university of groningen / faculty of science and engineering

Generating synthetic images featuring cracks in masonry surfaces using Unreal Engine 5

Internship Project Report

Vlad Muscoi, s4718267, v.n.muscoi@student.rug.nl,
Supervisors: Jiří Kosinka, David Hidde Boerema, Ihsan Engin Bal

Abstract: Masonry structures constitute a significant portion of our architectural heritage. Timely crack detection is essential to preserving their structural integrity. Current inspection methods rely predominantly on manual assessments, which are time-consuming, expensive, and subjective. Recent advances in deep convolutional neural networks (CNNs) have significantly improved automated crack detection. However, the principal limitation remains the availability of annotated data. To address this, a recent study has employed Blender's 3D engine to procedurally generate synthetic masonry datasets. While augmenting up to approximately 30% of a training set with synthetic images yields a small drop in F1 score, further increases lead to a marked decline in overall performance. This degradation is likely due to limited scene diversity in the synthetic data.

In this study, we introduce a novel framework built in Unreal Engine 5 to enhance the building of virtual scenes for use in generating synthetic masonry datasets. By leveraging the engine's advanced features—including Nanite for high-fidelity displacement, Blueprints and Materials for procedural crack generation, and the Procedural Content Generation (PCG) system for automated placement of scene clutter—we aim to improve both the realism and variability of synthetic datasets. This framework offers a scalable and flexible approach to producing training data for deep learning-based crack detection models.

1 Introduction

Masonry, from ancient cathedrals to modern brick façades, comprises the bulk of our built heritage but ages poorly under environmental loads and seismic events. Detecting cracks early is crucial to safeguarding both structural integrity and historic value. Yet today's standard of inspecting such structures manually is slow, costly, and subjective (Phares et al., 2004; Laefer et al., 2010). Inspectors face access challenges on ornate or high façades, and post-disaster scenarios only amplify the backlog of urgent assessments.

Deep convolutional neural networks (CNNs) have reshaped automated crack detection on uniform surfaces like concrete and asphalt (Zhang et al., 2018). By learning hierarchical features end-to-end, these models now classify image patches at 95% accuracy and segment cracks at nearly 80% F1 score even against complex backgrounds (Dais et al., 2021). But the key bottleneck remains data: CNNs crave thousands of pixel-perfect examples of masonry—brick and mortar, weathering, occlusions—to generalize reliably such that the model is able to detect cracks within unseen data. Gathering and annotating such rich datasets from protected heritage sites is both impractical and expensive.

To advance crack segmentation performance, recent work explored the use of Blender's 3D engine to procedurally generate synthetic masonry scenes (Boerema et al., 2025). The proposed pipeline used a surface-aware crack generation algorithm to trace realistic trajectories along mortar joints, apply them to displacement maps, and render diverse wall textures under varying lighting conditions. Incorporating up to $1,000$ synthetic Blender-rendered images—approximately 30% of a $2,434$-patch training set—allowed a substantial reduction in manual annotation effort with minimal performance loss (under 2% drop in F1 score). However, when synthetic

data dominated the training set, segmentation performance deteriorated significantly. Models trained on over 2,000 synthetic patches experienced a decline of more than 40% in F1 score, primarily due to reduced recall. These results highlight the need for more representative and diverse synthetic data to ensure generalization. Limitations such as limited scene variability, simplistic crack modeling, and discrepancies in image composition likely contributed to the domain gap. Nonetheless, the study confirmed that synthetic data can meaningfully support model training when integrated wisely, pointing toward further refinement as a pathway for improving segmentation outcomes.

In this study, we develop a custom framework within Unreal Engine 5, a state-of-the-art game development platform (VG Insights, 2025), to facilitate controlled generation of synthetic datasets. Unreal Engine provides an extensive suite of tools and features tailored for creating complex virtual environments and dynamic interactions (Epic Games, 2025). Key functionalities leveraged in this work include high-fidelity geometry rendering through Nanite and procedural environment construction using the Procedural Content Generator (PCG). By integrating these capabilities with Unreal Engine's visual scripting system and material editor, we establish a framework capable of building varied and realistic virtual scenes featuring masonry surfaces with synthetic cracks. From these scenes, RGB images and their corresponding pixel-accurate labels can be extracted to construct synthetic datasets, which can be used to augment real-world training data. The design and implementation of this framework are detailed in the following sections.

