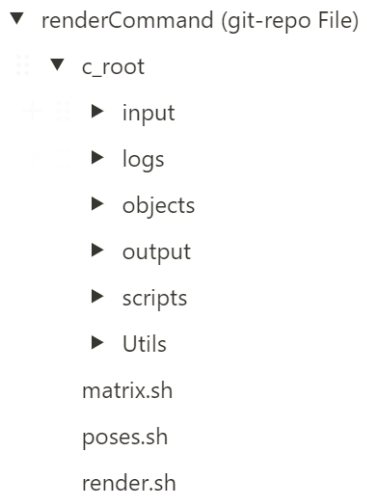

Figure 2.2: file organisation

2.1 Command Line

Another issue was the obligation to use the Blender GUI, which is very heavy. Imposing the command-line input system, would retrain errors from users and not be as dependent on past operations for the blender files to work. The GUI needs to have each file uploaded in some manner to its script system, making it very tedious. Then, it needs to run each script in the right order. Now the three commands impose the order and the inputs.

2.2 The New Commands

Emancipating ourselves from the Blender GUI because of the usability and performance issues meant we had to create an API that would be very functional and without flaws. This is why we decided early .sh commands would be the easiest way to guarantee this. The set of 3 commands allows, respectively: the creation of the matrix of 1 or all objects; the creation of motion or poses of 1 object or all; and the rendering of 1 motion or all motions in the input file; and adding backgrounds during processing or in post-processing (backgrounds are still very rudimentary).

```
./matrix.sh {object_name}or(-a)
./pose.sh {object_name}or(-a) {pose_id}
./render.sh ({object_name} {pose_id}) or (-a) (-b)/(-bu)
```

It is best to simply use the "-h" flag to understand the command before using it.

The new flowchart looks like this:

