# Utility Pole Synthetic Data Generator

A comprehensive system for generating synthetic training data of utility poles and their components using Blender. This project enables the creation of large-scale, annotated datasets for computer vision and machine learning applications in power infrastructure inspection.

![Example Synthetic Utility Pole](ExampleImage.png)

\*Example of a generated synthetic utility pole with segmentation mask and annotations\*

## Table of Contents

- [Features](#features)

- [Prerequisites](#prerequisites)

- [Installation](#installation)

- [Project Structure](#project-structure)

- [Usage](#usage)

- [Output Structure](#output-structure)

- [Code Overview](#code-overview)

- [Blender File Overview](#blender-file-overview)

## Features

- \*\*Diverse Pole Types\*\*: Support for multiple pole configurations:

- Modified Vertical Framing

- Vertical Framing

- Deadend Framing

- Crossarm Framing

- DoubleDeadend Framing

- DoubleCrossarm Framing

- Alleyarm Framing

- \*\*Component Variations\*\*:

- Multiple phase configurations (single, two, and three-phase)

- Various equipment types (AETX, ATS, ALS, ATS, Fuse, Surge Arrester, FCI)

- Support for anomaly generation (open switches, damaged components, rust, flashes)

- Different pole materials (wood, concrete)

- \*\*Advanced Rendering\*\*:

- GPU-accelerated rendering with CYCLES

- Support for OPTIX/CUDA acceleration

- Configurable resolution and quality settings

- HDR background integration

- \*\*Annotation Export\*\*:

- COCO format annotation export

- Instance segmentation support

- Automated label generation

## Prerequisites

- Blender 4.3+

- Python 3.7+

- Required Python packages (installed via requirements.txt)

- SyntheticDataProject.blend (Blender file with 3D assets)

- HDRI Image Folder

## Installation

1. Clone the repository:

```bash

git clone https://github.com/jabrams-11/Synthetic-Data.git

cd Synthetic-Data

```

2. Install required Python packages into Blender's Python environment:

First, locate your Blender Python executable and site-packages directory. For Blender 4.3 on Windows, these are typically:

```bash

BLENDER\_PYTHON="C:\Program Files\Blender Foundation\Blender 4.3\4.3\python\bin\python.exe"

BLENDER\_SITE\_PACKAGES="C:\Program Files\Blender Foundation\Blender 4.3\4.3\python\Lib\site-packages"

```

Then install all required packages using the requirements.txt file:

```bash

& "$BLENDER\_PYTHON" -m pip install -r requirements.txt --target "$BLENDER\_SITE\_PACKAGES"

```

Or install packages individually if needed:

```bash

& "$BLENDER\_PYTHON" -m pip install [package-name] --target "$BLENDER\_SITE\_PACKAGES"

```

Note: For other operating systems or Blender versions, adjust the paths accordingly.

3. Configure the output and HDRI images path in the rendering.yaml file

## Project Structure

```

project\_root/

-scripts/

generate.py # Main generation script

process\_output.py # Post-processing utilities

save\_coco.py # COCO format conversion

core/

\_\_init\_\_.py

trackers.py # Rotation/Material Anomaly Trackers

generators/ # Cycles through the variation of rust

anomalies.py

poles/ # Pole type implementations

Alleyarm.py

DoubleDeadend.py

DoubleCrossarm.py

ModifiedVertical.py

Vertical.py

Deadend.py

Crossarm.py

rendering/

renderer.py # Rendering configuration

camera.py # Camera setup

background.py # Background handling

scripts/ # Batch pole config generation

batch\_config.yaml

batch\_generate.py

cleanup\_coco.py

combine\_datasets.py

count\_coco\_tags.py

generate.py

deleteMasks.py

process\_output.py

utils/

progress\_window.py # GUI progress monitor

wire\_generator.py # Wire generation utilities

configs/

rendering.yaml # Render settings

pole\_generation\_config.yaml # Pole configuration

```

## Usage

### Command Line Execution

The system is designed to be run from the command line using Blender's Python interface. Use the following command structure:

Batch generation, used when generating multiple images for one specific pole framing that has different configs in folders.

```bash1

"[path-to-blender]/blender.exe" -b "[path-to-blend-file]" -P "[path-to-script]/batch\_generate.py"

```

Example:

```bash1

"C:\Program Files\Blender Foundation\Blender 4.3\blender.exe" -b "C:\path\to\your\scene.blend" -P “C: \Users\FPL Laptop\Desktop\BlenderUpdatedSyntheticDataCode\Synthetic-Data\scripts\batch\_generate.py”

