Crack segmentation on concrete pipes with YOLO v8 based on synthetic data

Victor Escribano Garcia and Alejandro Acosta Montilla

Abstract—Detecting cracks in materials, particularly concrete, is a critical task for maintaining structural integrity and ensuring safety. This project focuses on the development of a crack segmentation system using the state-of-the-art YOLO v8 model. Given the challenges in obtaining and labeling segmentation data, we employed a novel approach to data generation and labeling. Using Blender, we created a virtual environment to simulate concrete pipes with varying textures, backgrounds, lighting, and camera parameters. Procedural crack image generation was employed to create realistic training data. The trained YOLO v8 model demonstrated effective crack detection, showcasing the potential for advanced automated inspection systems.

Index Terms—Crack segmentation, YOLO v8, concrete inspection, deep learning, synthetic data generation, Blender, procedural generation.

1 Introduction

The detection and segmentation of cracks in materials such as concrete is a vital task in various engineering and maintenance applications. Cracks can indicate structural weaknesses that need to be addressed to prevent potential failures. Traditional methods of crack detection involve manual inspections, which are time-consuming and prone to human error. With advancements in computer vision and deep learning, automated crack detection systems have become a promising alternative.

Bibliography shows that several type of cracks can appear over concrete material with more or less relevance for the structure integrity of the element. Hence, to context the project that is deployed and understand the importance of developing an approach to detect these defects, the cracks that can be found on concrete are mainly depicted below as explained in [1]:

- **Circumferential cracks:** Typically caused by bending forces applied to the pipe. It can be caused by thermal contraction or other interference. It is one of the most common failure mode for smaller diameter pipes.
- **Longitudinal cracks:** More common for large diameter pipes and can be caused by different types of loads applied on the tube such as the internal water pressure or thermal changes. It can be originated by a small crack that is expanded along the pipeline or as a consequence of two cracks originated in opposite locations.
- **Bell splitting:** More common in small diameter pipes and can be caused by wrong joint sealing with thermal expansion coefficients different than the pipe material.
- **Corrosion pitting cracks:** Caused by the reduction of thickness in the pipes' wall so that at the end the internal pressure inside the pipe might cause a hole on its surface.
- **blow-out holes:** Idem as the corrosion pitting cracks, but the water pressure can make a fairly big hole instead of cracks.
- **Spiral cracks:** Occurs in medium diameter pipes and the crack that is originally of type circumferential is propagated along the pipe length in spiral shape. It can be caused by internal pressure combined with bending forces.

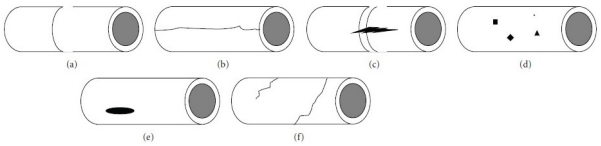

Fig. 1. From [2], representation of different crack types. (a) Circumferential cracking; (b) Longitudinal cracking; (c) Bell splitting; (d) Corrosion pitting; (e) Blow-out hole; (f) Spiral cracking;

On this case, when it comes to the crack detection, the project is focused on just detecting any crack on the surface of concrete pipes, regardless of its severity and the type of crack that can be found. Additionally, it has not been conducted any structural research to determine the distribution of cracks over concrete surfaces or how they are formed.

This project explores the application of the YOLO v8 model, a cutting-edge object detection algorithm, for crack segmentation. Despite the availability of numerous datasets on the internet, labeling data for segmentation tasks remains challenging. To address this, we focused on generating our dataset using Blender, a 3D computer graphics software. By creating a virtual scenario involving concrete pipes as a proof of concept of this synthetic data approach, we were able to manipulate various parameters such as texture, background, lighting, and camera angles. Procedural crack generation techniques were used to produce realistic cracks on the pipe surfaces, providing a robust dataset for training the YOLO v8 model.

2 State of the Art

The field of crack detection and segmentation has seen significant advancements with the advent of deep learning. Traditional image processing techniques relied heavily on edge detection and thresholding methods, which often struggled with varying lighting conditions and noise. Recent approaches leverage convolutional neural networks (CNNs) and their ability to learn hierarchical features from large datasets.

2.1 Traditional Computer Vision Techniques

In the early 2000s, initial methods for detecting cracks on surfaces were developed based on several techniques: threshold methods, morphology methods, percolation theory, and supervised machine learning using random forests and SVM.

- **Threshold Methods:** These methods rely on pixel intensity across the image, using techniques such as OTSU, global thresholding, and local thresholding by dividing the image into small grids and analyzing image entropy. However, these methods can struggle with complex textures and noise.

- **Morphology Methods:** Utilizing edge detection techniques (e.g., Canny, Sobel, FFT) and mathematical operators (dilation, opening, closing), these methods detect crack morphology similar to edges. Advanced approaches use graph theory (e.g., Dijkstra, Minimum Spanning Tree) to close crack contours.

- **Percolation Theory:** Based on the principle that fluid can flow on any surface, this method connects pixels using intensity and shape features of local neighbors, starting from a point and expanding based on a probability model.

- **Supervised Machine Learning:** Methods such as SVM and random forests classify pixels as part of a crack or not, often combining SVM for noise removal with random forests for enhanced detection.

2.2 Segment Anything Model (SAM)

The Segment Anything Model (SAM) is an advanced image segmentation model designed for promptable segmentation, generating valid segmentation masks from prompts like spatial or text clues. While SAM provides versatility and generality, it does not perform as desired for specialized tasks like crack detection due to its lack of specificity.

2.3 U-Nets

Originally designed for biomedical image segmentation, U-Net architectures (including variants like U-Net++ and Attention U-Net) have proven effective for various segmentation tasks, including crack detection. Using transfer learning with architectures such as VGG16 and ResNet101, U-Nets have shown improved performance with large, diverse crack segmentation datasets.

2.4 YOLO (You Only Look Once)

The YOLO series revolutionized object detection with real-time processing and high accuracy. YOLO v8, the latest iteration, incorporates advanced features such as improved anchor box calculations, better feature pyramids, and enhanced training techniques, making it highly suitable for segmentation tasks.

2.5 Synthetic Data Generation

Several studies [3], [4] have highlighted the effectiveness of simulated environments for overcoming labeled data scarcity. Tools like Blender enable the creation of diverse and realistic datasets essential for robust model training. Procedural generation of cracks allows for extensive, varied datasets that enhance the training process.

