Stare: A Toolkit for Bringing Simulation Models and Data to Augmented Reality

Andreas Barba

Aug 29, 2025

1 Introduction

Stare is toolkit to use scientific data and models in augmented reality (AR). It has two parts: a Python package and an AR application. The Python package can be used to convert simulation models and data to a 3D model format for use in general 3D graphics applications. In the conversion process, the models can be modified and customised, much like a plotting library. This includes overlaying data, smoothening geometry, or applying materials to surfaces.

After conversion, the models can be imported into Stare's associated AR application to display it. A QR code encoding a description of a model can be scanned in the application to load the model, after which an image tag present in the real world is used to place the model in the correct position in the real world.

Stare is mainly intended to be used in connection with particles physics simulations of beam lines using BDSIM. In particular, it may allow for users of the experimental areas of CERN to 'see' the beam lines in person that they otherwise wouldn't be able to see due to shielding. Futher, Stare can be used in an educational setting, allowing for people not at a particle accelerator site to visualise beam lines.

1.1 Features

- Convert simulation models from GDML to glTF.
- Overlay data from ROOT files such as histograms onto models.
- Extract the beam line shape from a ROOT file to, for example, place histograms along a beam line.
- Modify meshes including UV unwrapping, vertex normal generation, and smoothening.
- Import from standard 3D model formats such as obj, FBX, or STEP.
- Import a 3D model in augmented reality using a QR code.
- Display the processed 3D model in AR at a specified position in the real world relative to an image tracker.

2 Getting Started

2.1 Installation

The simplest way to install Stare is via pip.

```
pip install stare-ar
```

Alternatively, the repository can be cloned locally and installed.

```
git clone ssh://git@gitlab.cern.ch:7999/en-ea-le-groups/software/stare.git
cd stare
pip install .
```

ROOT also needs to be installed if loading ROOT files is necessary.

2.2 Loading a Model

A model in GDML format can be loaded using the following.

```
import stare.io.pyg4ometry
model = stare.io.pyg4ometry.load_gdml("my_model.gdml")
```

This will return the root of the scene graph (a `SceneNode` object). Loading a 3D model (in glTF, obj, FBX, STEP, etc.) can also be done.

```
import stare
model = stare.io.model.load("my_model.gltf")
```

2.3 Loading and Visualising Data

We will give a small example of getting a 2D texture from a 3D histogram and placing it on a rectangular mesh to illustrate the basic process of overlaying data. A more detailed description can be found under *Overlaying Data*.

Data is usually loaded from a file. For ROOT files, 1D, 2D, or 3D histograms can be extracted from it.

```
histogram = stare.io.root.load_histogram3d("my_data.root", "Event/MergedHistograms/my_
↪histogram")
```

After loading, the data should be converted such that it can be placed in a 3D model. For example, we can take a slice i of a 3D histogram to get a 2D histogram and create a texture from it.

```
texture = histogram.get_slice(i).to_texture()
```

The texture can be placed on a rectangular mesh (a quad) and added to the scene graph at a position of (5, 1, 0) relative to the root node.

```
obj = place_texture_on_quad(texture)
instance = stare.ObjectInstance(obj, stare.transform.translation_matrix(5.0, 1.0, 0.0))
model.instances.append(instance)
```

2.4 Making it Pretty

To make a model more visually appealing, materials and geometry processing can be applied to the meshes.

To apply custom materials on the objects in the model, a material generator can be used when loading a GDML model. A material generator takes a pyg4ometry material and converts it to a Stare material. If no material is generated, the material embedded in the GDML file is used.

```
def my_material_generator(material):
    if "concr" in material.name.lower():
        return stare.Material("concrete_colour.jpg", "concrete_roughness.jpg", 0.0,
↪"concrete_normals.jpg")
    return None

model = stare.io.pyg4ometry.load_gdml("k12_2021.gdml", prioritise_embedded_
↪materials=False, material_generator=my_material_generator)
```

Alternatively, materials can be modified after loading.

```
for m in model.all_materials():
   if "concr" in m.name.lower():
       m.base_colour = "concrete_colour.jpg"
       m.roughness = "concrete_roughness.jpg"
       m.metalness = 0.0
       m.normal_map = "concrete_normals.jpg"
```