```

Number generation is used when generating a certain amount of images that cycle through all the possible cases of these pole configs.

```bash2

"[path-to-blender]/blender.exe" -b "[path-to-blend-file]" -P "[path-to-script]/generate.py" -- --num-images <count>

```

Example:

```bash2

"C:\Program Files\Blender Foundation\Blender 4.3\blender.exe" -b "C:\path\to\your\scene.blend" -P Synthetic-Data/scripts/generate.py -- --num-images 5

```

Parameters:

- `-b`: Run Blender in background mode

- `--num-images`: Number of synthetic images to generate

- Additional parameters can be configured in the YAML config files

### Python API Usage

For programmatic control, you can also use the Python API within Blender:

```python

from scripts.generate import batch\_render

# Generate 100 synthetic images

batch\_render(num\_images=100)

```

## Output Structure

```

output\_dir/

Image\_0001.png # Rendered images

Mask\_0001.png # Segmentation masks

coco\_annotations.json # COCO format annotations

all\_frame\_mappings.json # BW Mask Index to Category ID

generation\_status.json # Progress tracking

```

## Progress Monitoring

The system includes a GUI progress monitor that displays:

- Equipment display

- Generation progress

- Time estimates

- Pole type distribution

- Current status

## Code Overview

### Core Components

#### 1. Generation Pipeline

The main generation pipeline is handled by `scripts/generate.py`, which orchestrates the entire process:

- Scene initialization and cleanup

- Pole type selection and generation

- Camera and background setup

- Rendering and output processing

#### 2. Pole Generation

The system uses a hierarchical class structure for pole generation:

- `PoleBase`: Base class with common functionality

- Specialized classes ('ModifiedVertical', 'Vertical', 'Deadend', etc.)

#### 3. Component Management

Components and their probabilities are managed through Blender collections and configured via YAML:

3.1

For batch usage run this:

```yaml:scripts/batch\_config.yaml

batches:

-name:

pole\_type:

setup:

phases:

cam\_distance:

material:

count:

focus\_point:

anomalies:

enable:

apply\_rust: or porcelain\_fuse\_flashed:

fuse\_open:

3.2

For probabilities generation when you’re not doing batch run this:

```yaml:configs/pole\_generation\_config.yaml

equipment\_chances:

bare: 10 # Chance of having bare pole (just insulators)

aetx: 25 # Chance of having AETX when eligible (2+ phases)

fuse: 50

ats: 50

als: 40

afs: 20 # only coss arm poles

three\_phase\_aetx: 20

surge\_arresters: 30 # 30% chance of having surge arresters

fcis: 40 # 10% chance of having fcis (only w/ 3-ph)

support\_bracket: 50 # 40% chance of having support bracket (vertical only)

```

#### 4. Rendering Pipeline

The rendering pipeline (`scripts/process\_output.py`) handles:

- Image rendering

- Mask generation

- COCO format annotation export

### Configuration System

The project uses two main configuration files:

1. \*\*rendering.yaml\*\*: Controls rendering settings

- Resolution and quality

- Camera parameters

- Output formats

- Performance settings

2. \*\*pole\_generation\_config.yaml\*\*: Defines pole generation parameters

- Pole type probabilities

- Component configurations and probabilities

- Material settings

- Anomaly chances

### Integration Flow

1. \*\*Initialization\*\*

- Script is launched via Blender's Python interface through Terminal or Blender Script Editor

- Configuration files are loaded

- Scene is reset and prepared

2. \*\*Generation\*\*

- Random pole type is selected based on weights

- Components are added according to configuration

- Anomalies are generated if specified

3. \*\*Rendering\*\*

- Camera is positioned

- Background is randomized

- Scene is rendered

- Masks and annotations are generated

4. \*\*Output Processing\*\*

- Images are saved

- COCO annotations are generated

- Progress is tracked and reported

### Example Workflow

1. \*\*Setup\*\*: Configure generation parameters in YAML files

2. \*\*Execute\*\*: Run the main script through Blender

3. \*\*Monitor\*\*: Watch progress through the GUI monitor

4. \*\*Collect\*\*: Gather generated images and annotations from output directory

## Blender File Overview

### Blender File Requirement

The Blender file (`SyntheticDataProject.blend`) is a required component of this system as it contains all the necessary 3D assets, configurations, and components for generating utility pole data. Any new configurations or equipment variations can be added by creating a new folder and placing the required components, textures, and models inside.

To integrate new elements into the pipeline, the associated scripts must be modified to recognize and process the new components during the data generation phase. This ensures that the rendering pipeline correctly incorporates new assets and updates annotations as needed.

### Custom Tags for Annotation and Labeling

The system supports custom tags for annotation and labeling, allowing precise control over which assets are included in the generated dataset. These tags are defined as string properties assigned to each object in Blender. When adding a new asset, ensure that the following attributes are set:

- \*\*`annotate` (Boolean, Default: False)\*\* â€“ If set to `True`, the object will be included in the annotation output.

- \*\*`label` (String)\*\* â€“ Defines the category label for the object. This label is used in COCO annotations to assign the correct class to each object.

### HDRI Images

The system utilizes HDRI images to provide realistic environmental lighting and background variations. These images are essential for achieving high-quality renders with accurate reflections, shadows, and scene realism.

#### Adding HDRI Images

1. Place HDRI images inside the designated HDRI folder (`/hdri/`).

2. Ensure that the HDRI paths are correctly referenced in the `rendering.yaml` configuration file:

```yaml

hdri\_path: "path/to/hdri/folder"

#### Adding a New Annotated Asset

To include a new asset in the dataset:

1. Import or create the asset in Blender.

2. Assign the custom properties:

```python

bpy.context.object["annotate"] = True # Enables annotation

bpy.context.object["label"] = "NewComponent" # Sets the object label

### Wire Builder Utility

The Wire Builder Utility automates the creation of realistic overhead power lines between two points using a Bezier curve. This function generates sagging wires with slight randomization to simulate real-world conditions.

#### How It Works

- Connects two empty objects (representing wire attachment points).

- Uses Bezier curves to model realistic sagging behavior.

- Supports customizable sag depth, thickness, and randomization for natural variation.

- Automatically places the wire inside the `Wires` collection.

- Annotates the wire for dataset labeling.

#### How to Use

1. Create Two Empty Objects in Blender at the locations where the wire should connect.

2. Run the function by calling `create\_power\_wire(empty1, empty2)`.

3. Modify Parameters such as `wire\_thickness` and `sag\_factor` as needed.

#### Function Location

The wire generation function is located in `utils/wire\_generator.py`.

### Anomalies Generation

The system includes anomalies to enhance dataset variability, simulating real-world conditions such as damaged components, misalignments, or material wear. These anomalies are applied to specific utility pole components through object rotation and material augmentation.

#### Types of Anomalies

1. \*\*Rotational Anomalies\*\*

- Certain objects, such as fuse barrels, ALS, ATS, and insulators, can be rotated out of their correct alignment to simulate open, loosened, or displaced equipment.

2. \*\*Material Augmentation\*\*

- Equipment surfaces can be altered using paint overlays or material deformation within Blender. Using nodes and Blender Paint, these can be procedurally applied.

- This includes rust simulation, charring, faded paint, and physical distortions.

#### Anomalies Management

- \*\*`trackers.py` (Located in `core/`)\*\*

- Keeps track of applied anomalies to ensure proper scene resets between dataset generations.

- Ensures anomalies are randomly distributed while maintaining realistic constraints.

- \*\*`anomalies.py` (Located in `scripts/`)\*\*

- Responsible for applying transformations and material changes.

- Defines the range of rotations, material edits, and structural deformations.

- \*\*`pole\_generation\_config.py` (Located in `config/`)\*\*

- Allows parameterized control over anomaly frequency.

#### How It Works

1. \*\*Scene Initialization\*\*

- The system initializes the pole configuration.

- Objects eligible for anomalies are identified.

2. \*\*Anomaly Application\*\*

- Randomized rotation is applied to objects like fuse barrels.

- Material alterations are introduced through texture blending or shader modifications.

3. \*\*Tracking and Reset\*\*

- All transformations are recorded in `trackers.py`.

- Object label is updated to include "\_Anomaly"

- Before generating the next sample, the scene is reset to prevent anomaly overlay or unintended object persistence.

#### Customizing Anomalies

To modify anomaly behaviors:

- Adjust rotation ranges and probabilities in `config/pole\_generation\_config.yaml`.

- Modify material augmentation rules to introduce new types of damage.

- Ensure all new anomalies are properly tracked in `trackers.py` for correct scene resets.

This system allows for controlled yet randomized\*\* anomaly generation, making datasets more robust for fault detection and predictive maintenance tasks.