Our project leverages these advancements by using YOLO v8, a state-of-the-art deep learning model, trained on a high-quality synthetic dataset generated with Blender. This approach ensures effective crack segmentation, demonstrating the potential for advanced automated inspection systems in detecting cracks on concrete surfaces.

3 Methodology

3.1 Data Collection and Preprocessing

In this section, we detail the methodologies employed for data collection and preprocessing. Given the challenges in obtaining and labeling real-world datasets for crack segmentation, we developed a synthetic data generation pipeline using Blender. This synthetic data was then augmented and merged with existing real-world datasets to enhance the robustness and accuracy of our model. Below, we describe the processes involved in creating the synthetic data and the steps taken to integrate it with real datasets.

3.1.1 Synthetic Dataset

To generate the synthetic dataset Blender has been used as main 3D software tool. Thanks to its node shading structure we can create modules to make stitch variations on the elements of the scene and the environment around them.

Synthetic datasets have the advantage of create images and the corresponding label maps for data that might be difficult to obtain, replicate or annotate in real life. On this project we centered on the data generation of procedural cracks on concrete pipes as a proof of concept. Note that this methodology of creating a virtual environment can also be applied not only to pipes but to another objects like, bridges [5], road structures [6], etc. But in this project we are going to center on the synthetic generation on pipes due to the simplicity of the 3D modeling (simple cylinder).

3.1.1.1 **Blender addOns**: There exists several addons on blender for multiple applications, nevertheless on the field of the machine learning for object detection there are some community contributions, but as blender is not mainly designed for this porpoise the contributions are not that much. However there exists a small addon package developed by a single user [7] that was designed to segment defects on objects, our project is based on this implementation as it settled the basis of our development.

3.1.1.2 **Texture definition**: In the synthetic environment we generated the pipe surface alternates between two main textures:

- Gravel Concrete [8]
- Conrete wall [9]

This texture variation provide enough variations for the amount of data that we want to obtain, giving different

Fig. 2. Gravel Concrete texture applied to simle pipe.

Fig. 3. Concrete Wall 007 texture applied to simple pipe.

textured pipes that along with the random variations that are explained along this paper will provide enough variability. Its important to have in mind that this textures are fixed, not procedural. That means that are not subject to any external parameter that makes detail variations in every sample.

3.1.1.3 **Crack Generation**: The generation of cracks on the synthetic pipe surfaces was achieved through a custom script developed in Python, leveraging the 'matplotlib' library. The following outlines the key steps involved in the crack generation process.

To generate realistic crack patterns, we defined a function 'draw_crack' which simulates the appearance of cracks by randomizing various parameters such as noise level, number of points, bifurcations, and line width. The cracks are then plotted onto a blank image which is subsequently used as a texture overlay on the pipe model in Blender.

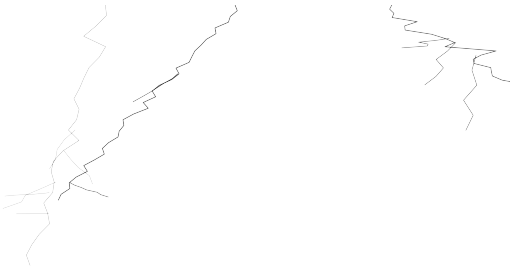

Fig. 4. Sample generated crack image.

The core function, 'draw_crack', is responsible for generating the crack pattern:

Listing 1. Single crack generator

```
def draw_crack(ax, start_point,
               length=10,
               noise_level_range=(10, 50),
               num_points_range=(20, 41),
               min_bifurcations_range=(0, 4),
               max_bifurcations=3,
               line_width_range=(0.5, 3)):
    ...
```

This function draws a single crack by determining a starting point and randomly adjusting parameters to create variations in each crack's appearance, as shown in Figure 4.

The main function, 'generate_crack_image', calls the previously defined function:

Listing 2. Final crack image

```
def generate_crack_image(
                file_path='cracks_image.png',
                image_size=(2048, 2048),
                crack_regions=None,
                crack_length=1024,
                max_bifurcations=5):
    ...
```

This function sets up the image canvas, defines the regions where cracks should be generated, and saves the final image as a PNG file. These images are then mapped onto the pipe models in Blender.

Fig. 5. Crack applied on pipe texture.

3.1.1.4 **Crack texture block**: In order to add a layer of realism to the cracks drawings on the pipe texture it is needed to add depth to the crack surface. Making use of the blender shading capabilities, a bump block has been used. The procedure is simple, as the cracks image is a transparent png file with only the cracks on it when we take the alpha value and connect it to the bump block we can add some strength to the image (crack) and connect that output to the normals of the pipe texture block as it can be seen on figure 6

Also, as obvious it may seem when combing the alfa value of the crack with the concrete textures, using the *mix* block inside the *Real Pipe* frame on figure 6, the colour on the pipe is chosen to be black, as a normal pipe should be. This colour factor mix could be changed along with the material

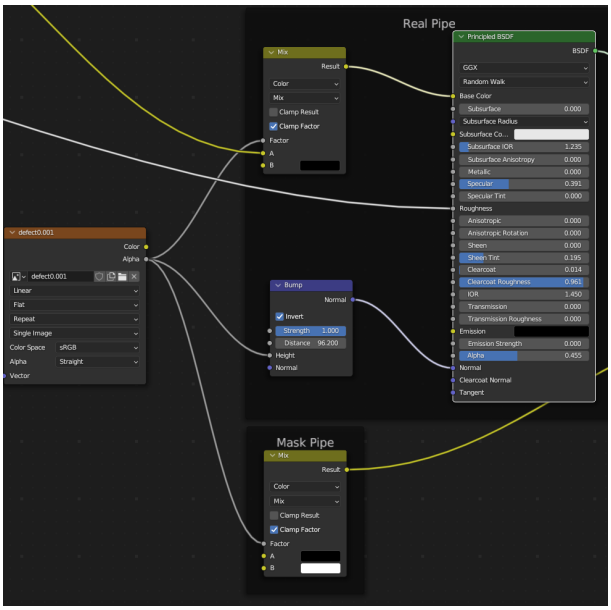

Fig. 6. Bumb block connected between crack texture block and pipe texture block.

in case that the crack was generated on a special coloured surface. With all this combined we give this extra layer of realism from a flat png over the pipe to a crack with the depth extra dimension

Fig. 7. *
(a) Flat crack texture

Fig. 8. *
(b) Crack texture with bump mapping

Fig. 9. Comparison between flat crack texture (a) and crack texture with bump mapping (b).

