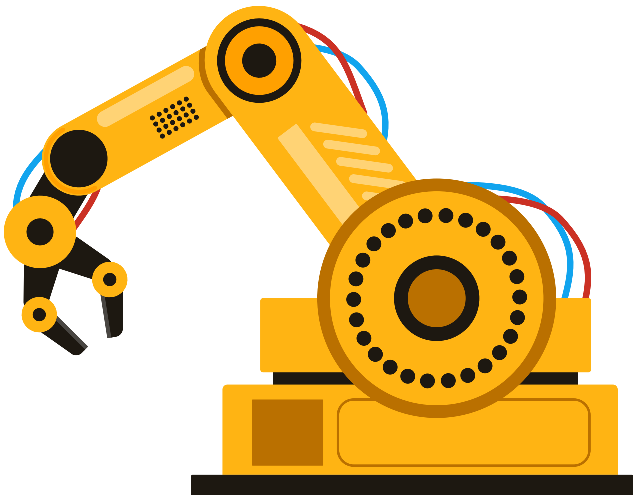
POLITECNICO
DI MILANO

DF_01 REPORT



Presented To

PROF:- MARCELLO URGO

PROF:- WALTER TERKAJ

Presented By

Assem Meselhy Melegy Shabayek

Mahmoud Samy Rezk Elashry

Table of Contents

Abstract

In this report we are going to explore the utilization of digital factory techniques, and to develop a Python code that effectively converts any URDF file into a JSON file.

Thereby enabling the precise definition of any robot as a crucial component of the digital twin ecosystem. Specifically, we will be examining the Comau_NJ220 robot arm, which is commonly found in manufacturing labs and is used for the automatic disassembly of batteries.

Our objective is to provide a comprehensive representation of robots, including their hierarchical structure, joints, links, and 3D visualization, to enhance the overall efficiency and performance of the digital twin system.

Introduction

Industry 4.0

The Fourth Industrial Revolution, also known as Industry 4.0, has been making significant strides in recent years as industrial companies strive to integrate advanced technologies such as artificial intelligence, the Internet of Things, and big data analytics into their manufacturing and industrial processes. The ultimate goal of Industry 4.0 is to create "smart factories" that are highly efficient, flexible, and able to quickly adapt to changing market demands. To achieve this, Industry 4.0 is characterized by automation, data exchange, decentralization, and the possibility of self-optimizing systems.

One of the most crucial technologies enabling Industry 4.0 is digital twin technology. A digital twin is a virtual representation of a physical object, process, or system. It is a digital replica of the real-world entity that is used for simulation, analysis, and optimization. This technology allows organizations to create a virtual copy of an asset, process, or system, and then use it to test and optimize the performance of the physical version. This can include monitoring and analyzing data, simulating different scenarios, and making predictions about how the physical version will behave in the future. In this report, we will focus on using digital twin technology to represent robots in a smart factory environment and how it can be implemented in the industry.

The Virtual Learning Factory Toolkit (VLFT)

is a set of existing digital tools to support advanced engineering education in manufacturing. The aim of the VLFT is to bring back to the engineering students the results of research activities in the field of digital manufacturing related to the modeling and analysis of a manufacturing system, virtual and augmented reality, as well as the role of the human workers in a factory.

VLFT grounds on a common knowledge base that consists of data model and data repository relying on semantic web technology. In principle, any digital tool can be integrated in the VLFT framework if it is possible to access and modify its internal data structures (e.g. via exchange files or an API), thus creating data flows with the knowledge base, possibly in an automated way.
VEB.js is one of these digital tools, and this is how we want to visualize our robots.

Virtual Environment based on Babylon.js

VEB.js (Virtual Environment based on Babylon.js) is a powerful JavaScript library that allows developers to create interactive and immersive 3D virtual environments for web applications. It is built on top of the powerful open-source 3D engine Babylon.js, which is known for its wide range of features for creating realistic 3D environments using JavaScript and WebGL. VEB.js extends the capabilities of Babylon.js by providing additional tools and APIs such as support for physics, collision detection, user interactions and pre-built assets and components that can be easily integrated into a virtual environment. Additionally, VEB.js may also include a visual editor similar to Babylon's "Sandbox" allowing developers to quickly create 3D scenes without writing code. Overall, VEB.js is a powerful tool for creating interactive 3D web experiences.

URDF Files

To define a robot we need to know its **URDF** file which is (Unified Robot Description Format) is an XML format for representing a robot's structure and kinematics. It is used to describe the physical layout of a robot, including its joints, links, and sensors. URDF files are often used in robot operating systems (ROS) to define the structure of a robot and its components, as well as its kinematic and dynamic properties. URDF files are commonly used to create simulations of robots and to generate code for controlling the robot's movement. They also allow for integration with other software tools and packages such as physics engines, visualization tools, and inverse kinematics solvers.

From this URDF we can extract all the data we need to define our asset according to the schema of the gitbook documentation ".JSON"

JSON Files

(JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is a text format that is completely language-independent but uses conventions that are familiar to programmers of the C family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.

JSON is often used to transmit data between a server and a web application, as an alternative to XML. JSON data is represented as a collection of key-value pairs, similar to a dictionary in Python or a HashMap in Java. JSON is also commonly used to store data in NoSQL databases and as an alternative to XML in the configuration files.

The .JSON schema to define the asset in the factory model consists of three root properties:

1- **context**: - definition of context setup.

2- **Scene**: - definition of the 3D scene

3- **assets**: - detailed definition of assets that are included or not in the scene. Assets not included in the scene are models/templates that are referenced or could be later instantiated in the scene.

In context you are required to define all this **(The unit of measure scale-Zup or Yup-Path of the repository)**

For the Scene, it is an array consisting of asset IDs that are included in the 3D scene.

For The Assets it’s an array contains items characterized by the following properties:

"**id**": unique identifier of the asset [required];

"**type**": The type of the asset, i.e., OWL class of the Factory Data Model it belongs to [required]

"**model**": ID of the model of the asset (if existing), e.g., the model of a machine tool that is described in a catalog. In turn, the model can have a model. Please refer to the Object Typing pattern.

"**representations**": Array of 2D/3D representations of the asset. Each item of the array may have the following properties:

"**file**": file path of the 2D/3D model representation, as relative to the RepoPath [optional]. The file can be available on a local or remote file system (see example), as any online repository accessible via HTTPS (see example). The “file” property can be used also as a reference to a specific component inside the hierarchy of a 3D model by adding a hashtag and the ID of the component to the file path (e.g., #componentID'). For instance, the “file” property will have the value "FileName.glb#nodeId" if it refers to a node with unique id "nodeId" inside a GLTF file named "Filename.glb".