This study is guided by the following research question, which aims to evaluate the feasibility, quality, and usability of Unreal Engine 5 for generating synthetic crack datasets, in comparison to a previously established pipeline based on Blender:

Can synthetic crack images generated using Unreal Engine match or exceed those produced with Blender in terms of visual realism and their effectiveness in improving crack segmentation performance?

To address this question, datasets generated using the Unreal Engine framework are used to retrain crack detection models previously trained on Blender-generated data. The resulting performance metrics are then compared to evaluate differences in data quality and their impact on model segmentation performance.

2 Theoretical framework

Building upon the prior work of Dais et al. (2021) in crack detection using real masonry images, Boerema et al. (2025) propose a novel framework for the automatic generation of synthetic crack datasets using 3D rendering in Blender. The motivation lies in the difficulty of acquiring and annotating diverse, high-quality real-world masonry crack images, which are essential for training effective deep learning models. The framework of Boerema et al. (2025) addresses this by automating the creation of realistic masonry environments and embedding surface-aware crack patterns directly into them.

The dataset generation pipeline consists of four main steps:

1. **Randomization** - setting random parameters for the steps to come;
2. **Crack generation** - generating a crack which will be used within the scene;
3. **Scene preparation** - varies the objects present in the scene for diversity as well as apply crack to wall surface;
4. **Rendering** - generating the images along side their labels.

Each iteration of the pipeline begins with randomizing camera positions, environmental lighting and crackable surfaces. The core of the framework is a surface-aware, based on semi-realistic behaviour, crack generation algorithm, which models cracks as shortest paths over a displacement-informed surface map. Pivot points are randomly placed across the surface, and the crack path is calculated by blending gradient directions from the displacement map with direction vectors toward the next pivot point. Crack width is also dynamically varied along the path. Figure 2.1 provides an overview of how the pivot points are generated.

Once the crack path is generated, it is converted into a 2D displacement map representing crack depth using a Gaussian profile. This depth map

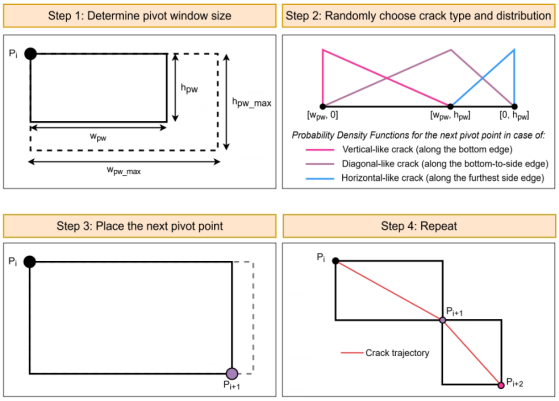

Figure 2.1: Overview of the pivot point generation. Image taken from Boerema et al. (2025).

is then integrated into the original surface's displacement texture. To avoid rendering artifacts, the area surrounding the crack is slightly blurred. After composing the scene with only relevant geometry visible, Blender's path tracer is used to render images. Labels are generated via a postprocessing pipeline that isolates crack regions from rendered images using compositing and morphological operations. The framework supports both single-image and patch-based rendering for higher efficiency and detail.

The authors generated a synthetic dataset consisting of 4,058 images using Blender, applying their framework to 3D scenes built from publicly available assets. These scenes featured four crackable masonry walls and were illuminated with four different environmental lighting to introduce variability.

To evaluate the effectiveness of the synthetic data, several convolutional neural network (CNN) architectures were tested: U-Net (Ros et al., 2016) and Feature Pyramid Networks (FPN) (Lin et al., 2017), each with MobileNet (Howard et al., 2017) and InceptionV3 (Szegedy et al., 2016) as backbone networks. The models were trained on combinations of real and synthetic data, with the real dataset (Dais et al., 2021) consisting of 4,058 labeled image patches. A filtered version of the real dataset, containing 3,044 cleaner patches, was also used in experiments.