Additionally, meshes can be tweaked to get the desired look. Here, we smoothen and then simplify/decimate the mesh as well as generate vertex normals for it such that edges and corners are pronounced.

```
for m in model.all_meshes():
    smoothened = m.smoothen(flattening_iterations=2, smoothening_iterations=1)
    smoothened = smoothened.decimate(removed_fraction=0.6, quality=0.4)
    m.positions = smoothened.positions
    m.indices = smoothened.indices

    m.align_winding_order()
    if m.face_normals_are_inwards():
        m.flip_winding_order()

    m.generate_flat_shading_normals()
```

2.5 Exporting

When customisation of the model is done, the model can be exported to glTF (or GLB, the binary version of glTF recommended for performance). It is recommended to UV unwrap meshes that need unwrapping and to make materials glTF compatible before exporting.

```
model.uv_unwrap_meshes(stare.mesh.UvUnwrapMethod.XATLAS)
model.make_gltf_compatible()
stare.io.gltf.to_gltf(model, "my_scene.gltf") # Change to .glb for a GLB file
```

2.6 Viewing the Model in Augmented Reality

The AR application can load models using a QR code. When scanning a QR code, the application expects that it contains a JSON string either embedded in the QR code or as a URI to a JSON file. The JSON file contains information on which models need to be loaded and how image tags, used to detemine where a model should be placed in the real world, are are placed in relation to the model's origin. Currently, AprilTags are used, and the only supported tag set is `tagStandard41h12`. It is important that the physical size of the tag matches what is specified in the application's settings. Tag images with IDs from 0 and up can be found here, or they can be generated and printed with the correct size here.

```
{
  "Name": "Hello World Model",
  "Models": [
    {
      "Name": "Example Model",
      "Uri": "https://raw.githubusercontent.com/path/to/my_model.gltf",
      "Translation": {
        "x": 0.0,
        "y": 1.0,
        "z": 5.0
      },
      "Rotation": {
        "x": 90.0,
        "y": 0.0,
        "z": 0.0
      },
      "Scale": {
        "x": 0.05,
        "y": 0.05,
        "z": 0.05
      }
    }
  ],
  "TagPlacements": [
    {
      "TagID": 0,
      "Position": {
        "x": -5.0,
        "y": 0.0,
        "z": -10.0
      },
      "Rotation": {
        "x": 0.0,
        "y": 90.0,
        "z": 0.0
      }
    }
  ]
}
```


3 Structure and Design

3.1 Scene

A scene is an object consisting of instances of meshes with materials applied. A scene is represented as a graph, a tree of nodes (`SceneNode`), where each node has a transformation (expressed by a matrix) and a number of object instances (`ObjectInstance`). Each instance consists of a transform (a matrix) and a scene object (`SceneObject`), which consists of a mesh (`Mesh`) and a material (`Material`). Materials are based on the roughness-metallic PBR model. Note that there is no scene class, but rather a scene is the root of the scene graph.

3.2 Data

The `stare.data` sub-module handles data that usually come from files (e.g. ROOT files). Using this sub-module histograms can be created, modified, and converted to a texture or 3D model, or the beam line shape can be read so that data can be overlayed along a beam line model.

3.3 Input and Output (IO)

The `stare.io` sub-module is used to read and write files to and from persistent storage. This is used for loading models, both simulation models (e.g. in GDML) and 3D models (glTF, obj, FBX, STEP, etc.), as well as for exporting to a 3D model format (currently only glTF). Furthermore, histograms and the beam line shape can be extracted from ROOT files.

4 Overlaying Data

In order to overlay data on models, the data most likely needs to be extracted from a file. The type of data dealt with is usually histograms. They can be converted to various visual forms such as textures, meshes, or 3D models. Furthermore, data sometimes needs to follow a beam line. The beam line shape can be extracted from a ROOT file and can then be used for placing data along the beam line.

4.1 Histograms

Histograms can be loaded from a ROOT file or created from a Numpy array. Histograms are loaded via `load_histogram1d`, `load_histogram2d`, or `load_histogram3d`.

```
import stare.io.root
histogram2D = stare.io.root.load_histogram2d("my_root_file.root", "path/to/histogram/in/
↪root/file")
```

Additionally, a histogram can be created by slicing a histogram of higher dimensionality.

```
histogram1D = histogram2D.get_slice(i, axis=1)
```

All histograms can be voxelised, i.e. turned into a grid of meshes with varying materials (usually heatmap-like).

```
voxel_grid = histogram.voxelise((width, height, depth), decimation=2.0)
```