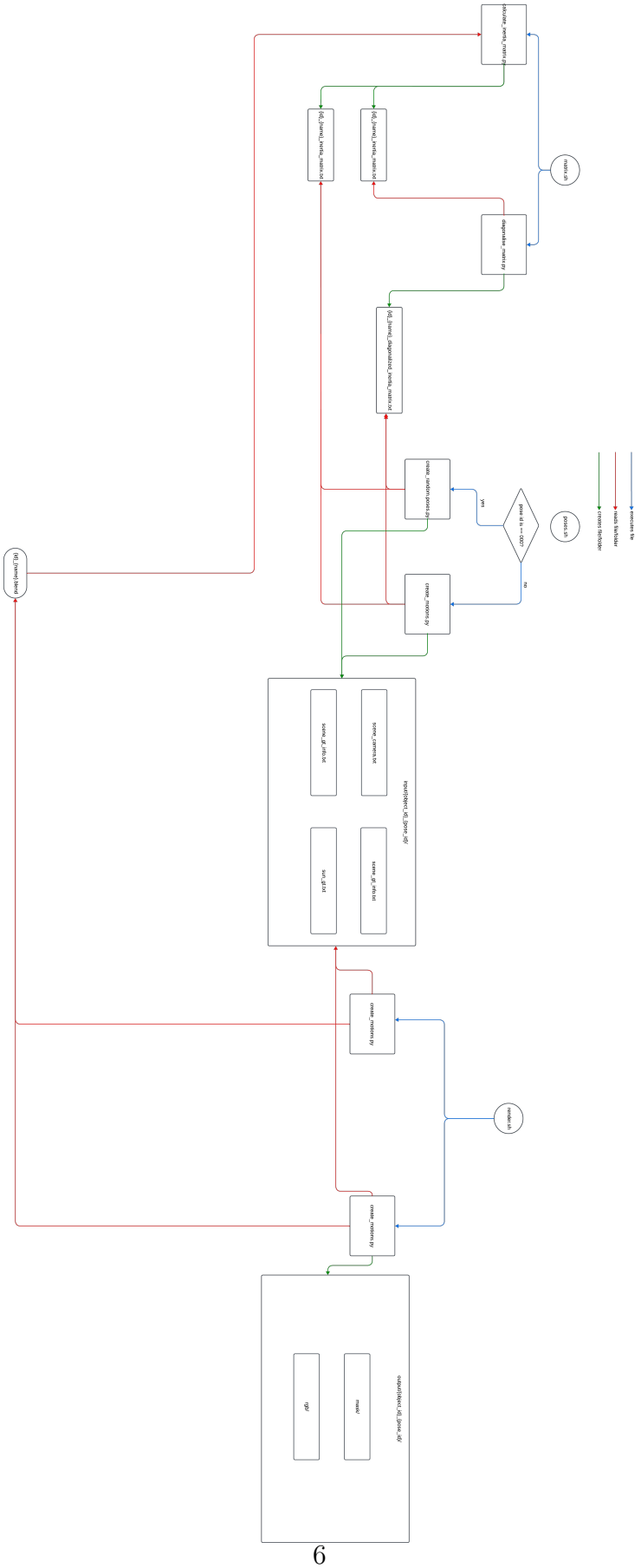

Figure 2.3: New Flowchart

Chapter 3

Import

To augment the number of objects, we imported many models to the dataset, with the authorisation to get the new ones from the https://scifleet.esa.int/model-downloads site from the ESA models.

3.1 Format

FBX is the best way to import blenders for this use, as Blender has a way to import them and they keep an excellent texture.

3.2 Parameters

Marc Pitteloud from the precedent project did a great job carefully choosing the many parameters of Blender to fit the objective of the dataset. For this reason, I imported the objects into the files he created, which kept the same parameters and composition trees. The composition trees had to be flipped for the mask, as it was asked to have the objects in white and the background black.

3.3 The Importing process

The importing process is quite quick, yet the longest time spent on those imports was the rescaling of the models, as some were several kilometres wide. So we had to find information on the size of each object to rescale the object; this could be optimised. There is a way to create blender files from FBX files in Python and the Blender API, but the issue of calling and renaming could be fixed with a simple data sheet as input or even a set of questions asked by the script to the user that would take 20 seconds to answer if the person had some answers and would let the code rescale and rename the object in the correct format for use. This would save some time in using the import system of the blender for the user and possibly creating datasheets for each model. We did not concentrate on this part as importing objects was not the hardest part of the project and not the part where the most time was spent, so using Amdahl's law, we concentrated on other parts that took more time.

3.4 New imported models

Here are some of the 42 models:

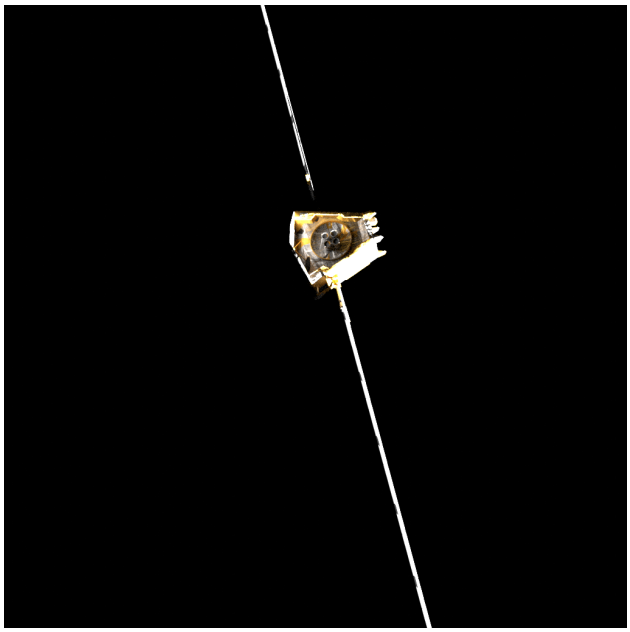

Figure 3.1: BepiColombo MTM

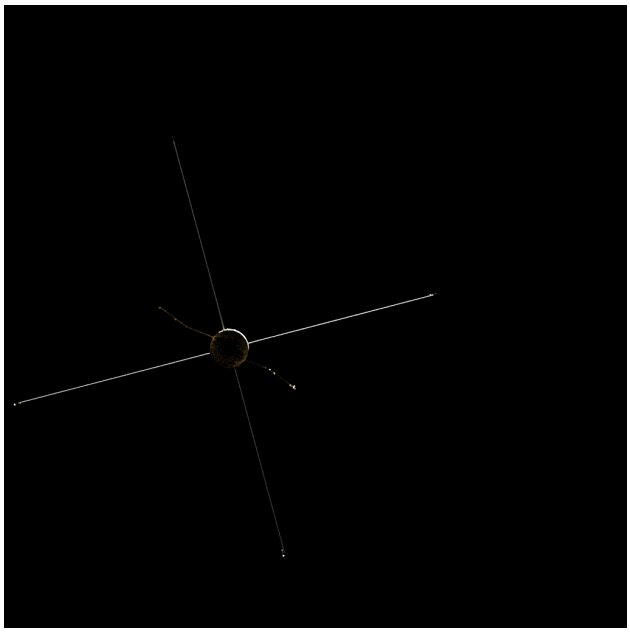

Figure 3.2: Cluster

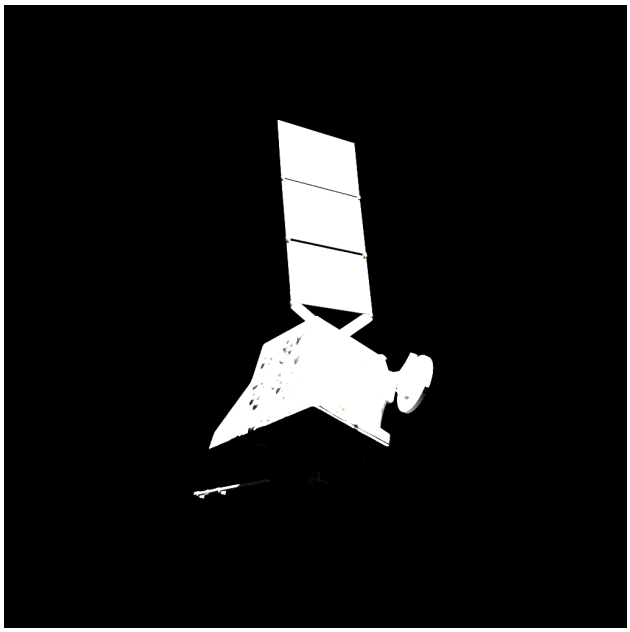

Figure 3.3: BepiColombo MPO

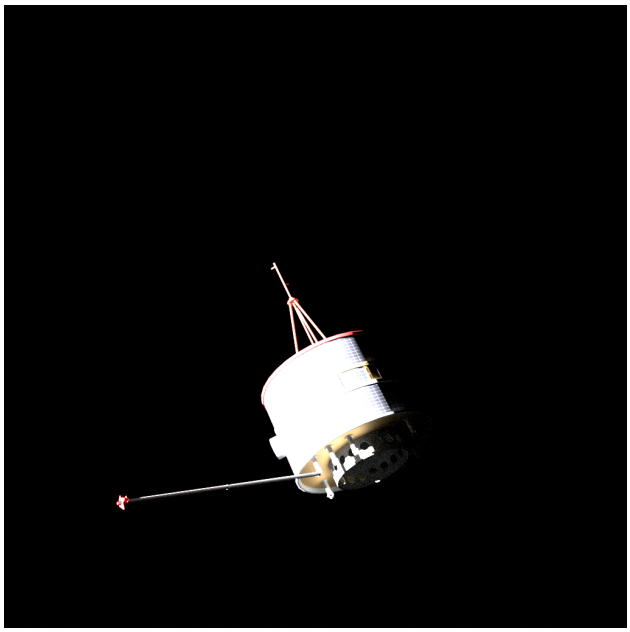

Figure 3.4: Double Star

Figure 3.5: Comete 67P/TG

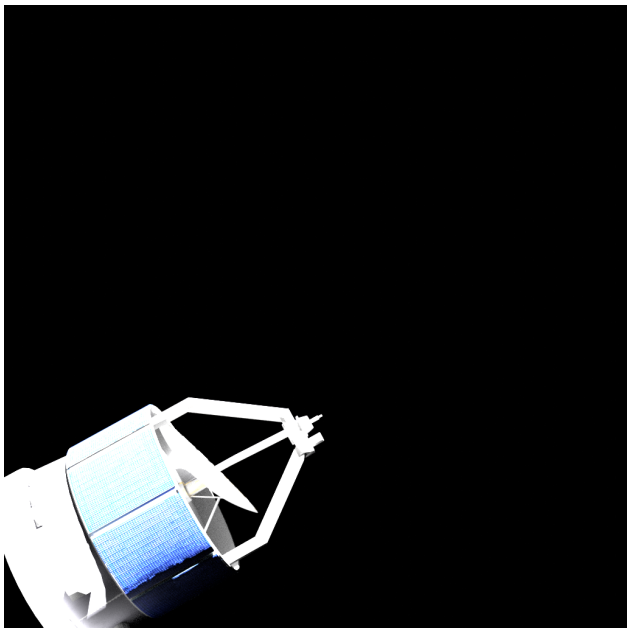

Figure 3.6: Giotto

Chapter 4

Processing

The previous project was heavily tailored for the four objects imported by the previous team; this would have demanded a huge amount of time. The most important thing is to have absolutely no constants that are linked to objects. This was greatly facilitated by the usable approach; the script could read in the associated file to the object the needed information, and non-readable values are now calculated on the spot from the readable ones.

4.1 Motions and poses

For the creation of poses and motions, the script needs a diagonalised inertia matrix associated with the object and initial position. The objective was to get as many uniform positions and directions in the frustum as possible.

4.2 Randomness

For the creation of motions, the uniformity of the motions in the frustum was one of our main targets. Three parameters (initial position, direction, and velocity) would define this uniformity.