"**unit**": Unit of measure to interpret a 3D representation (e.g. 0.01 stands for centimeter, whereas 1 stands for meter) [required if “file” is defined]

"**position**": The Position of the asset. If missing, the default value is [0.0,0.0,0.0];

"**scale**": The scaling of the asset. If missing, the default value is [1.0,1.0,1.0]. Scaling is defined independently of the unit of measurement of the 3D representation (cf. “unit” in “representations”);

"**rotation**": The rotation of the asset defined as Euler angles YXZ in radians, or quaternion or rotation matrix. If missing, the default value is the rotation of the corresponding node in the GLTF hierarchy (if available), otherwise [0.0,0.0,0.0] as Euler angles, [1.0, 0.0, 0.0, 0.0] as quaternion, or [[1 0 0], [0 1 0], [0 0 1]] as rotation matrix.

"**placementRelTo**": ID of the asset with respect to which the placement (position and rotation) is defined in relative terms. This means that a roto-translation must be applied with respect to the placement of the asset identified by the value of **placementRelTo**. This relation happens between nodes directly connected in a scene graph."**parentObject**": ID of the asset that is decomposed (it may be empty or missing) when an aggregated asset is represented. If a **parentObject** is defined, then **placementRelTo** is defined as equal to parentObject. However, if a **placementRelTo** value is defined, then it is not necessarily also the **parentObject**. For instance, machine components decompose a workstation, whereas a pallet doesn't decompose a conveyor.

This JSON file needs also another file holding mesh data to be able to visualize the robot, this file will have the extension of .GLB

GLB Files

A GLB file (.glb), which stands for “GL Transmission Format Binary file”, is a standardized file format used to share 3D data. Precisely, it can contain information about 3D scenes, models, lighting, materials, node hierarchy and animations. When you open a glb file format, you are able to visualize and interact with a complete 3D scene. This is why it is also known as the JPEG (image file format) of the 3D asset world.

What is a GLB file used for?

The glb-file is a fairly new format as it was introduced only in 2015 in order to represent GLTF files (.gltf) in a binary format, and not in a JSON format.

Project work

This project aims at exploiting digital factory technologies to instantiate a digital twin of a manufacturing lab containing robots and other devices for the automatic disassembly of batteries. The focus is on the complete representation of robots (hierarchy, joints, links, 3D).

INPUT: Description and 3D models of assets

Main ACTIVITIES:

1. Collect, check, and organize information about the lab. Input material: spreadsheet, updated STP file of the cell, draft/outdated GLB files of the assets. Focus on robots: 1) Comau_NJ220, 2) Kuka_KR50R2500

2. Develop a python script to convert a robot model from URDF/.xacro format to JSON format of the Digital Twin Environment

3. Conversion of .stl files into .glb/.gltf files for each robot link. Aggregation of .glb files into a single .glb file of the whole robot. Blender software program can be used for this task. Improve photorealism with Physical Based Rendering (PBR).

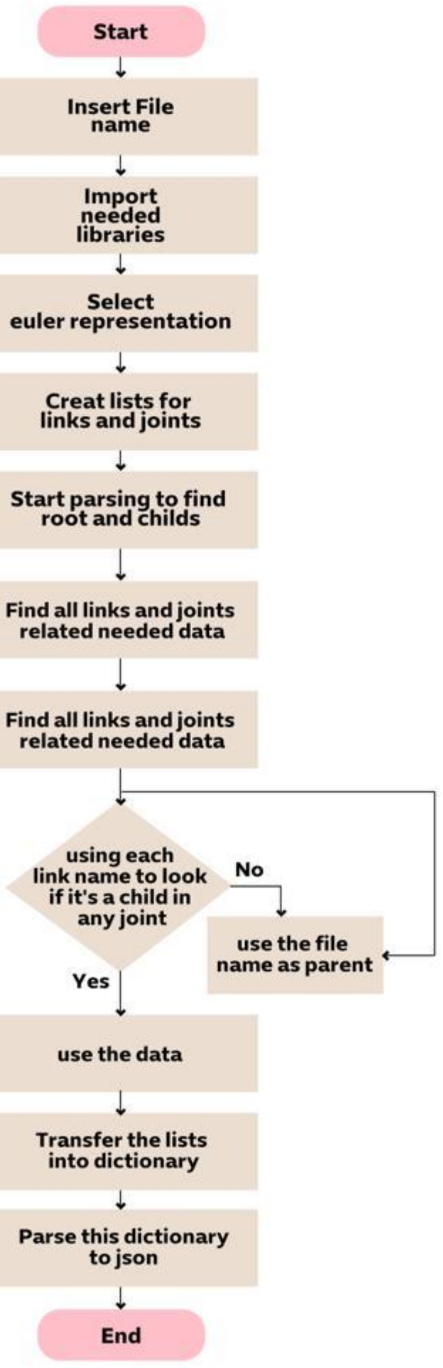
4. Update the digital model of the lab.

6. Write a report and documentation of the work done, including structured data and possible code/scripts.

URDFtoJSON Converter

In order to satisfy these requirements we worked on developing a python code able to convert URDF files to JSON files as per the previous requirements and as per the Json Schema.

Simplified flowchart for the URDFtoJSON Converter


Start
Insert File name
Import needed libraries
Select euler representation
Creat lists for links and joints
Start parsing to find root and childs
Find all links and joints related needed data
Find all links and joints related needed data
using each link name to look if it's a child in any joint
No
use the file name as parent
Yes
use the data
Transfer the lists into dictionary
Parse this dictionary to json
End

Used Libraries

The xml.etree library

A part of the Python standard library is used for parsing and manipulating XML data. The ElementTree class in the library provides a simple way to work with XML elements and their attributes, and can be used to read, create, and modify XML files. It also provides a way to navigate and iterate through the elements of an XML document.

Since URDF file is based on XML we can use xml.etree library to extract the data Since XML is a naturally hierarchical data format, a tree is the most appropriate visual to represent it. For this purpose, ET provides two classes: ElementTree and Element, each of which represents a single node in the XML document as a whole as a tree. Reading and writing to/from files as well as interactions with the entire document are often done at the ElementTree level. The Element level is where interactions with a single XML element and its children take place.

JSON in Python

Python has a built-in package called json, which can be used to work with JSON data.

Mathutils in python

A general math utilities library providing Matrix, Vector, Quaternion, Euler and Color classes. It’s used mainly for Euler angles convention conversion.

Blender in python

Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing.

This package provides Blender as a Python module for use in studio pipelines, web services, scientific research, and more

Mainly this library is used in reading the plain axes names.

Tkinter in python

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit. Import the Tkinter module.

Used for making pop-up lists.