4.2 1D Histogram

A 1D histogram can be turned into a 2D texture (an image) that depicts a bar plot.

```
texture = histogram.to_texture(height=50, bar_width=1, bar_colour=stare.Colour(1, 0.2, 0.
↪3, 1), background_colour=stare.Colour(0, 0, 0, 0))
```

Similarly, it can be turned into a mesh either depicting a bar plot or a smoother plot where data points are connected by straight lines. The meshes can also be set to follow a beam line.

```
bar_mesh = histogram.to_bar_mesh(width=15, height=15)
smooth_mesh = histogram.to_smooth_mesh(width=20, height=10)
bar_mesh_along_beam_line = histogram.to_bar_mesh_following_line(beam_line=bl, height=15)
smooth_mesh_along_beam_line = histogram.to_smooth_mesh_following_line(beam_line=bl,␣
↪height=10)
```

4.3 2D Histogram

Similarly to 1D histogram, a 2D histogram can be turned into a 2D texture. The value of each bin in the histogram maps to a pixel value on the texture. A value-to-colour function can optionally be given.

```
texture = histogram.to_texture(value_to_colour)
```

4.4 3D Histogram

Like a 1D and 2D histogram, a 3D histogram can be voxelised, i.e. turned into a grid of meshes. Additionally, it can be made to follow a beam line.

```
voxel_grid = histogram.voxelise_along_beam_line(beam_line, (width, height), decimation=2.
↪0, axis=2)
```

4.5 Beam Line Shape

In order to work with the shape of a beam line, it must be extracted from a ROOT file. Using this, data can be overlayed such that it follows the beam line.

```
bdsim_model = stare.io.root.load_bdsim_model("from_bdsim.root")
beam_line = bdsim_model.beam_line
```

A useful function to use in order to overlay data is the beam line's `transform_at_distance` function. Given a distance along the beam line, it returns the transformation matrix at the point at the given distance.

```
instance.transform = beam_line.transform_at_distance(0.5 * beam_line.total_length())
```

4.6 Using a Matplotlib Figure

Matplotlib figures can be converted to textures.

```
texture = stare.texture.from_matplotlib_figure(fig)
```

Or alternatively from a canvas using `stare.texture.from_matplotlib_canvas`.

4.7 Placing the Data in the Scene Graph

After the data is converted to the desired visual form, it can be placed in the scene graph. For example, if data is converted to a mesh, it can be overlayed on the model by placing it in the scene graph.

```
data_object = SceneObject(data_mesh, my_material, name="My Data")
instance = ObjectInstance(data_object, transformation_matrix)
scene.instances.append(instance)
```

Or, if data is converted to a texture, it can be used in a material applied on a rectangular mesh which can be instanced and placed in the scene.

```
textured_quad = place_texture_on_quad(texture)
instance = ObjectInstance(textured_quad, transformation_matrix)
scene.instances.append(instance)
```

And sometimes, another scene graph with the data needs to be overlayed.

```
scene.children.append(another_scene_with_data)
```

5 Modifying Geometry

Minimally, a mesh is an array of vertex positions and triangles, where each triangle corresponds to three indices into the array of vertex positions.

> **Note**
>
> Triangle meshes (i.e. meshes with made up of solely triangles) are the usual types of meshes that Stare deals with. It is in some cases possible to use quad meshes or meshes with a mixture of arbitrary polygons.

Meshes can optionally have vertex normals and texture coordinates (UVs). In this case, similarly to vertex positions, each of the three indices in a triangle is an index into the vertex normal and/or the texture coordinate array.

5.1 Generating Texture Coordinates

If a mesh has a material with a texture applied to it, it must have texture coordinates (UVs) that tell how the image is to be mapped on the mesh's surface. If UVs are not included in the model, they can be generated.

```
my_mesh.uv_unwrap(stare.mesh.UvUnwrapMethod.XATLAS)
```

Alternatively, all meshes that need unwrapping in a scene can be unwrapped in one line.

```
my_scene.uv_unwrap_meshes(stare.mesh.UvUnwrapMethod.BOX_MAPPING)
```

5.2 Generating Vertex Normals

Vertex normals influence for how lighting is calculated for a mesh. They are used to calculate surface normals used for lighting calculations that override the actual normals of the triangles in the mesh. In many cases, it is not strictly necessary to include vertex normals in an exported mesh, as programs that use meshes for graphics will usually generate ones on import based on the actual normals of the triangles in the mesh. Nevertheless, it can be desirable to have control over what normals are used. If vertex normals are not already included in the mesh, they can be generated with Stare.

```
my_mesh.generate_flat_shading_normals()
my_mesh.generate_smooth_shading_normals()
```

Here, the example of normals for flat and smooth shading are given, where flat shading makes the mesh's edges and corners pronounced and smooth shading make them look completely smooth.