At first, the method used was complete uniformity and independence of each parameter. This gave us, with the camera settings chosen by the previous team and Dr. Price, 50 motions of the same object, a mean of 46.3 frames, and a standard deviation of 8.6 frames per motion. With our target being:

$Mean - StandardDev > 100$

This was not the correct method, so we decided to try another approach.

The goal was to get the points closer to the border of the frustum and have a higher chance of going away from the wall than straight to it. The method then went to the normal distribution in the middle of the frustum, where we would take points randomly to set them as the target for the initial point, then get the direction from them using simple vector maths, and finally normalise to then apply a random positive velocity.

Below is the distribution to arrive at the objective above:

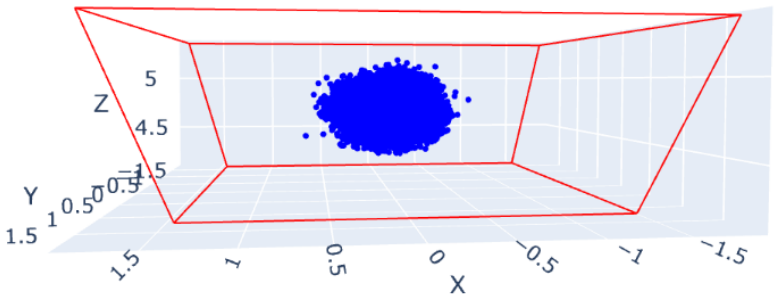

Figure 4.1: distribution of the target points to comply with requirements

However, the results were not convincing, as the distribution was too centred and gave motions that imperatively passed through the centre of the frustum.