Notice that there is another *mix* colour block on another frame called *Mask Pipe*, this mix colour will define the colour of the crack and the background of the crack, thus the pipe. It is chosen black for the pipe colour and white for the crack, so when generating the ground truth mask image, the crack would be white along the dark pipe and the dark background (explained in 3.1.1.5)

3.1.1.5 **HDRI background**: To enhance the realism of the synthetic data, we used HDRI (High Dynamic Range Imaging) backgrounds in the Blender environment. HDRI images provide realistic lighting and reflections, which are crucial for creating high-quality synthetic datasets.

Instead of modeling complex backgrounds, which would require significant time and effort, HDRI images allow us to easily incorporate varied and realistic environments into our scenes. This not only saves resources but also introduces a wide range of visual elements that can help our model learn to differentiate between real cracks and other artifacts that might appear similar. For instance, the reflections and textures in an HDRI image can include elements that resemble cracks but are not, thus adding redundancy and robustness to the model.

We utilized two HDRI images to alternate the backgrounds:

- Boiler room: This HDRI image simulates an industrial boiler room environment.

Fig. 10. Boiler room HDRI image used as background.

- Aircraft workshop: This HDRI image provides a workshop setting with various machinery and equipment.

Fig. 11. Aircraft workshop HDRI image used as background.

The HDRI images were integrated into the Blender scene using the environment texture node. This node allows us to switch between different HDRI images easily, thereby providing a diverse set of lighting conditions and environments for our synthetic data. The *mode switch* block facilitates this alternation, as shown in the upcoming sections.

Furthermore, our code includes a parameter that adjusts the lighting of the HDRI in each sample. By tweaking this parameter, we can simulate different times of day and lighting conditions, adding even more variability to our dataset. By incorporating these HDRI backgrounds, we ensure that the synthetic data closely mimics real-world scenarios, thereby improving the robustness and generalization capability of our crack detection model.

3.1.1.6 **Mode switcher block**: The mode switcher blocks in our Blender setup are crucial for introducing variability and control in our synthetic data generation process. These blocks utilize global variables that are randomly altered via our Python script, allowing us to dynamically switch between different backgrounds, textures, and modes (real and ground truth) without manual intervention.

There are two types of mode switcher blocks in our setup, each tailored for specific types of data:

1. **Shader Mode Switcher**: This block is designed to toggle between the real and ground truth modes as can be seen on figure 15. It operates on shader data and is used to switch the output material between the realistic appearance of the scene and the binary mask required for training. The binary mask highlights the cracks, showing them in white against a black background, for later segmentation model learning.

- The mode switcher takes two inputs: the real shader and the ground truth color.
- It outputs the selected mode based on a global variable ('mode') defined in the scene properties.
- The mixing between the real and ground truth modes is controlled by a 'Mix Shader' node, whose input is driven by the 'mode' variable.

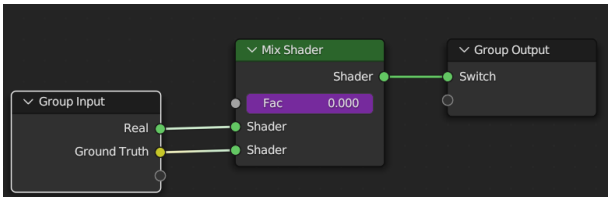

Fig. 12. Node setup for mode switching between real and ground truth.

Fig. 13. * Real image seen when mode switcher mode value is in 0.

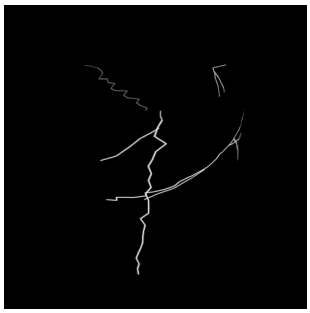

Fig. 14. * Mask image seen when mode switcher mode value is in 1.

Fig. 15. Comparison between real and ground truth image seen depending on the mode switcher value.

2. **Color Mode Switcher**: This block is used for switching between different HDRI backgrounds and textures. Since HDRI images and textures can be treated as color data, this switcher block operates on color inputs.

- The color mode switcher takes two color inputs, allowing us to switch between different HDRI backgrounds or concrete textures.
- The output is a mixed color based on a global variable ('scene_mode') which is randomly set in the script to provide different environments and surface appearances for each sample.
- A 'Mix RGB' node is used to blend the two color inputs based on the 'scene_mode' variable.

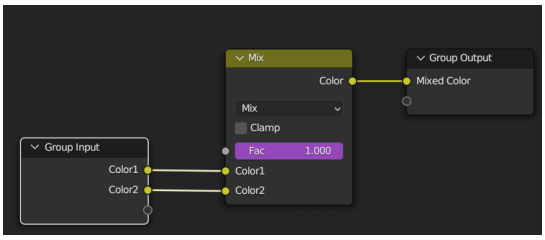

Fig. 16. Background HDRI mode switcher node setup.

The implementation of these mode switchers involves creating custom node groups in Blender's shader editor. The shader mode switcher group includes nodes to mix shader inputs, while the color mode switcher group includes nodes to blend color inputs. The mode and scene_mode variables are defined in the scene properties and are used to drive the mixing operations.

By integrating these mode switchers, we significantly enhance the efficiency and flexibility of our synthetic data generation pipeline. The ability to randomly switch between different configurations ensures a diverse and extensive dataset, which is crucial for training robust machine learning models capable of performing well in various real-world scenarios.

For detailed Python code on how these mode switcher node groups are created, please refer to Appendices A and A.

3.1.2 Existing Annotated Data

To further enhance the robustness and generalization capability of our crack detection model, we incorporated a hybrid dataset consisting of both real and synthetic data. The real dataset comprises 950 images of concrete surfaces with annotated crack segmentation masks, sourced from various publicly available databases on the internet.