The Python Code of URDFtoJSON Converter

```
#initiating the libraries

general_name = input("Please insert the name of the files: ")
xmlfile=general_name+".xml"
import json
import xml.etree.ElementTree as ET
import bpy
import mathutils
tree = ET.parse(xmlfile)
root = tree.getroot()
import supportive_functions as spf

blend_path = general_name+".blend"
blend_object_names = spf.get_meshes_names(blend_path)

main_filename = general_name+".glb"

eulers = ["XYZ", "XZY", "YZX", "YXZ", "ZXY", "ZYX"]
message = "Select input Euler angles convention:"
selected = spf.select_thing(eulers, message)
euler_convention = selected

#lists for saving links and joints
links =[]
joints=[]

#finding all robots names
robotname = root.attrib.get("name", main_filename[:-4])

#finding all the links names
for elm in root.findall(".//link"):
    linkname=elm.attrib["name"]
    links.append(linkname)

#finding all the joints names
for elm in root.findall(".//joint"):
    jointname=elm.attrib["name"]
    joints.append(jointname)

y=len(links)
assets=[0]*(y+1)

#looking for the data of each link name
x=1
for link in links:
    fname=0
    parent=0
    child=0
    assets[x]={"id":(robotname+"."+link)}
    #finding the file name
    for elm in root.findall(f".//link[@name='{link}']/visual/geometry/mesh"):
        fname=(elm.attrib["filename"])

    if fname!=0:
        message2 = "Select object representing"+link
        selected = spf.select_thing(blend_object_names, message2)
        object_name = selected
        assets[x]["representations"]=([{"file":(main_filename+"#"+object_name),"unit":1}])

        for joint in joints:
            #finding if there are any parents for the link
            for elm in root.findall(f".//joint[@name='{joint}']/child[@link='{link}']"):
                child=1
                #finding the parent name
                for elm in root.findall(f".//joint[@name='{joint}']/parent"):
                    parent=(elm.attrib["link"])
                #finding position and rotations of the link
                for elm in root.findall(f".//joint[@name='{joint}']/origin"):
                    rpy=(elm.attrib["rpy"])
                    xyz=(elm.attrib["xyz"])
                if xyz!=0:
                    xyz_list = [float(num) for num in xyz.split()]
                    assets[x]["position"]=(xyz_list)
                if rpy!=0:
                    rpy_list = [float(numz) for numz in rpy.split()]
                    xs, ys, zs = rpy_list[0], rpy_list[1], rpy_list[2]
                    rot_ZYX = mathutils.Euler((xs, ys, zs), euler_convention)
                    rot_mat = rot_ZYX.to_matrix()
                    rot_YXZ = rot_mat.to_euler('YXZ')
                    new_rpy_list=[(round((rot_YXZ.x),4)),(round((rot_YXZ.y),4)),(round((rot_YX
Z.z),4))]
                    assets[x]["rotation"]=(new_rpy_list)
                if parent!=0:
                    assets[x]["placementRelTo"]=(robotname+"."+parent)
                    assets[x]["parentObject"]=(robotname+"."+parent)
            if child ==0:
                    assets[x]["placementRelTo"]=(robotname)
                    assets[x]["parentObject"]=(robotname)
    x=x+1

assets[0]={"id":(robotname)}
assets[0]["representations"]=([{"file":main_filename,"unit":1}])
links = spf.add_word_to_list_items(links, robotname)
links.insert(0, robotname)
data=dict()
data['context']={"UnitOfMeasureScale": 1,"Zup": True,"RepoPath":''}
data ['scene']=links
data['assets']=assets

output_filename = xmlfile.replace(".xml", ".json")
x= json.dumps(data, indent=4)
with open(output_filename, "w") as outfile:
    outfile.write(x)
```

Steps of using the URDFtoJSON converter

Main Requirements

We must have 3 files, an xml file, a GLB file and a blend file to be able to generate the JSON file.

- all must be with the same name (case sensitive) except for the extensions at the end.

- in that folder, we need to put the python code files. (URDFtoJSON.py & supportive functions.py)

I. XML file preparation

The URDF file should be saved as xml, this can be done by opening the file using Visual Studio Code and use Save as and select .xml

II. GLB File preparation

We need to prepare the GLB file from blender file, and both will be the same in the final state.

1. Download the mesh files of the robot from the repository.

They can be available in 2 states: -

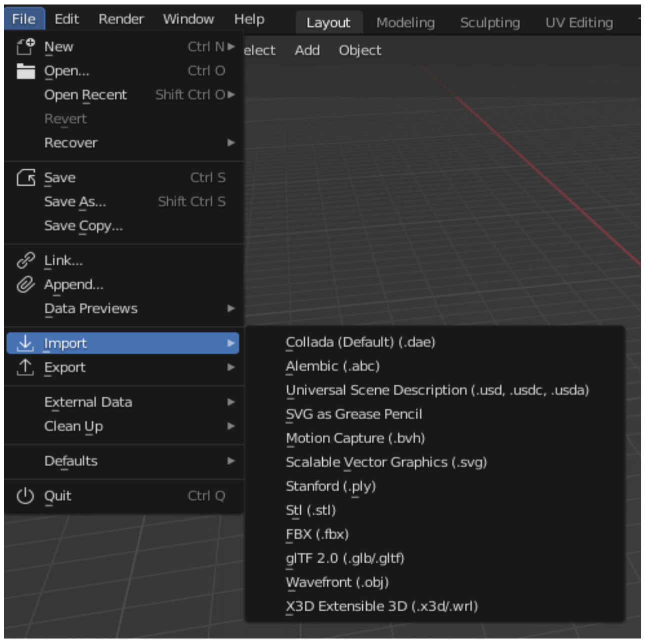
a) multiple files, each file represent one link.
b) single file representing all the robot.

The extensions can vary from being STL, GLB, DAE, OBJ …etc.
the only difference between these file extensions is how you import them to blender will be shown in the next step.

2. Import all the file/files to blender.

To import files into blender, you have to press File >> Import >> and select the extension similar to the available files.

In some cases you have to import them file by file, like the case of .DAE files.

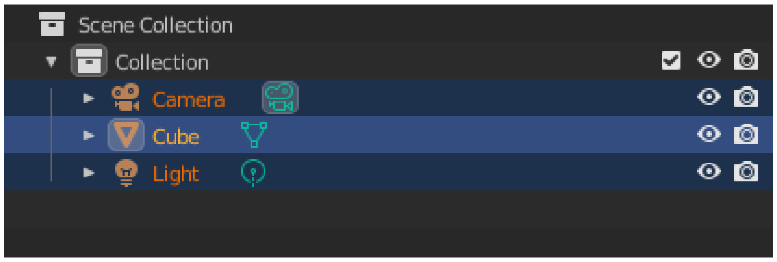


1- Screenshot to show importing files process.

3. Organize the hierarchy of the objects inside blender.

It should be better to first remove everything in the default workspace, like the camera, lamp, and the default cube.

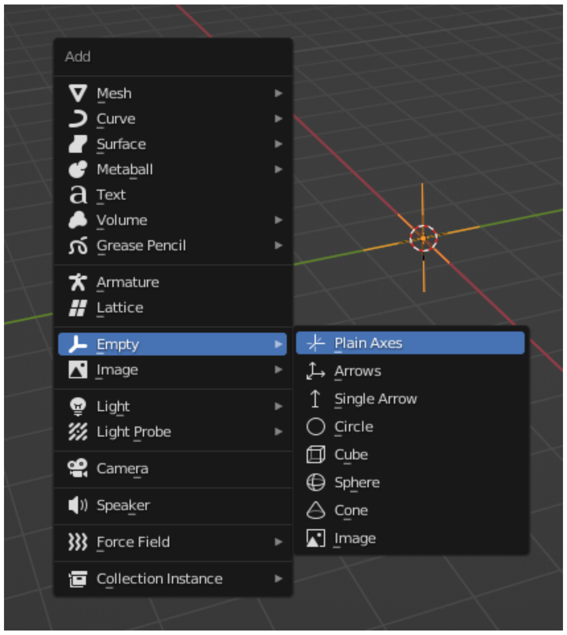
this can be done by pressing A (to select all) then press X to delete selected. (before importing the files). if any object has cameras or lamps, it will be better to remove them as well and keep the meshes only.