> **Note**
>
> When generating vertex normals, the winding order of all triangles in a mesh should be consistent. See *Triangle Winding Order* for more details.

5.3 Subdividing and Smoothening a Mesh

To introduce more detail in a mesh, its faces may be subdivided, i.e. its faces split into smaller faces.

```
new_positions, new_indices = stare.mesh.subdivide(my_mesh.positions, [my_mesh.indices],␣
↪stare.mesh.SubdivisionMethod.CATMULL_CLARK)
```

The returned polygons are always quads and can be turned into triangles using `stare.mesh.triangulate_quads`. Note that currently, vertex normals and UVs are lost during the subdivision process.

The `CATMULL_CLARK` subdivision method will make the mesh smoother while the `SIMPLE` method will preserve the shape of the mesh and will simply create additional vertices.

Before subdividing the mesh, it is recommended to turn the mesh into a mesh of quads. This usually makes the subdivision look more natural.

```
quads, remaining_triangles = stare.mesh.quadify(my_mesh.positions, my_mesh.indices)
```

Currently, it is not always possible to completely quadify a given mesh. Quadification works well for meshes that used to consist solely of quads and have been triangulated. Note that currently, no new vertices are generated using this function, meaning some meshes cannot be turned into quad meshes.

Using the above, it is possible to implement a smoothening function. However, the `Mesh` class already comes with this functionality.

```
smoothened_mesh = m.smoothen(flattening_iterations=2, smoothening_iterations=1)
```

Here, we have 2 iterations of simple subdivision followed by 1 iteration of Catmull-Clark subdivision to achieve a mesh where the corners and edges are smoother.

5.4 Triangle Winding Order

Many real time rendering programs use backface culling, i.e. not rendering triangles of a mesh if the face visible to the camera is not considered the front face. For closed meshes, this means less computation time is spent, as backfacing triangles are covered by front facing triangles and won't be visible anyway. However, this implies that triangles have a direction that they are facing. This is detemined by the triangle's winding order, the order in which the three indices to vertices of the triangle appear in memory. There are two winding orders, clockwise (CW) and counterclockwise (CCW). When rendering from a given viewpoint and assuming a CCW winding order, triangles whose vertices are arranged in a CCW order with respect to the camera view are considered to be frontfacing and equivalently for a renderer using a CW winding order.

There is often no guarantee that meshes have consistent winding order. Using Stare, the winding order of a mesh's triangles can be aligned such that all triangles agree on what is considered the mesh's inside and outside.

```
my_mesh.align_winding_order()
```

This makes no guarantees for what winding order is used. To get a desired winding order, the winding order can be flipped for misbehaving meshes.

```
if m.face_normals_are_inwards(): # with respect to a CCW winding order
    m.flip_winding_order()
```

6 Displaying Models in Augmented Reality

When you have a model that you would like to display in augmented reality at a fixed real world location, you can use the AR app associated with Stare. Models can be imported via a QR code and their real world placement detemined via image-based tags.

6.1 Importing a Model

Models can be imported into the AR app by scanning a QR code.

The QR code either encodes the contents of a JSON file or a URI to a JSON file. The JSON file includes information on the model to be loaded, including URIs to the necessary 3D model files, its placement and scaling, and the relative placement of the image-based tags (AprilTags) used for tracking the real world position. An example JSON file for describing a model is shown below.

```
{
  "Name": "Hello World Model",
  "Models": [
    {
      "Name": "Example Model",
      "Uri": "https://raw.githubusercontent.com/path/to/my_model.gltf",
      "Translation": {
        "x": 0.0,
        "y": 1.0,
        "z": 5.0
      },
      "Rotation": {
        "x": 90.0,
        "y": 0.0,
        "z": 0.0
      },
      "Scale": {
        "x": 0.05,
```

(continues on next page)