The integration of real data into our training set offers several key benefits:

- **Realism**: Real images provide genuine examples of cracks with natural variations in lighting, texture, and noise that are difficult to replicate perfectly in synthetic data.
- **Diversity**: Combining real and synthetic data increases the diversity of the dataset, ensuring that the model is exposed to a wide range of crack appearances and environmental conditions.
- **Robustness**: Training on a hybrid dataset helps the model generalize better to real-world scenarios, improving its robustness and accuracy when deployed in the field.
- **Validation**: Real data can be used to validate the performance of the model, ensuring that it performs well not only on synthetic examples but also on actual images of cracks.

Fig. 17. *
(a) Real image 1

Fig. 18. *
(b) Label map 1

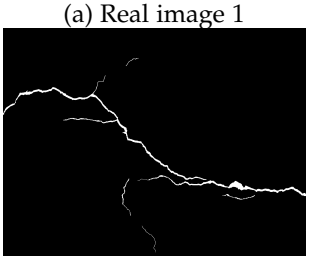

Fig. 19. *
(c) Real image 2

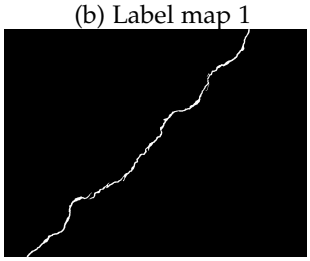

Fig. 20. *
(d) Label map 2

Fig. 21. Examples of real images and their corresponding label maps. (a) and (c) are real images; (b) and (d) are their respective label maps.

The inclusion of 600 real images and 3000 synthetic images in our dataset ensures a comprehensive and varied training set. This hybrid approach takes advantage of the strengths of both data types: the authenticity and complexity of real images and the controlled, extensive variability of synthetic data. By combining these datasets, we create a balanced and effective training set that improves the performance and reliability of our crack detection model in real-world applications.

4 YOLO v8 Segmentation Architecture

The YOLO (You Only Look Once) v8 architecture is a state-of-the-art model for object detection and segmentation. It is designed to be both efficient and accurate, making it suitable for real-time applications. The YOLO v8 model consists of three main components: the backbone, the neck, and the head.

The **backbone** is responsible for extracting essential features from the input image. It consists of several convolutional layers that progressively downsample the input, capturing increasingly complex features. The backbone used in YOLO v8 is often based on CSPDarknet, which includes Cross Stage Partial networks to enhance gradient flow and reduce computation.

The **neck** aggregates features from different stages of the backbone to create a rich feature map that combines both low-level and high-level information. This is achieved using layers like PANet (Path Aggregation Network), which enhances information flow and improves localization capabilities.

The **head** is the final part of the network, responsible for predicting bounding boxes, object classes, and, in the case of segmentation, mask coefficients. YOLO v8 uses anchor-free detection, meaning it does not rely on predefined anchor boxes. Instead, it predicts bounding boxes directly, simplifying the detection process and reducing computational overhead [10].

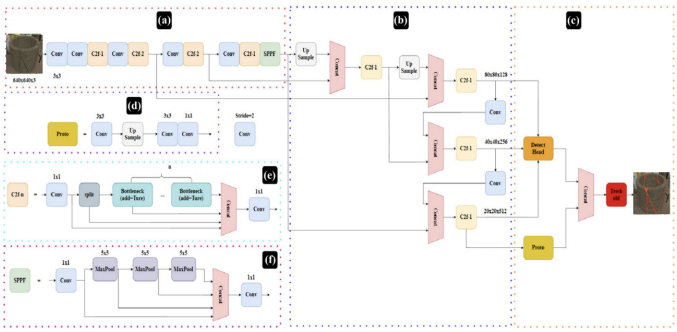

Fig. 22. YOLO v8 Segmentation Architecture. (a) Backbone, (b) Neck, (c) Head, (d) Proto module, (e) C2f module, and (f) SPPF module. [11]

For segmentation tasks, the architecture is similar to the detection model but includes an additional output module in the head for mask coefficients and a Proto module consisting of Fully Convolutional Networks (FCNs) to generate the segmentation masks. This design is inspired by the Yolact architecture.

5 YOLO v8 Training Process

To train the YOLO v8 model on our custom dataset, we needed to convert the binary mask images generated by Blender into the label format required by YOLO. This was achieved using the `masks_to_polygons.py` script. This script reads the binary mask, extracts contours, converts them into polygon coordinates, and saves them in the appropriate label format [12].

5.1 Converting Masks to Polygons

The conversion process involves the following steps:

1) Load the binary mask and get its contours using OpenCV.
2) Convert the contours to polygons.
3) Save the polygons in the YOLO label format. [12]

An example of the conversion process is shown below:

Listing 3. Converting masks to polygons

```
# Convert the contours to polygons
polygons = []
for cnt in contours:
    if cv2.contourArea(cnt) > 100:
        polygon = []
        for point in cnt:
            x, y = point[0]
            polygon.append(x / W)
            polygon.append(y / H)
        polygons.append(polygon)

# Print the polygons
with open('{}.txt'.format(os.path.join(output_dir, j
    )[:-4]), 'w') as f:
    for polygon in polygons:
        for p_, p in enumerate(polygon):
            if p_ == len(polygon) - 1:
                f.write('{}\n'.format(p))
```

```
            elif p_ == 0:
                f.write('0 {} '.format(p))
                else:
                    f.write('{} '.format(p))
```

The YOLO label format specifies each polygon with a class index followed by the normalized coordinates of the polygon vertices:

```
<class-index> <x1> <y1> ... <xn> <yn>
```

Where `<class-index>` is the index of the class (e.g., 0 for cracks), and `<x1> <y1> <x2> <y2> ... <xn> <yn>` are the normalized coordinates of the polygon vertices.

Note that setting an appropriate threshold for `cv2.contourArea` is critical. If the threshold is too low, it can lead to noisy labels and small artifacts being detected as cracks. Fine-tuning this parameter is essential for accurate training. Improper tuning can result in errors, as illustrated in Figure 23.