2- The default workspace objects to be deleter

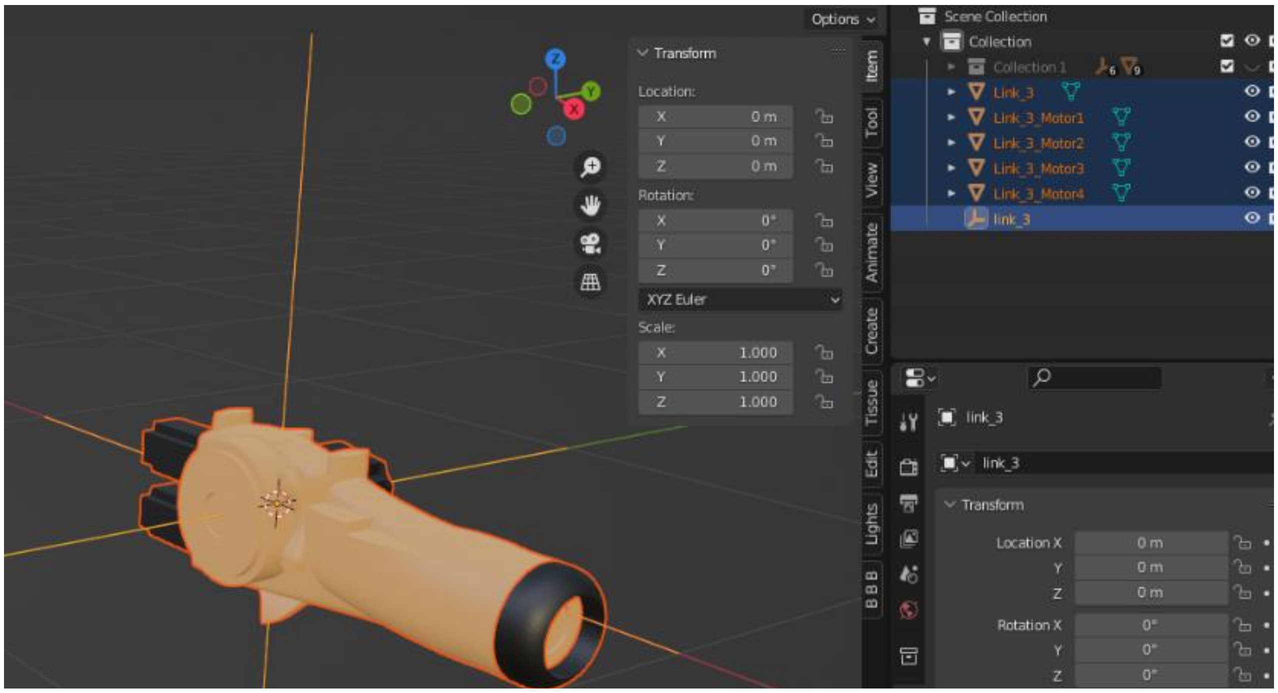
This is not a critical step, but just to have a clean start to not make any errors in the next steps.

a) Each link might have a single object or multiple objects, so we need to standardize this by having all the link's objects/object placed inside one object, this object will be an empty object of the type of plain axis.
this can be done by pressing Shift + A (for add), select empty and select Plain axes as shown in the picture.

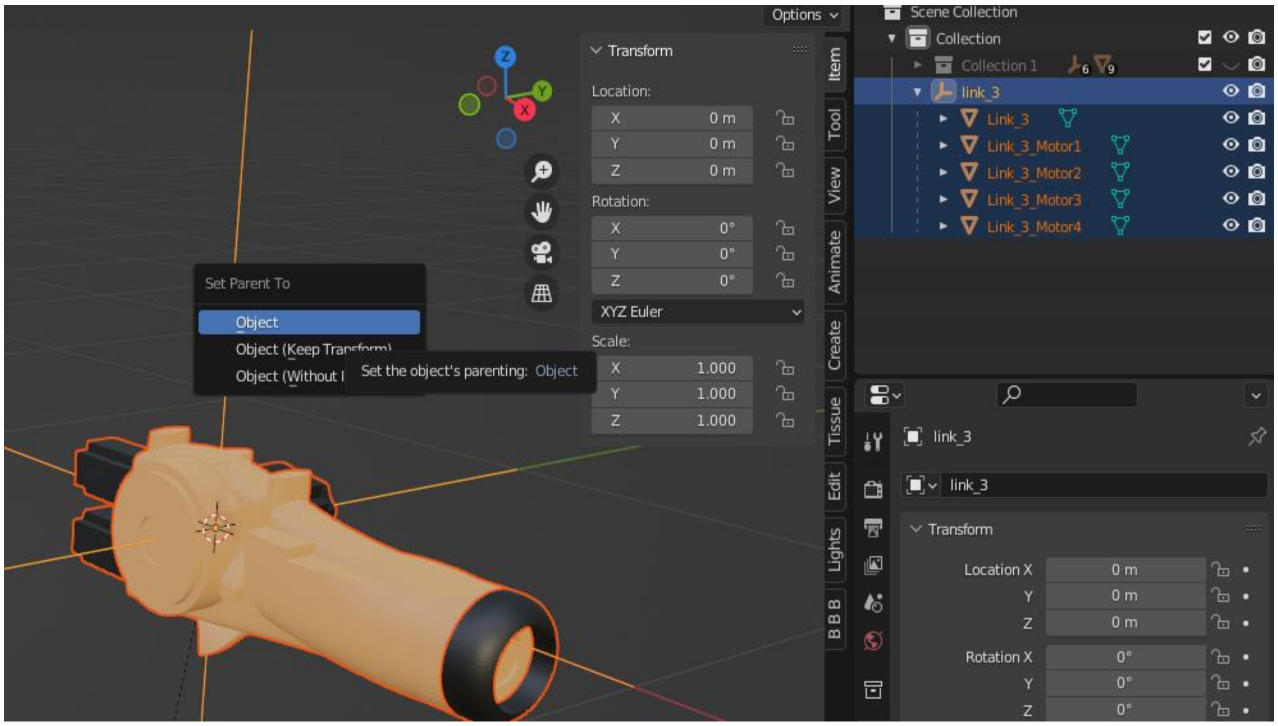


3- Adding plain axes to be a parent for object meshes.

b) The parenting can be done by selecting the object/objects to be a child for an empty plain axis, then select the plain axes while keeping the objects selected. (Press ctrl while selecting objects in the outliner for multiple selection) This order of selection is to make sure that when we press ctrl + p in the next step, The parent will be the last selected object (the plain axes).
the last part will be more yellowish than the other parts in the outliner (at the right).
then we press ctrl + p to have it as a parent for the other parts. (Select the first option *set parent to object).*
This Plain Axes should be named carefully as it will be the effective name in the next steps, that's why we need to try to make it as clear as possible.