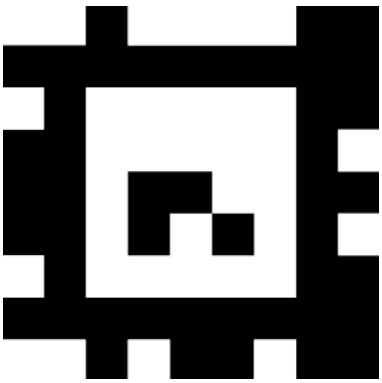
(continued from previous page)

```
        "y": 0.05,
        "z": 0.05
      }
    }
  ],
  "TagPlacements": [
    {
      "TagID": 0,
      "Position": {
        "x": -5.0,
        "y": 0.0,
        "z": -10.0
      },
      "Rotation": {
        "x": 0.0,
        "y": 90.0,
        "z": 0.0
      }
    }
  ]
}
```

Here, we specify the name of the model, the model files it consists of, and the placement of the tags relative to the model's origin. Note that there can be more than one model and more than one tag. The tag ID indicates which AprilTag image is used. Currently, the only supported tag set is `tagStandard41h12`. A list of tag images can be found here, and a generator that allows for printing a specific size can be found here.

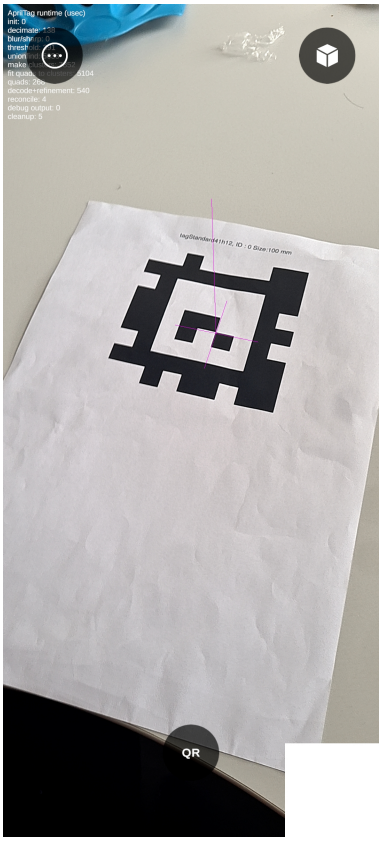
6.2 Real World Tracking and Placement

In order to know where to place a model, the app makes use of image-based tags known as AprilTags. An example of the AprilTag with ID 0 from the tag set `tagStandard41h12` is shown below.



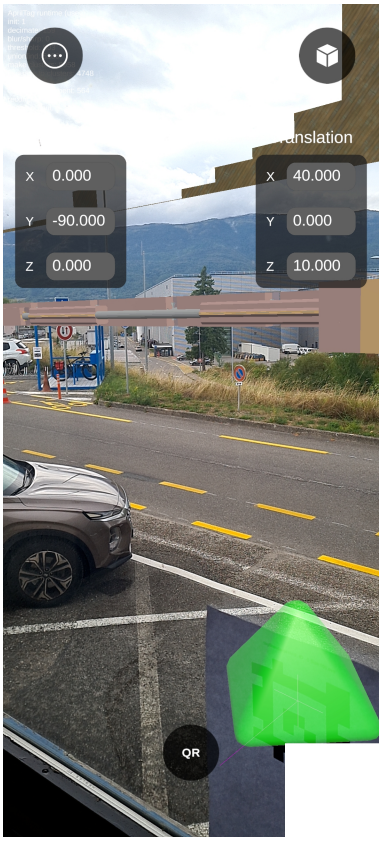
When using the app, when such a tag is in the view of the camera, the tag will be tracked.

> **Note**
>
> It is important that the physical size of the tag matches the size specified in the application's settings. Otherwise, the depth information will be incorrect and the model will appear closer or further away than expected. Note that the size of a tag is the distance between the detection corners, not the outside edges. For example, a tag of size 5.56 cm (the value to be used in the app) will have a total size of 10 cm.



6.3 Adjusting the Placement of the Model

Inside the AR app, the placement of the models can be adjusted manually and in a way that is immediately responsive, making it easier to fine-tune the placement of a model, as opposed to having to change the associated JSON string and reimporting. When the model has been adjusted, the new JSON string associated with the model can be copied to the clipboard so that the adjusted placement can be encoded in a QR code.



7 Examples

This section includes some examples to show how model conversion may be done.