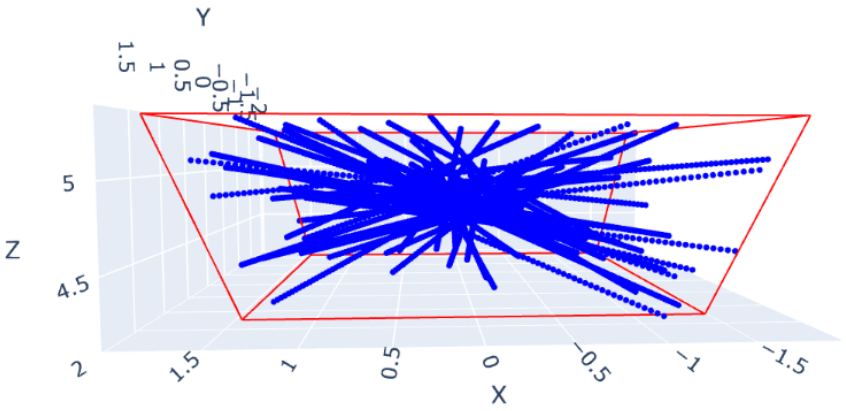

Figure 4.2: 50 motions using the normal target method along the Y axis

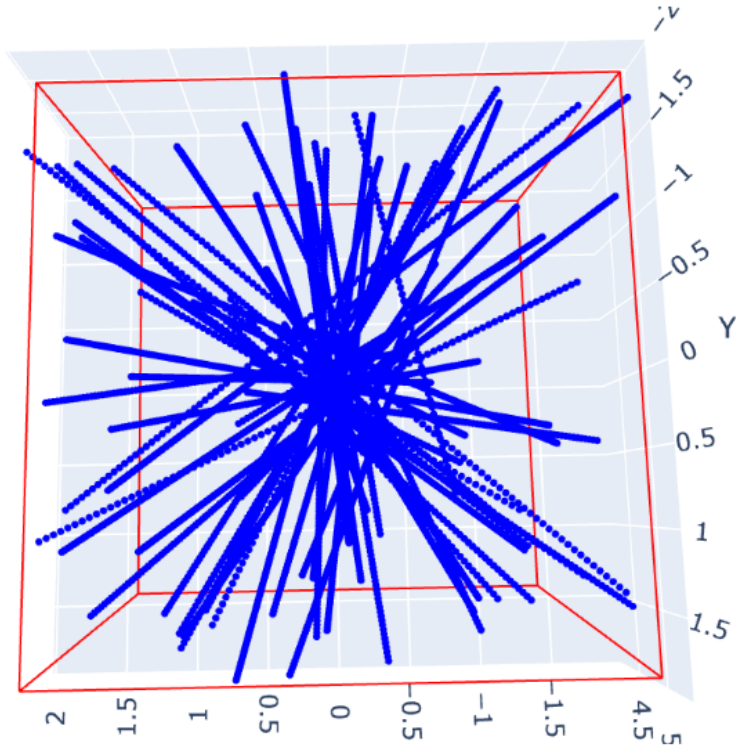

Figure 4.3: 50 motions using the normal target method along the Z axis

For many reasons, this did not work, but most of all, the dependence of direction on position would create unwanted patterns. If done correctly, we may have found a way to deter the point from going to the nearest wall, but we decided to try another approach from

scratch.

This approach had the same principle of creating a target point for each initial position that is in the frustum and setting the speed so that the object hits the target point at 100 frames and continues to create other frames until 400 or hitting a wall. Nevertheless, we changed the size of the frustum, and the Z axis became much larger than the X and Y axes; thus, our uniform distribution of target and initial position had a much bigger standard deviation on the Z axis, making the objects have a bias to travel along the Z axis.

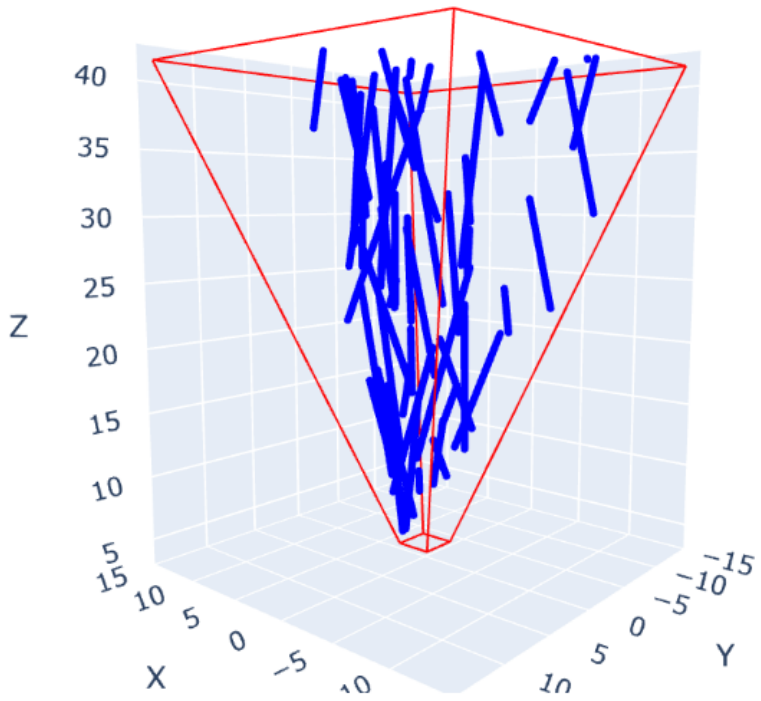

Figure 4.4: 50 motions using the uniform target method along the Z axis

Although the X and Y axis have the same length and so the same standard deviation, we have seemingly uniformity and independence.