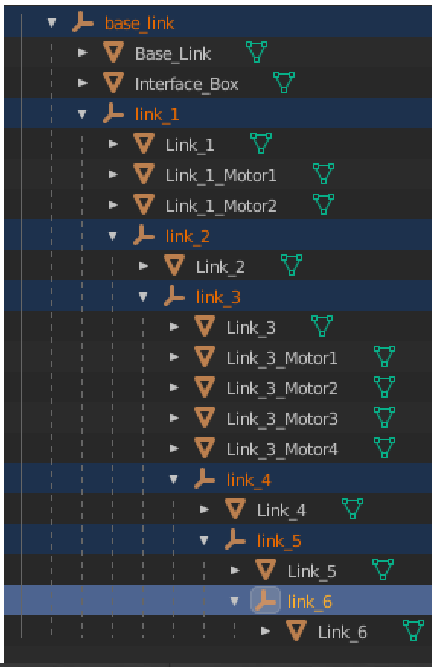


4- An example of selected objects before parenting.

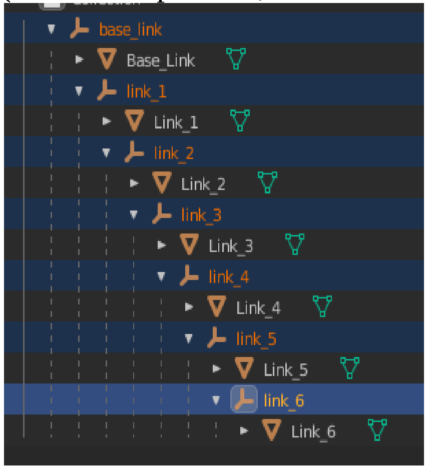


5- An example of objects included in a parent object.

c) The hierarchy should be starting from bottom to up.
for example, the base link should be a parent for link_1, and link_1 should be the parent of link_2, and so on, so we need to add multiple empty objects, which will contain each object meshes.
so, after doing the previous step with every link plain axes and its objects.
we will repeat the same step but by making the plain axes of the first link a parent for the second link, and so on up to the last link. (In the next pictures, it shows how it should look like after finishing parenting)



```
base_link
  Base_Link
  Interface_Box
  link_1
    Link_1
    Link_1_Motor1
    Link_1_Motor2
    link_2
      Link_2
      link_3
        Link_3
        Link_3_Motor1
        Link_3_Motor2
        Link_3_Motor3
        Link_3_Motor4
        link_4
          Link_4
          link_5
            Link_5
            link_6
              Link_6
```

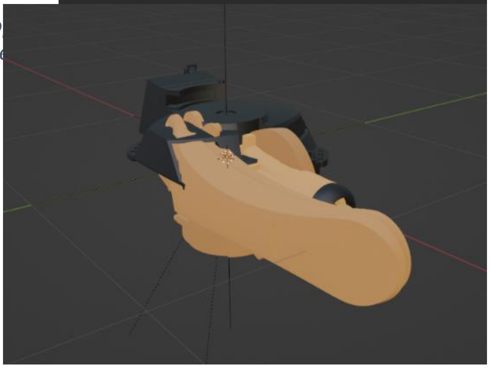


```
base_link
  Base_Link
  link_1
    Link_1
    link_2
      Link_2
      link_3
        Link_3
        link_4
          Link_4
          link_5
            Link_5
            link_6
              Link_6
```

7- An example of the required hierarchy o the objects and Plain axes in case of single meshe per link.

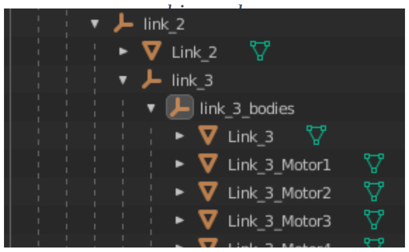
important notes:

- it might **not** look organized, but it's not a problem as the JSON contains the required translations to make it look correct on VEB.js. Because the VEB.js is not taking the positions and rotations from the GLB file as long as there are positions and rotations available in the JSON file.



8- The robot links after making the correct

-if needed, we can make another layer of parents to hold the bodies of the link, in case of there are many meshes making the link. As shown in the next figure for link 3 as it has 4 meshes representing link3, making sure that we will select the main parent of the link while running the python code.



```
link_2
  Link_2
  link_3
    link_3_bodies
      Link_3
      Link_3_Motor1
      Link_3_Motor2
      Link_3_Motor3
```

9- Adding another parent inside link 3 for organizing purposes

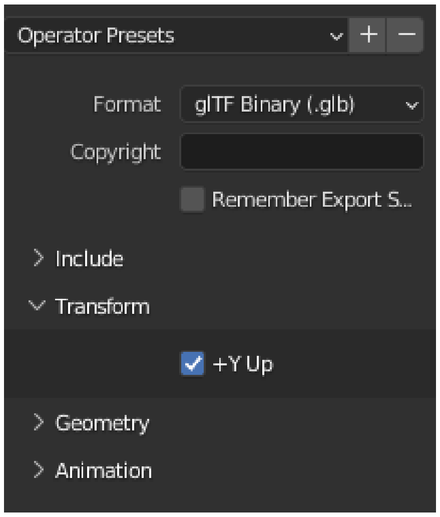
4. Save & Export the GLB and Blend files: -

We must have these two files for the next steps, the GLB is essential for the visualization on VEB.js , and the .blend file is important for the JSON generation.

We need to have both of them exactly with the same name as it's one of the main requirements.

We will export it with the standard options of the GLB export.

We use +Y Up in our case.



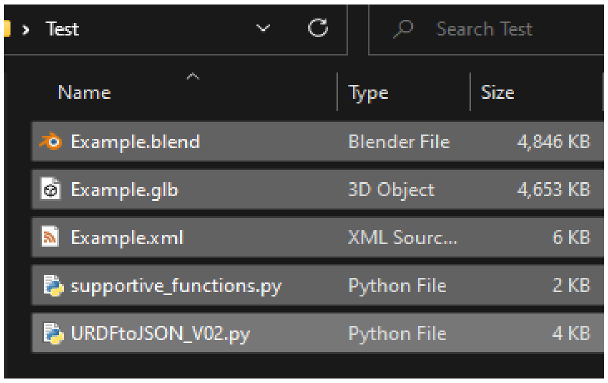
Format: glTF Binary (.glb)
Copyright:
☐ Remember Export S...
☑ +Y Up

10- Options of GLB export

III. JSON file Preparation

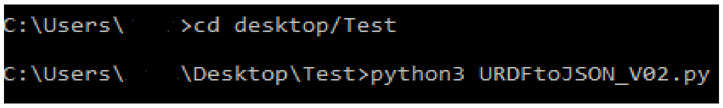
From the steps **I** & **II** we should have now an XML file, a GLB file and a Blend file.