Performance was assessed using F1 score, precision, and recall, with all models validated exclusively on real data to test generalization. When using only synthetic data (0% real), models achieved high training accuracy (F1 scores around 88.8%) but it does not fully generalize to the real data. However, blending synthetic data into real datasets in small proportions improved training diversity by allowing the augmentation of the dataset with synthetic data without severely degrading performance. For instance, the U-Net-MobileNet model trained on the original real dataset (F1 = 70.4%) experienced only a minor drop in F1 when 500 synthetic samples were added (F1 = 68.9%, a decrease of 1.47%). Similarly, using the filtered dataset with 500 synthetic samples resulted in a minimal F1 difference of just –0.92%.

Unfortunately, adding larger proportions of synthetic data (e.g., a 1:1 or higher ratio of synthetic to real) led to poor generalization, with F1 scores dropping below 30% for mixed datasets containing over 2000 synthetic samples.

Despite promising results, the data generated by the framework has several limitations. It currently does not feature crack branching, includes limited scene variation (e.g., no foliage or debris), and exhibits framing inconsistencies between real and synthetic data. These factors reduce generalization when synthetic data dominates training.

3 Methods

While recent work has demonstrated the value of synthetic data generated using Blender for masonry crack detection, its limitations, including scene diversity, generation speed, and integration flexibility, raise questions about its scalability for broader applications. To address these challenges, this study explores the use of Unreal Engine 5, a modern real-time rendering platform that offers advanced capabilities such as procedural content generation (PCG), dynamic material systems, and high-fidelity scene rendering. By leveraging these features, we propose a framework for efficiently generating realistic, labeled synthetic masonry scenes, aiming to both reduce reliance on manual annotation and improve model generalization.

The Unreal Engine (UE) framework developed in this study consists of several interconnected components. The following subsections provide an overview of each key component, along with brief

descriptions of the tools and technologies employed in their implementation.

3.1 General Overview

The framework developed in this study is structured around three core components, each fulfilling a distinct role in the synthetic data generation pipeline:

- **Crack generator - The Brick Wall**: An actor containing a static mesh component representing a brick wall, to which a synthetic crack texture is applied via displacement mapping. This component is primarily developed using Unreal Engine's visual scripting system, Blueprints, in combination with the Material Editor for procedural shader and displacement logic as well as custom C++ code.
- **Camera Blueprint - The Camera**: A virtual camera system responsible for capturing rendered RGB frames and generating corresponding label images for use in training crack detection models. Its functionality is implemented predominantly through Blueprints, leveraging Unreal Engine's rendering capabilities.
- **Procedural environment generator - PGC Planes**: A system that populates the 3D environment with predefined meshes according to configurable spatial rules. This component enhances scene variety and realism, and is implemented using a combination of Blueprints and Unreal Engine's Procedural Content Generation (PCG) framework.

The three main components interact to form a cohesive pipeline: the crack generator produces surface-level defects, the capture component records labeled imagery, and the environment generator ensures visual diversity by modifying the scene layout procedurally. Figure 3.1 provides an overview of how these components are placed within the engine.

3.2 Crack generator

The crack generator is implemented as a Blueprint that encapsulates a static mesh representing a brick wall (modeled as a box), a Procedural Content Generation (PCG) component, and the logic required to synthesize and apply cracks. The generation process is initiated by a communication link with the camera blueprint, which issues crack regeneration commands during dataset capture.

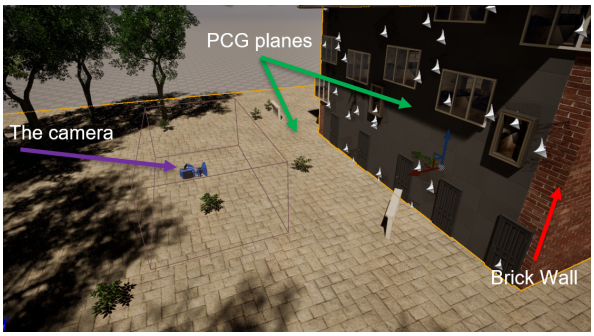

Figure 3.1: An annotated snapshot of the engine viewport.