- *K12 Beam Line*
- *P42 Beam Line*

7.1 K12 Beam Line

This is an example showing the conversion of the K12 beam line model. Here, the shielding blocks get a concrete material applied and all meshes are smoothed and shaded flat.

```
import stare
import stare.io.pyg4ometry


# Load model
def my_material_generator(material):
    if "concr" in material.name.lower():
        return stare.Material("concrete_colour.jpg", "concrete_roughness.jpg", 0.0,
↪"concrete_normals.jpg")
    return None


model = stare.io.pyg4ometry.load_gdml(
    "k12_2021.gdml", prioritise_embedded_materials=False, material_generator=my_material_
```

(continues on next page)

(continued from previous page)

```
↪generator
)
scene = stare.SceneNode(children=[model])

# Modify scene
model.filter_instances(lambda x: "ecn" not in x.name)

# Modify geometry
for m in model.all_meshes():
    subdivided = m.smoothen(flattening_iterations=2, smoothening_iterations=1, quadify_
↪first=True)
    subdivided = subdivided.decimate(removed_fraction=0.4, quality=0.6)
    m.positions = subdivided.positions
    m.indices = subdivided.indices

    m.align_winding_order()
    if m.face_normals_are_inwards():
        m.flip_winding_order()

    m.generate_flat_shading_normals()

# Export to glTF
scene.uv_unwrap_meshes(stare.mesh.UvUnwrapMethod.XATLAS)
scene.make_gltf_compatible()
stare.io.gltf.to_gltf(scene, "k12.gltf", embed_all_textures=False, embed_binary=False)

scene.print_summary()
```

The JSON file needed for displaying the model in the AR application is shown below.

```
{
  "Name": "K12 Beam Line",
  "Models": [
    {
      "Name": "Beam Line",
      "Uri": "https://raw.githubusercontent.com/anveba/aredu/refs/heads/master/Assets/
↪Stare/Web/k12.gltf"
    }
  ],
  "TagPlacements": [
    {
      "TagID": 0,
      "Position": {
        "x": -50.0,
        "y": 0.0,
        "z": -10.0
      },
      "Rotation": {
        "x": 0.0,
        "y": 90.0,
        "z": 0.0
      }
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

The QR code containing the URL to the JSON file is shown below.

7.2 P42 Beam Line

In this is example, the P42 beam line is converted. Here, the beam loss data is overlayed along the beam line as a histogram above it.

```
import math
import stare
import stare.io.pyg4ometry
import stare.io.root

# Load GDML file
model = stare.io.pyg4ometry.load_gdml("p42_2024.gdml")
scene = stare.SceneNode(children=[model])

model.filter_instances(lambda x: "XXR" not in x.name and "shielding" not in x.name.
↪lower())

# Load Eloss histogram from ROOT file and place along beam line
bl = stare.io.root.load_bdsim_model("final_combined.root").beam_line
histogram = stare.io.root.load_histogram1d("final_combined.root", "Event/
↪MergedHistograms/ElossHisto")
histogram_mesh = histogram.to_bar_mesh_following_line(bl, 15)
histogram_mesh.rotate_uvs(math.pi / 4, (0.5, 0.0))
histogram_mesh.flip_uvs()
histogram_mesh.uvs *= 500
hist_material = stare.Material(base_colour="bamboo_colour.jpg", roughness="bamboo_
↪roughness.jpg", metalness=0.0)
hist_obj = stare.SceneObject(histogram_mesh, hist_material, name="ElossHisto")
instance = stare.ObjectInstance(hist_obj, transform=stare.transform.translation_matrix(0,
↪ 2.5, 0), name="ElossHisto")
scene.instances.append(instance)

# Modify meshes
for m in model.all_meshes():
```

(continues on next page)

(continued from previous page)

```
    m.align_winding_order()
    if m.face_normals_are_inwards():
        m.flip_winding_order()

    if not m.has_normals():
        m.generate_flat_shading_normals()

# Export to glTF
scene.make_gltf_compatible()
stare.io.gltf.to_gltf(scene, "p42.gltf", embed_all_textures=False, embed_binary=False)

scene.print_summary()
```

The JSON file needed for displaying the model in the AR application is shown below.

```
{
  "Name": "P42 Beam Line",
  "Models": [
    {
      "Name": "Beam Line",
      "Uri": "https://raw.githubusercontent.com/anveba/aredu/refs/heads/master/Assets/
↪Stare/Web/p42.gltf"
    }
  ],
  "TagPlacements": [
    {
      "TagID": 0,
      "Position": {
        "x": -40.0,
        "y": 0.0,
        "z": -10.0
      },
      "Rotation": {
        "x": 0.0,
        "y": 90.0,
        "z": 0.0
      }
    }
  ]
}
```

The QR code containing the URL to the JSON file is shown below.