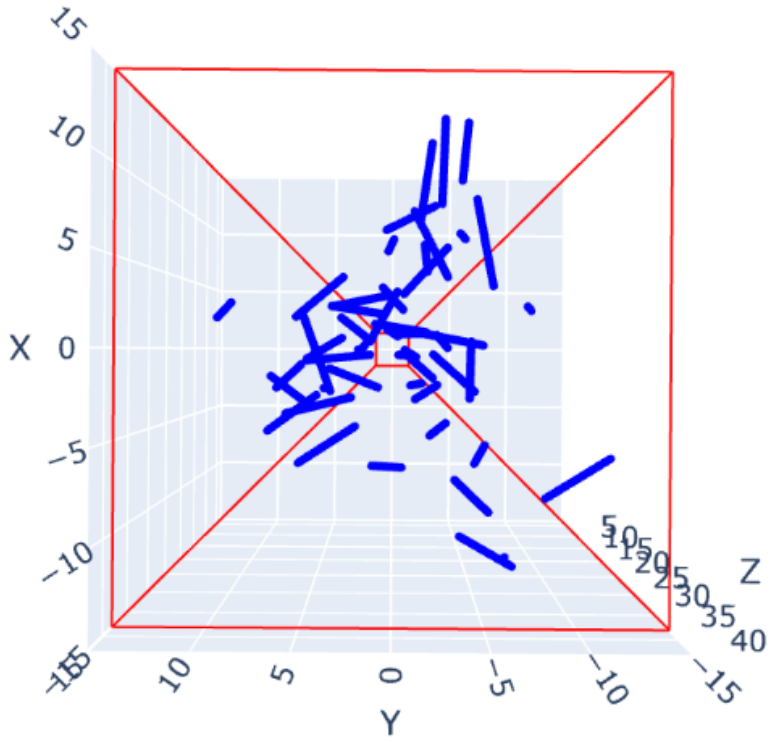

Figure 4.5: 50 motions using the uniform target method along the Z axis

The last method we tried and chose was going back to total uniformity but keeping the same velocity-restraining idea. Here we would find when the object intersections with frustum boundaries, extract that point, and set the velocity so that it hits the wall at frame 100. The frustum is then divided in this manner:

```
side1_norm = [0,0,1]
side2_norm = [0,0,-1]
side3_norm = [math.cos(math.radians(fov/2)),0,math.sin(math.radians(fov/2))]
side4_norm = [-math.cos(math.radians(fov/2)),0,math.sin(math.radians(fov/2))]
side5_norm = [0,math.cos(math.radians(fov/2)),math.sin(math.radians(fov/2))]
side6_norm = [0,-math.cos(math.radians(fov/2)),math.sin(math.radians(fov/2))]
```

So then we want to find the coefficients "t" of the direction to hit all the planes. We use the

following formulas to calculate "t" for each plane. ("D" is found using a known point in the plane):

$\mathbf{n} = \langle A, B, C \rangle$

$\mathrm{Ax} + \mathrm{By} + \mathrm{Cz} + \mathrm{D} = 0$

$\mathrm{P}_{intersection} = P_{origin} + t \times \mathbf{v_{direction}}$

$\mathrm{A} \times P_{intersection} + B \times P_{intersection} + C \times P_{intersection} + D = 0$

We then have a list of coefficients "t" and we want the smallest positive "t" that corresponds to the first intersection in the direction of the vector to collide with the border. Now we can find the exact intersection point and thus find the velocity. This gives us a completely uniform and independent initial position and direction; nonetheless, the speed is dependent on them, so in a future project, this might be addressed and looked at, if not problematic.