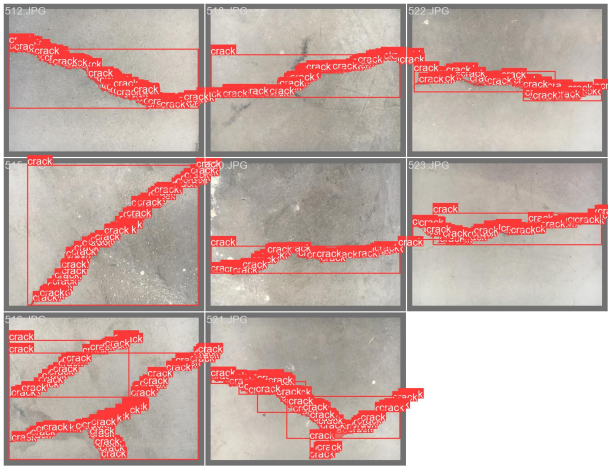

Fig. 23. Example of errors due to improper contour area threshold setting.

Corrected labeled images should look as on figure 24, achieved by tuning the `cv2.contourArea` over 100px.

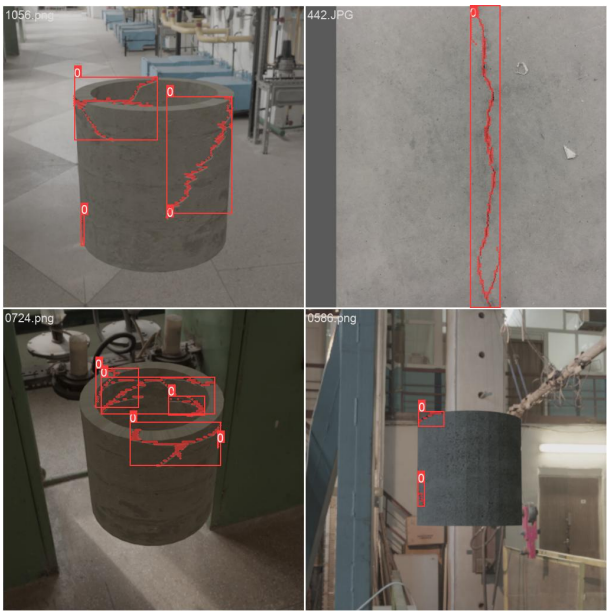

Fig. 24. Example of correct labeling adjusting parameter of contours bigger than 100.

5.2 Training

The training process was configured using the `train.py` script, which loads a pretrained YOLO v8 model, sets the training parameters, and starts the training process. The key training parameters include the number of epochs, image size, batch size, and device (GPU) used for training.

5.2.1 Training Parameters

Key parameters for training the YOLO v8 model include:

- **Data**: Path to the dataset configuration file.
- **Epochs**: Number of training epochs.
- **Image Size**: Size of the input images, set to 512 pixels to match the output size of the Blender-generated images.
- **Batch Size**: Number of samples per batch, reduced to 4 (from the default 16) to lower VRAM consumption during training, making it suitable for our training devices.
- **Device**: GPU device used for training.

The script to initiate the training process is shown below:

Listing 4. Training the YOLO v8 model

```
from ultralytics import YOLO

def train_model():
    model = YOLO('yolov8m-seg.pt')  # load a
        pretrained model
    model.train(data='config.yaml', epochs=100,
        imgsz=512, batch=4, device=0)
```

5.2.2 Model Selection and Hardware Configuration

We used the medium model (yolov8m-seg) due to its balance between performance and computational requirements. The table below shows the different YOLO v8 segmentation models and their specifications:

Model	size (pixels)	mAPval 50-95	mAPmask 50-95	Speed - A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n-seg	640	36.7	30.5	1.21	3.4	12.6
YOLOv8s-seg	640	44.6	36.8	1.47	11.8	42.6
YOLOv8m-seg	640	49.9	40.8	2.18	27.3	110.2
YOLOv8l-seg	640	52.3	42.6	2.79	46	220.5
YOLOv8x-seg	640	53.4	43.4	4.02	71.8	344.1

Fig. 25. Specifications of YOLO v8 segmentation models. [13]

Given the VRAM limitations of our local machines (8GB RTX 3070 Ti and 8GB RTX 3050 Ti laptops), we adjusted the batch size to ensure smooth training without running into memory issues. The medium model provides a good compromise between computational load and segmentation accuracy, making it suitable for training on local hardware.

5.2.3 Data Augmentation

YOLO v8 applies various data augmentation techniques [13] to improve the model's generalization. Key augmentations used during our training include:

- **Mosaic Augmentation**: Combines four images into one, providing context variation and making the model more robust.
- **Random Perspective**: Applies random perspective transformations to simulate different camera angles.
- **Scaling and Flipping**: Randomly scales and flips the images horizontally.

These augmentations help the model to generalize better by exposing it to a wide range of variations during training, that combined with the variability in camera positions, light conditions, procedural cracks and camera internal parameters variation from blender improve its performance on unseen data.

6 Obtained Results

The results from the crack segmentation algorithm are assessed by checking the values obtained on several metrics as depicted in figure 26.

From these results it is clear that the segmentation model is improving continously and getting better to predict the different cracks given since the loss function either in the validation or in the training procedure is descending for the creation of the bounding boxes and for the cracks masks. Additionally, in figure 26 it is compared the precision and the recall metrics for both the bounding boxes (metrics with a B) and the actual segmentation mask applied over the image to predict the crack (metrics with an M). This metric comparison reveals that model is getting more accurate in predicting the crack with respect to the rest of the image as long as the training procedure is conducted.

However, choosing the precision or the recall to assess the performance of the model is always a difficult task. Normally, precision and recall have a trade-off between both and they are intended to be maximized as much as possible without leading to an over-fitting situation. Thus, to tackle this situation, the f1-score is also computed as an evaluation metric. Hence, figure 26 yields that the f1-score is also increased as long as the training procedure is done so that there is no an imbalance between the precision and the recall. This approach might correct the class imbalance given that only one class is provided to the dataset apart from the background and checking if the f1-score is increased and maximized will lead to checking if both precision and recall are maximized.

Finally, it must be commented the results obtained for both the mAP50 and mAP50-90 for either the bounding box and the segmentation mask evaluated. In the case of the metric mAP50, it is clear that both (B and M) are showing an increasing trend during all the training procedure, meaning that the intersection over union for both at 50% of overlap in the model present a high match rate. However, when it comes to checking the same metric changing the overlap between the ground truth and the predicted images, the rate for the segmentation mask is significantly smaller and lower than the rate for the bounding box. However, this can be as a consequence that mapping pixel a pixel and overlap both masks with a perfect match might be more difficult than attempting to overlap 2 bounding boxes. However, due to the big values of the f1-score and the other metrics, this feature is detected but not considered as an issue for the model.