All the files should be added to the same folder along with the 2 python code files
URDFtoJSON_V02.py
supportive_functions.py



Name	Type	Size
Example.blend	Blender File	4,846 KB
Example.glb	3D Object	4,653 KB
Example.xml	XML Sourc...	6 KB
supportive_functions.py	Python File	2 KB
URDFtoJSON_V02.py	Python File	4 KB

1. We should open the command prompt and run the python code.

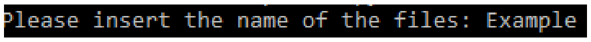


```
C:\Users\    >cd desktop/Test

C:\Users\    \Desktop\Test>python3 URDFtoJSON_V02.py
```

12- 1st command to open the folder, the 2nd to run the code.

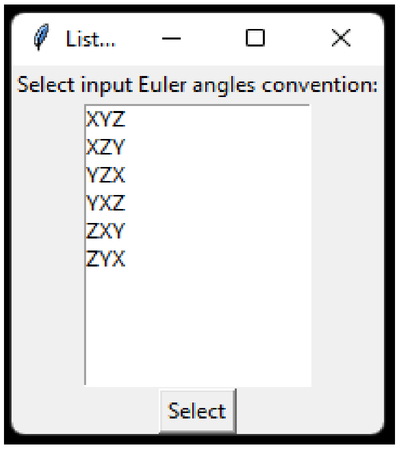
2. After running the code, you will be required to write the files names, as discussed before, all the files should be with the same name, if they are not exactly the same (case sensitive) the generator will not work properly.
so we will write Example without any extensions.



```
Please insert the name of the files: Example
```

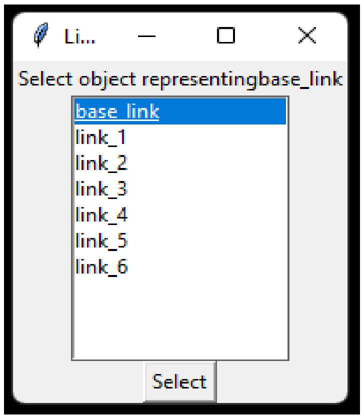
13- the code requiring the files name as an input.

3. A pop-up window will show up requiring you to select the euler convention of the input file.
   It can be found in the documentation of the file, it's very important step, as VEB.js is requiring the angles to be with YXZ convention in the JSON file.
   In the previous trials we have found out that ZYX is working well most of the times.
   but some other times it might be XYZ.

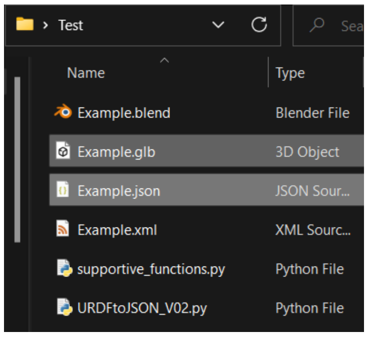


14- Euler convention selection of the input file

4. After selecting the Euler convention, a similar pop up window will show up requiring you to select the object name that resembles certain links from the URDF.
   it will show up *N times, N = number of links*, one time for each link, and each time you should select the object name representing the link written in the message of the pop-up window.



15- Selection of objects names representing certain links.



16- A screenshot showing the JSON file added to the folder, marking the two important files for the next steps.

5. After finalizing this process of the objects names selection.The process of generating the JSON file is done. You will find a JSON file with the same name added to the same file directory where you added the previous files in the first step of the JSON file preparation.

IV. Visualizing the Robot Using VEB.js

1- Uploading the JSON & the GLB to a GitHub repository.

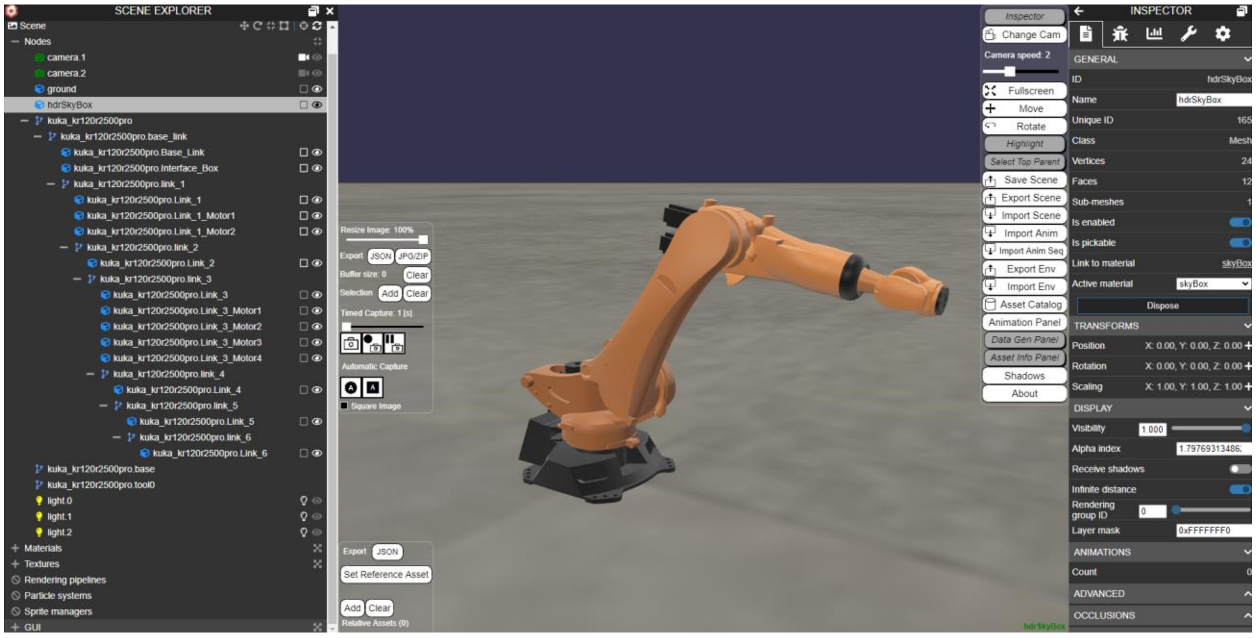
2- using this link to test the file replacing the indicated space with the raw link of the JSON file and the raw link of the path where the GLB file is located.

http://ec2-54-174-51-194.compute-1.amazonaws.com/vebjs/?inputscene=INSERT_THE_RAW_LINK_OF_THE_JSON_HERE&&?inputenv=https://wterkaj.github.io/RepoExample/example_robot/Robot_env.json&repoMod3d=
INSERT_THE_RAW_LINK_OF_THE_GLB_LOCATION_HERE

in our case this will be the link

http://ec2-54-174-51-194.compute-1.amazonaws.com/vebjs/?inputscene=https://raw.githubusercontent.com/AsemShabayek/URDFtoJSON/main/Example.json&&?inputenv=https://wterkaj.github.io/RepoExample/example_robot/Robot_env.json&repoMod3d=https://raw.githubusercontent.com/AsemShabayek/URDFtoJSON/main/

3- we should see the robot visualized



ID hdrSkyBox
Name hdrSkyBox
Unique ID 165
Class Mesh
Vertices 24
Faces 12
Sub-meshes 1
Is enabled
Is pickable
Link to material skyBox
Active material skyBox
Position X: 0.00, Y: 0.00, Z: 0.00
Rotation X: 0.00, Y: 0.00, Z: 0.00
Scaling X: 1.00, Y: 1.00, Z: 1.00
Visibility 1.000
Alpha index 1.79769313486
Receive shadows
Infinite distance
Rendering group ID 0
Layer mask 0xFFFFFFF0
Count 0

17- A screenshot from the VEB.js showing that the Example files are working (KUKA_kr120r2500pro), and tested the rotations angle and they are rotating around the joints in a correct manner.