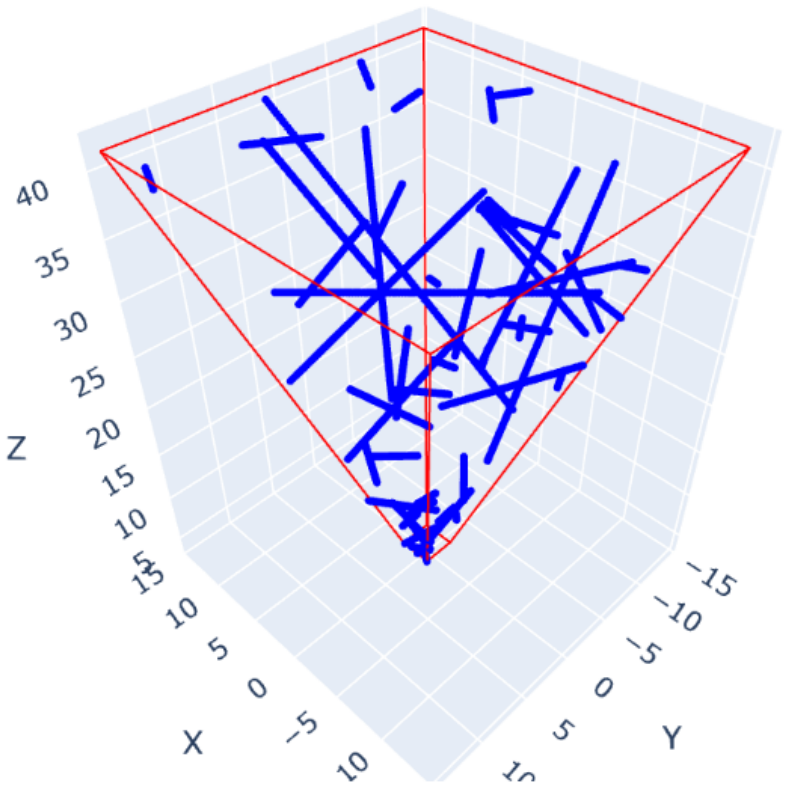

Figure 4.6: 50 motions using the complete uniform method

4.3 Visualisation

To be able to make those choices, we had to have an idea of our motions in 3D. It was extremely hard to assume patterns and positions relative to the borders just by reading the successive coordinates, so we created a small script to create the shape of the frustum automatically and how to read the motions and plot them in 3D.

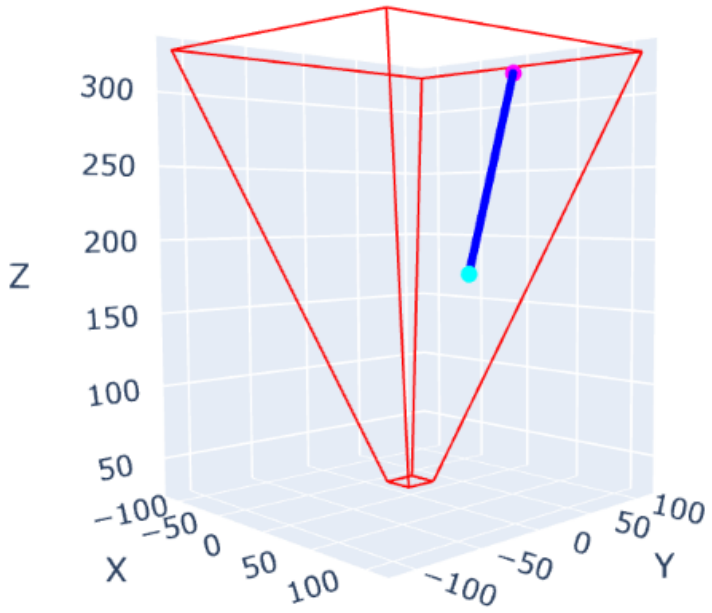

Figure 4.7: Motion of the BepiColombo Object in the Frustum

Chapter 5

Optimisation

The inertia matrix calculation was the bottleneck of the project, as it was needed to calculate the motions to test the object and see how the code accepted it. The issue was that not all objects were made the same, and objects with antennas that had a huge bounding box, but did not fill it would take several hours on the workstation; for example, the satellite "cluster" took 24 hours before a result. There was also an issue with the rendering, which took a lot of time and had to pass by the GUI, which slowed the process down by a lot.