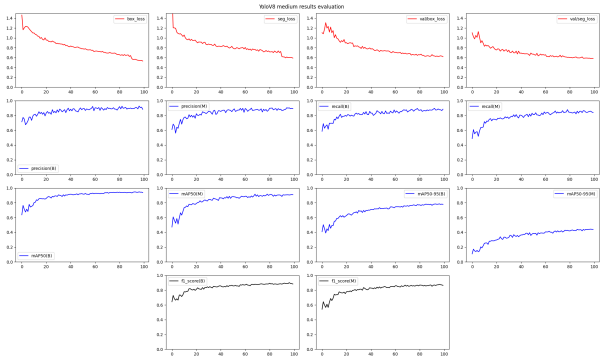

Fig. 26. Metrics evaluation for segmentation model.

All in all, it is clear that the given dataset and the training procedure yielded to a successful segmentation model that allowed to detect the different cracks presented on given images regardless of being artificial or real since it has been trained using both kind of images to enrich and augment the data. The usage of this merged dataset has led to a more augmented data that allowed to the segmentation model to be more invariant and not only rely on the synthetic dataset when it comes to classify if something is a crack, making it more robust against noisy environment. This situation can be proved on figures 27 and 28, where the results from the model using the validation dataset are depicted with some examples over the synthetic images and the real images, showing the power of the model in both situations.

However, from both figures 27 and 28, it is clear that it is sometimes creating more than one bounding box for the same crack, representing that more than one segmentation is done, but it must prevail over this fact that it is only identifying the crack on the model.

Finally, the confusion matrix for the model is obtained from the given data obtained as shown in figure 29. The data given on that matrix yields the model is performing

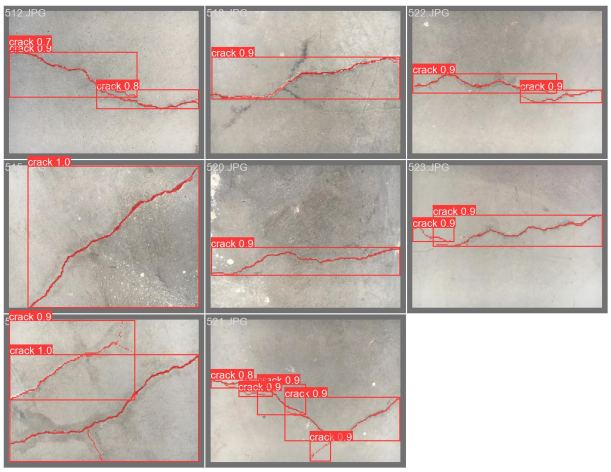

Fig. 27. Segmentation for validation real images.

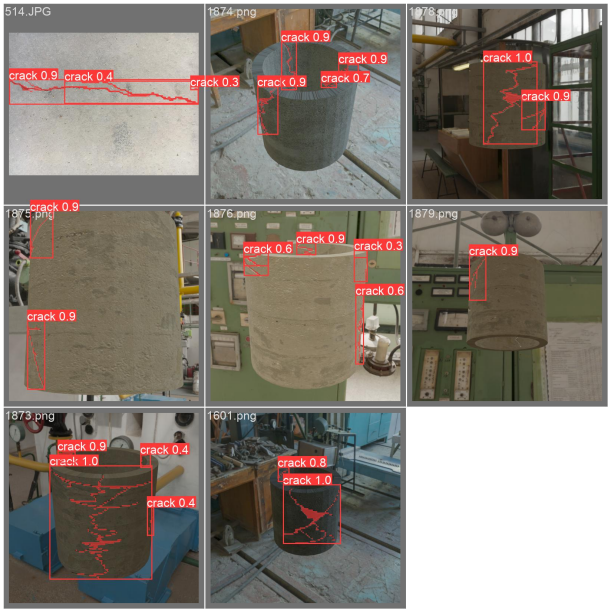

Fig. 28. Segmentation for validation synthetic images.

properly detecting with high rates of precision and recall the true positives and the true negatives with small probability of error.

7 Test procedural cracks with real Dataset

The purpose of the project has been to detect cracks in concrete pipes regardless of its shape, position and the image noise. For this purpose the synthetic dataset has been created to enhance the given real dataset and add some richness and invariant due to the capacity of changing several features when creating the cracks and the images. Nevertheless, there is the need to also compare if the generated dataset is representative to the reality and not just pure synthetic. Hence, another training has been conducted in this case only considering the real images and using on the validation dataset only synthetic images. This second training has been done using all the real images during 60

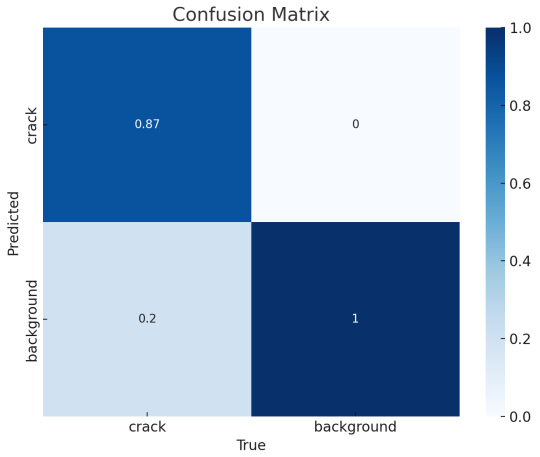

Fig. 29. Confusion matrix for the validation process.

epochs and with the same YoloV8 architecture employed on the previous algorithm and it has been validated using 200 synthetic images.