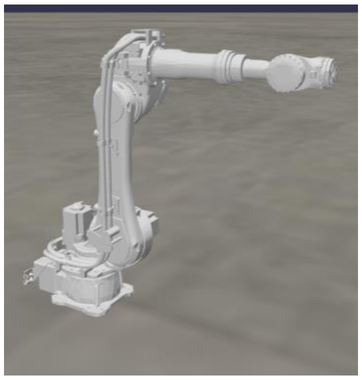
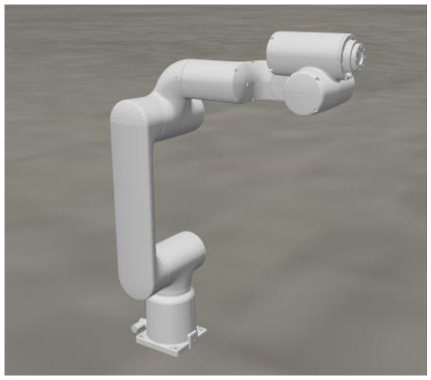
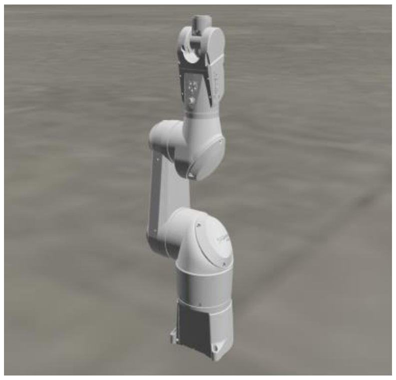
Testing the Converter on more robots

Applying the same steps on the required Robot (Comau_NJ220)

18- Comau_NJ220 on VEB.js

VEB.JS link.

Applying the same steps on other robots to demonstrate the robustness of the URDFtoJSON converter.

MOTOMAN_gp110	ABB_crb15000	STAUBLI_tx90
 ***19***- *MOTOMAN_gp110*	 ***20***- *ABB_crb15000*	 ***21***- *STAUBLI_tx90*
VEB.js Link	**VEB.js Link**	**VEB.js Link**

By making these tests and making sure that the robots are moving correctly on VEB.js, we made sure of the feasibility and robustness of the converter, that it's working on different robots from different robot producers, which are most probably using different methods for generating the URDFs.

Extra Steps

In case of no URDF available

In case of no URDF file available for a required robot, we need at least a .STEP file or any similar file for the robot, mostly it will be available at the manufacturer website, then we can generate the URDF and the meshes used in the previous steps using SW2URDF

SW2URDF

The SolidWorks to URDF exporter is a SolidWorks add-in that allows for the convenient export of SW Parts and Assemblies into a URDF file. The exporter will create a ROS-like package that contains a directory for meshes, textures and robots (URDF files). For single SolidWorks parts, the part exporter will pull the material properties and create a single link in the URDF. For assemblies, the exporter will build the links and create a tree based on the SW assembly hierarchy. The exporter can automatically determine the proper joint type, joint transforms, and axes.

It should be installed first then it can be started on SolidWorks from add-ins menu.

It works by defining the objects representing each link and exporting them in one part, so the total number of the exported parts equals the number of links.

It also adds joints, it can be done by two methods: -

1- by manually inserting axes in the desired places, it will not be so efficient and can do some errors.

2- by firstly making all the mates correctly then work on the add-in which will be able to define the joints' locations and type, as long as there is only one degree of freedom for each link.

There is a good tutorial for how to use it on this link

Important notes about SW2URDF.

After trying the add-in, there are two things needs to be taken care of to have a smoother and easier process.

1- to assemble the parts correctly and having only one degree of freedom at maximum for each part, except the first part, it should better be fixed.

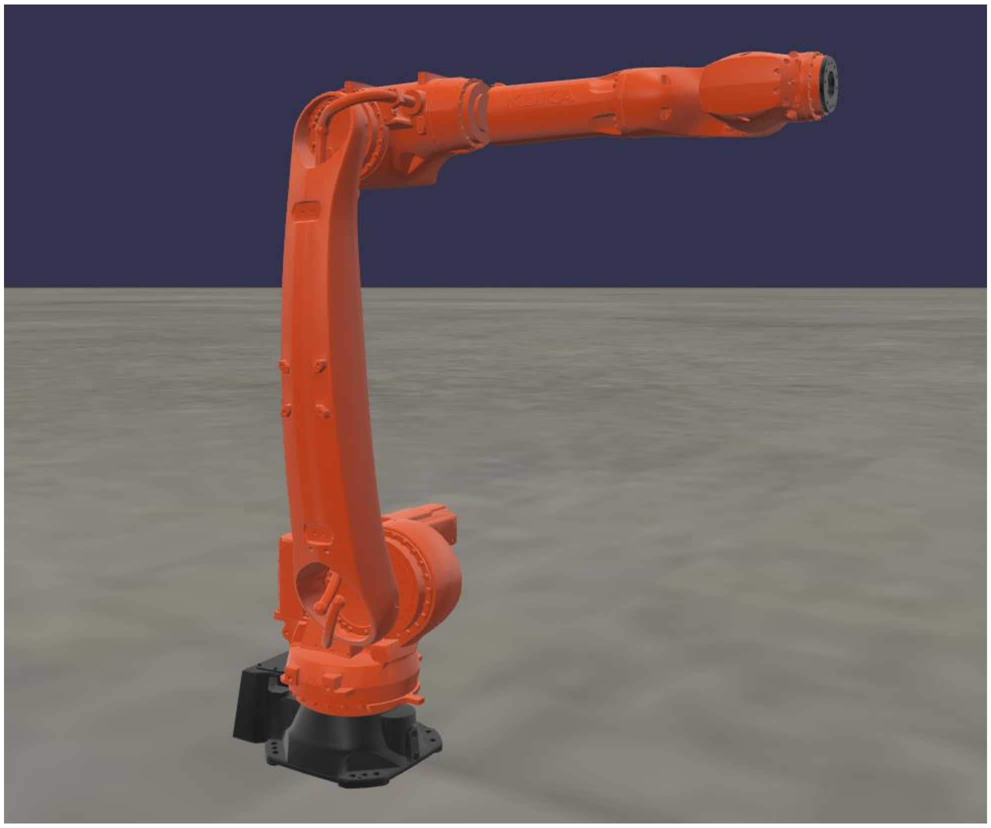
2- at the last step before exporting, we can change the roll, pitch, yaw for the joints, it will be much easier to make their original values as Zeros, as it will reduce the confusion about which Euler Angle convention should be selected.