5.1 Inertia matrix

The original code plotted points randomly in the bonding boxes and checked with 7 random rays if the point was in the object until we had 100'000 points inside. For objects like "CHEOPS" this is not an issue, but for "cluster" and many possible others, it was a problem. However, the library "Pymeshlab" allowed us to, in not more than 10 seconds, get the 100'000 in a CSV. Then the bottleneck discovered was the actual calculation of the matrix, which is a while loop, which in Python is very slow. Keeping the same algorithm used by the precedent project, I simply read the CSV file in NumPy and do the variable assigning with NumPy, allowing it to be parallelised by the library and thus dividing the work done for the calculation to the multiple cores we have on the workstation.

Here is how we extract the points from the object:

```
bpy.ops.export_scene.obj(filepath =os.path.join(obj_path,f"{object_name}.obj

#create mesh
ms = pymeshlab.MeshSet()
ms.load_new_mesh(os.path.join(obj_path,f"{object_name}.obj"))
#create cloud of points of mesh using montecarlo sampling
ms.apply_filter('generate_sampling_montecarlo', samplenum=100000)
vertices = ms.current_mesh().vertex_matrix()
```

Here is a section of code where NumPy replaced a loop:

```
num_inside_points = vertices.shape[0]
shifted_vertices = vertices - global_approx_com

# Extract individual coordinates
x, y, z = shifted_vertices[:, 0], shifted_vertices[:, 1], shifted_vertices[:,

# Calculate the contributions to the inertia matrix
I_local = np.zeros((3, 3))
I_local[0, 0] = np.sum(y**2 + z**2)
I_local[1, 1] = np.sum(x**2 + z**2)
I_local[2, 2] = np.sum(x**2 + y**2)
I_local[0, 1] = I_local[1, 0] = -np.sum(x*y)
I_local[0, 2] = I_local[2, 0] = -np.sum(x*z)
I_local[1, 2] = I_local[2, 1] = -np.sum(y*z)

I_local *= total_mass / num_inside_points
```

This simple code exploits the newly found inside points of the object.

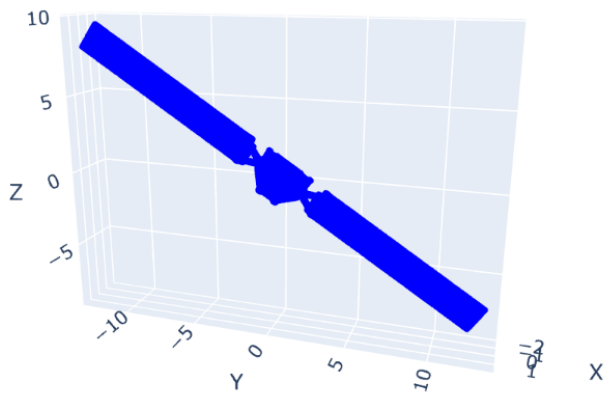

Figure 5.1: Plotted points of BepiColombo object using Pymeshlab

5.2 Rendering

Passing by the GUI is costly, thus the decision to have everything from the command line, getting the most out of the workstation, was clear. It is also possible to change the number of samples we have for each frame, allowing faster testing.

5.3 Project

It is now possible to create an image of any FBX file from the download of the first image in around 5 minutes, allowing many tests and a very expandable dataset.

Chapter 6

Conclusion

Finally, the tool is intended to import and render one object as fast as possible. There isn't a possibility yet for renders with multiple objects in the frame. The most important feature of this project was to keep it simple to enhance usability; giving too much freedom to the user, especially in an API only visible from the command line, is crucial. With so little visual feedback, giving rapid results was a point of focus, driving us to greatly optimise the steps. For now, the API works with four steps that are separable and can take exterior inputs between them. There are things to work on:

- Adding backgrounds was not tested extensively
- We don't have a lot of different backgrounds
- Importing is still very primitive and would require a bit of knowledge of Blender beforehand and some research on models
- The commands are very rigid; they don't allow errors and don't help the user if there is an error on their side
- The code needs to be cleaned and tested more extensively, as a lot of tests were made, and some old unused codes and methods are still probably inside

- To find the parameters people would want to change and their location in the code, most are in the data constants, but as we are a total of 5 people on the same code and not simultaneously, different conventions and styles were used. Therefore, for a third party, even if the code is very well documented, it will be tricky to understand