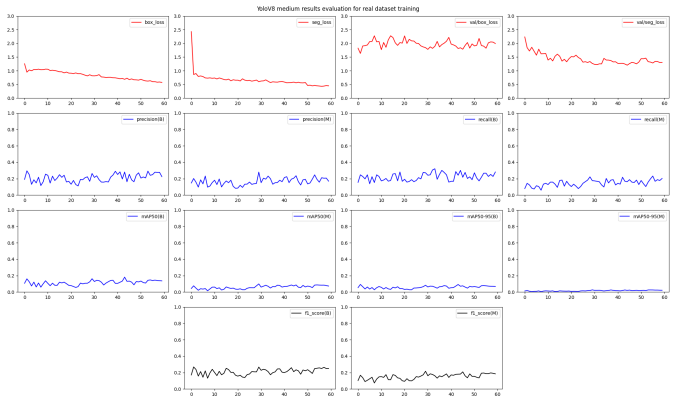

Fig. 30. Metrics evaluation for the segmentation model with real dataset

Results obtained for this training can be found in figure 30, from where it can be understood the classifier is not performing properly on this situation. However, it is not necessarily meaning that the synthetic dataset is not realistic or representative for the reality. Analysing the real dataset it is clear that all real images given are just focused on the cracks with the same plain texture background, which can be good when aiming to detect cracks on this environment, but not for any other noisy or cluttered situations as the ones represented on the synthetic dataset. For this reason, using only the real dataset is not working properly but the fusion of both enhances the results and increases the invariant of the model to detect any crack over any image regardless of its background and shape.

This latter situation described can be compared by checking 31 and 32, where in both images it is clear that the algorithm is able to predict plenty of cracks in the different concrete pipes synthetically generated, but it is also detecting some cracks on the background and other

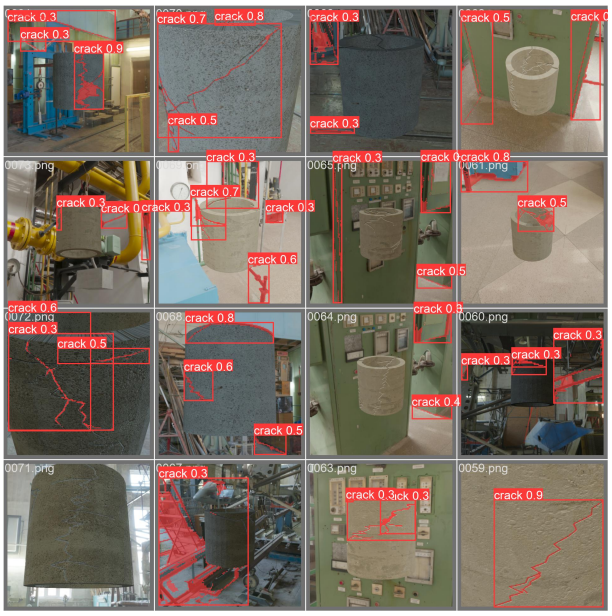

Fig. 31. Predicted results over synthetic images, first example

Fig. 32. Predicted results over synthetic images, second example

scene objects due to the lack of invariant and background variability that the real dataset has. Additionally, different cracks directly present on the pipes are not detected if they are not completely focus on the center of the image or if the crack is not really straight and have too much noise or bifurcations when it has been randomly generated. Nevertheless, the synthetic dataset cannot be considered as not realistic since it clearly improves the segmentation performance over real images and there is a lack of richness in backgrounds and textures for the real dataset.

8 Conclusion

A real State of the Art segmentation algorithm has been presented by means of using the YoloV8 pre-trained neural network with a merge between a real crack concrete and a synthetic dataset created in specific for this project.

It is important to remark the big contribution from Blender to the project since a Blender algorithm for creating and labeling the images has been deployed successfully. Then, Blender capabilities has been exploited along the project, determining that it can be helpful with the data annotation process and the data variation and recreation by means of changing the focal length of the camera, the illumination, the scene, the textures, the perspective and the cracks generated and inserted over the images. Additionally, the generated dataset with the Blender helped enriching the real dataset and adding this lack of variability present on the real one and improving the segmentation algorithm for the cracks in all cases.

When it comes to the YoloV8, the ease of its use has simplified the project development, focusing only on obtaining the best possible outcome on each situation by fine tunning the selected hyper-parameters given in the code. Moreover, the selection of the medium backbone architecture for the project yields that it is enough for performing good estimations and segmentation for the images introduced and helped with the results analysis due to the amount of data provided and the interface given. However, the usage of YoloV8 implied a transformation of the Blender data annotated into the specified Yolo format using a function with a threshold to determine when an amount of pixels must be considered as an annotation to be considered by Yolo. Then, the threshold has been evaluated as a trade-off that must be considered when it comes to refine when a number of pixels shall be treated as a label and its value will make vary the performance of the segmentation model.

All in all, the results yield the crack segmentation algorithm generated using the real and the synthetic dataset merged is really powerful and have big capabilities to detect cracks either on the synthetic environment and on the real dataset despite the small errors in the bounding boxes detected on the validation data. However, the generation of several bounding boxes on the same crack is not an issue to at least detect properly the crack and segment it with several instances.

Additionally, it has been tested the capabilities of the segmentation algorithm training the neural network with only the real dataset and validating it with the synthetic dataset as a possible point to check if the synthetic dataset is representative for the reality. The results from the given test is that the model is not able to perform appropriately in all situations and with several images on cluttered environments, detecting cracks on the background many times apart from the ones on the concrete pipe and missing the detection of several cracks on the pipe itself. Nevertheless, it is not meaning that it is not realistic enough, but it has been determined that it is an issue related with the lack of variability present in the real dataset, the lack of textures and background variability and the small amount of images (around 600) used for the training on this case. Then, due to all these reasons, it cannot be determined that the synthetic dataset is not realistic and it has been proved instead that it helps improving the performance of the model and its robustness on different environments.

Finally, as further research for the project, it could im-

prove by detecting and labelling not just the fact of having a crack or not in the image but also yielding the typology of the crack detected over the concrete surface. Moreover, structural research should be conducted to determine how the cracks are generated over the concrete pipes and the standard shapes they might have to increase dataset reliability and be more realistic. Last, it could be interesting on also obtaining features from the crack such as its size or depth given an image apart from its type and severity.