3-to make the assembly standing on the top plan to be in a normal orientation if we are looking at it from an isometric view as shown in the image.

22- a screenshot from SolidWorks of a robot assembly from isometric view Kuka KR_50_R2500

KUKA KR_50_R2500

After trying this , we were able to generate the JSON file of the 2nd required Robot which is **KUKA KR_50_R2500,** we have downloaded the file from KUKA website and converted it to a URDF file, then it was converted with the same previous method to a JSON file.



23- KUKA KR_50_R2500 on VEB.js

VEB.js link

The process of generating the JSON from the URDF generated from SolidWorks is the same as any URDF, which proves more robustness of the python code developed.

For better looking Mesh

To have a better-looking mesh and more size efficient, few steps can make big difference.

Shade Smooth

By pressing right click while an object is selected, then selecting shade smooth it makes the part looking much better.

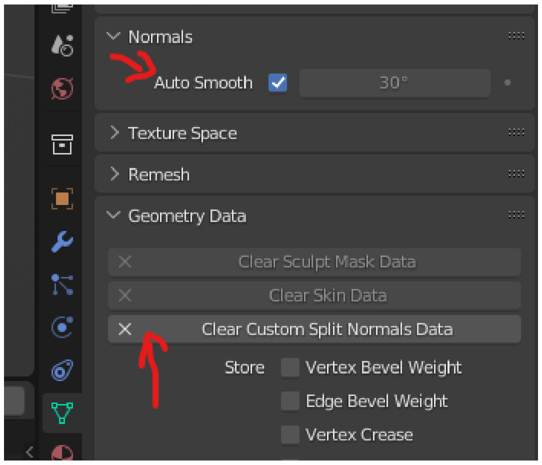
Some more steps are needed to have the maximum effect of this,

1- is to open the object in edit mode, press m and choose merge by distance, this will remove a lot of vertices, lowering file size and allowing the faces to look smooth as without doing this it's consisted of many faces not connected to each other.

2- choose auto smooth and keep it on 30 degrees to not make things smoother than it should.

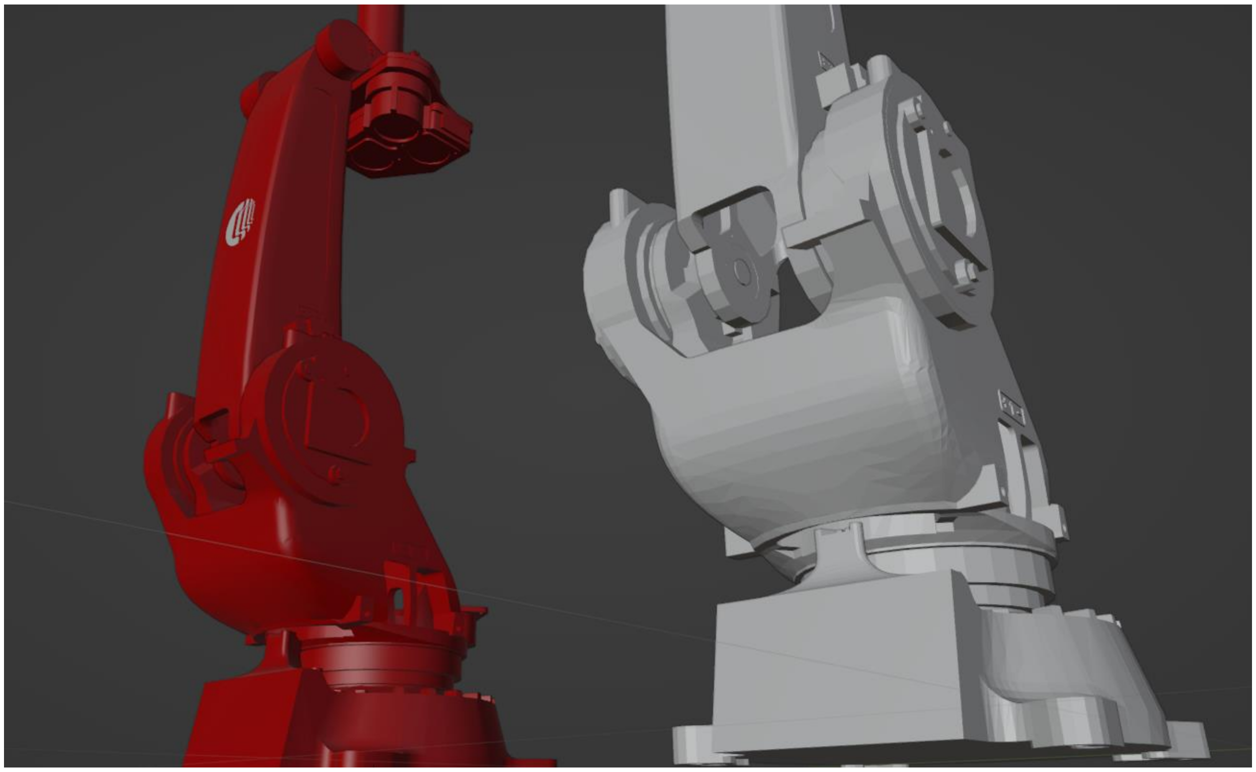
3- also choose clear custom split normal data, it will make things looking better.

With these steps in addition to some texturing to the part, the parts will look much better than the plain edgy white parts.



Normals
Auto Smooth ✓ 30°
Texture Space
Remesh
Geometry Data
Clear Sculpt Mask Data
Clear Skin Data
Clear Custom Split Normals Data
Store Vertex Bevel Weight
Edge Bevel Weight
Vertex Crease

24- options to make shade smooth look better.



25-difference showing the effect of enhanced mesh and texturing.

Future possible Developments

- A full GUI interface to generate the files without the need to run the python code from the command prompt.
- The ability to extract the colors and textures, either from the blend file or the STEP file.

Conclusion

In conclusion, after testing six different robots from various manufacturers, it can be seen that the urdf to json converter python code is reliable and capable of performing the necessary tasks. This code has the potential to be used in a variety of applications, such as visualizing and helping in controlling digital twin models of robots. Additionally, the code could be further enhanced by incorporating new features or functionality, such as the ability to convert other file types or integrate with other software. Overall, this python code serves as a valuable tool for anyone working with robotics and digital twin technology.

Appendix

Previous version of the code

This version didn’t require the file of supportive_functions.py and it was less automated than the current one.

The old one is URDFtoJSON.py

The current one is URDFtoJSON_V02.py

The added functions is supportive_functions.py

Github repository link

Table of figures

References

All references were added in hyperlinks to their tutorials or their documentations.