9 Code

All codes provided for the development of this project can be found on the following GitHub repository: Synthetic-Pipe-Crack-Dataset-Blender

Appendix

Appendix A

TABLE 1
Pros and Cons of Crack Detection Techniques

Technique	Pros	Cons
Threshold Methods	Simple and easy to implement Computationally efficient	Sensitive to noise and texture variations Struggles with complex backgrounds
Morphology Methods	Effective for detecting edge-like structures Can utilize advanced graph theory techniques	May alter crack features Requires image enhancement for better performance
Percolation Theory	Models fluid flow to detect cracks Effective for growing crack contours	Computationally intensive Dependent on accurate probability models
Supervised Machine Learning	More robust to noise Can handle complex textures and backgrounds	Requires labeled training data Computationally expensive to train
Segment Anything Model (SAM)	Highly versatile and general Can generate segmentation masks from various prompts	Lacks specificity for crack detection May not perform well on specialized tasks
U-Nets	Highly effective for segmentation tasks Improved performance with large datasets	Requires significant computational resources Performance depends on dataset quality
YOLO (You Only Look Once)	Real-time processing capability High accuracy with advanced features	Requires extensive training data Computationally demanding for training

Listing 5. Creating the Shader Mode Switcher node group

```
def create_mode_switcher_node_group():
    if 'mode_switcher' not in bpy.data.node_groups:
        test_group = bpy.data.node_groups.new('
            mode_switcher', 'ShaderNodeTree')
        group_inputs = test_group.nodes.new('
            NodeGroupInput')
        group_outputs = test_group.nodes.new('
            NodeGroupOutput')
        node_mix = test_group.nodes.new('
            ShaderNodeMixShader')
        modeDriver = bpy.data.node_groups['
            mode_switcher'].driver_add('nodes["Mix
            Shader"].inputs[0].default_value')
        modeDriver.driver.expression = 'mode'
        modeVariable = modeDriver.driver.variables.
            new()
        modeVariable.name = 'mode'
        modeVariable.type = 'SINGLE_PROP'
        modeVariable.targets[0].id_type = 'SCENE'
        modeVariable.targets[0].id = bpy.data.scenes
            ['Scene']
        modeVariable.targets[0].data_path = '
            uv_holographics.mode'
        # Link inputs and output
        test_group.links.new(group_inputs.outputs['
            Real'], node_mix.inputs[1])
        test_group.links.new(group_inputs.outputs['
            Ground Truth'], node_mix.inputs[2])
        test_group.links.new(node_mix.outputs[0],
            group_outputs.inputs['Switch'])
```

Listing 6. Creating the Color Mode Switcher node group

```
def create_scene_mode_switcher_node_group():
    if 'color_mode_switcher' not in bpy.data.
        node_groups:
        test_group = bpy.data.node_groups.new('
            color_mode_switcher', 'ShaderNodeTree')
        group_inputs = test_group.nodes.new('
            NodeGroupInput')
        group_outputs = test_group.nodes.new('
            NodeGroupOutput')
        node_mix_rgb = test_group.nodes.new('
            ShaderNodeMixRGB')
        modeDriver = test_group.driver_add(f'nodes
            ["{node_mix_rgb.name}"].inputs[0].
            default_value')
        modeDriver.driver.expression = 'scene_mode'
        modeVariable = modeDriver.driver.variables.
            new()
        modeVariable.name = 'scene_mode'
        modeVariable.type = 'SINGLE_PROP'
        modeVariable.targets[0].id_type = 'SCENE'
        modeVariable.targets[0].id = bpy.data.scenes
            ['Scene']
        modeVariable.targets[0].data_path = '
            uv_holographics.scene_mode'
        # Link inputs and output
        test_group.links.new(group_inputs.outputs['
            Color1'], node_mix_rgb.inputs[1])
        test_group.links.new(group_inputs.outputs['
            Color2'], node_mix_rgb.inputs[2])
        test_group.links.new(node_mix_rgb.outputs
            [0], group_outputs.inputs['Mixed Color'
            ])
```

References

1. MISIUNAS, Dalius. *Failure Monitoring and Asset Condition Assessment in Water Supply Systems*. 2008.
2. RIZZO, Piervincenzo. *Water and Wastewater Pipe Nondestructive Evaluation and Health Monitoring: A Review*. Vol. 2010. 2010. Available from DOI: 10.1155/2010/818597.

3. LU, Yingzhou; SHEN, Minjie; WANG, Huazheng; WANG, Xiao; RECHEM, Capucine van; FU, Tianfan; WEI, Wenqi. *Machine Learning for Synthetic Data Generation: A Review*. 2024. Available from arXiv: 2302.04062 [cs.LG].
4. ANDERSON, Jason W.; ZIOLKOWSKI, Marcin; KENNEDY, Ken; APON, Amy W. *Synthetic Image Data for Deep Learning*. 2022. Available from arXiv: 2212.06232 [cs.CV].
5. STRAZIMIRI, Ted. *Cracks and efflorescence detection*. Available also from: https://www.linkedin.com/posts/tedstrazimiri_drones-ai-3d-activity-7018613223591464961-U1dK/?utm_source=share&utm_medium=member_desktop.
6. HUYKHOI2407. *Procedural Road Generator Geometry Nodes*. Available also from: https://blendermarket.com/products/procedural-road-generator-geometry-nodes.
7. RVORIAS. *uvHolographics*. Available also from: https://github.com/rvorias/uvHolographics.
8. BARRESI, Charlotte Baglioni Dario. *Gravel Concrete*. Available also from: https://polyhaven.com/a/gravel_concrete.
9. CILLIERS, Charlotte Baglioni Dario Barresi Rico. *Gravel Concrete*. Available also from: https://polyhaven.com/a/concrete_wall_007.
10. MEDIUM. *How to train YOLOv8 instance segmentation on a custom dataset*. Available also from: https://gdemarcq.medium.com/how-to-train-yolov8-instance-segmentation-on-a-custom-dataset-276d44fa92fc.
11. UYGUN, Tahsin; OZGUVEN, Mehmet Metin. *Determination of tomato leafminer: Tuta absoluta (Meyrick) (Lepidoptera: Gelechiidae) damage on tomato using deep learning instance segmentation method*. Available also from: https://www.researchgate.net/publication/379569004_Determination_of_tomato_leafminer_Tuta_absoluta_Meyrick_Lepidoptera_Gelechiidae_damage_on_tomato_using_deep_learning_instance_segmentation_method.
12. ULTRALYTICS. *Ultralytics YOLO format*. Available also from: https://docs.ultralytics.com/datasets/segment/.
13. ULTRALITICS. *Ultralitics train*. Available also from: https://docs.ultralytics.com/modes/train/#train-settings.
14. ESCRIBANO, Victor; ACOSTA, Alejandro. *Synthetic-Pipe-Crack-Dataset-Blender*. Available also from: https://github.com/VictorEscribano/Synthetic-Pipe-Crack-Dataset-Blender.