Upon activation, the generator selects two points on a designated face of the cube mesh to define the trajectory of a new crack. The first point is randomly sampled within the face boundaries, while the second is determined by independently sampling values from user-defined intervals along the x and y axes, thus constraining the direction and length of the crack.

A dynamic material instance is then created to allow real-time updates to key parameters used in rendering the crack. This material supports the following core properties:

- **Mortar mask**: Similar to the surface mask used in the Blender approach, the mortar mask is a precomputed binary 2D texture derived from the displacement map of the brick wall, where white pixels represent mortar lines and black pixels denote brick regions. This mask informs the crack generation algorithm about surface structure.
- **Crack Texture**: A procedurally generated binary texture that encodes the shape of the crack. It is positioned on the mesh using two world-space vectors, which ensure proper alignment between the crack and the mesh surface.
- **Emissive control**: The material includes an emissive channel that can be programmatically

toggled to emit light from pixels defined by the crack texture or the mortar mask. This feature is crucial for producing clean label images, as it enables the capture of isolated crack masks without interference from scene lighting.

Following material initialization, the crack generator prepares a render target to isolate and process the portion of the mesh surface relevant to the crack. A render target functions as a texture that a scene capture component can write to, enabling off-screen rendering of specific scene elements. In this framework, it is used in combination with a 2D Scene Capture Component to extract the region of the mortar mask covering the selected start and end points of the crack.

The resulting mortar mask, as well as the positions of the start and end point in local space, serves as input for the crack path computation. A modified version of the A* pathfinding algorithm (Hart et al., 1968), tailored to operate on the binary mortar map, computes the shortest path between the two points. A* was selected due to its ease of use and implementation, its flexibility in behavior modification to suit the task, and its computational efficiency. To introduce variability and reduce visual regularity, stochastic perturbations are added at each decision step, encouraging the formation of jagged, non-linear trajectories that better resemble real-world cracks.

Once the path is determined, a Gaussian blur is applied with a sigma value that changes by small amounts each step to control the local intensity of the blur to contribute to a more naturalistic transition across crack edges. The final output is a grayscale texture encoding the crack's shape and intensity, which is sent back to the dynamic material. This texture modulates the mesh's displacement and emissive properties, and is used for generating both rendered images and corresponding label masks. Label masks are produced by first disabling all light-emitting objects in the scene and then configuring the material to render the crack texture as a white emissive region against a black background. This produces pixel perfect labels describing the exact position of the crack within the scene including occlusions such as objects covering the crack or the displacement of the wall.

Figure 3.2 depicts the crack texture generation steps starting from the render target.

Upon completion of the crack generation process, the generator triggers an event to notify the camera blueprint that the scene is ready for capture. Alongside this event, the generator transmits the world-space coordinates of the crack's center, allowing the camera blueprint to orient itself accordingly and ensure the crack remains the focal point of the rendered image.

3.3 Camera Blueprint

The Camera Blueprint manages the overall dataset generation workflow. Once both the crack and the procedural elements have been successfully generated, the camera initiates the rendering process, producing both an RGB image and its corresponding label.

Structurally, the blueprint contains two camera components: a CineCamera and a SceneCaptureComponent2D. The CineCamera simulates real-world camera systems and provides fine control over parameters such as focal length, aperture, and sensor type, enabling realistic image synthesis. In contrast, the SceneCaptureComponent is used specifically for generating label images through controlled, lighting-independent captures.

The logic begins with the camera being placed at a randomized location within a defined volume. It then awaits a signal from the crack generator, which includes the world-space coordinates of the crack's center. Using this information, the camera is oriented to focus directly on the crack. After orientation is complete, the camera waits for the PCG system to finish populating the environment.

Since PCG objects are placed independently of the camera position, occlusions may occur where objects block the view of the crack. To address this, a sphere trace is performed between the camera and the crack location, detecting any intersecting components. Undesired occluding elements are then selectively hidden from view prior to image capture creating direct line of site between the camera and the crack.

Rendering proceeds in two stages. First, the CineCamera captures the full-resolution RGB image at 3840×2160 pixels, saving it to disk. Then, to generate the label image, the SceneCaptureComponent disables all dynamic lighting in the scene and activates the emissive property of the crack material, causing only the crack to emit visible light.

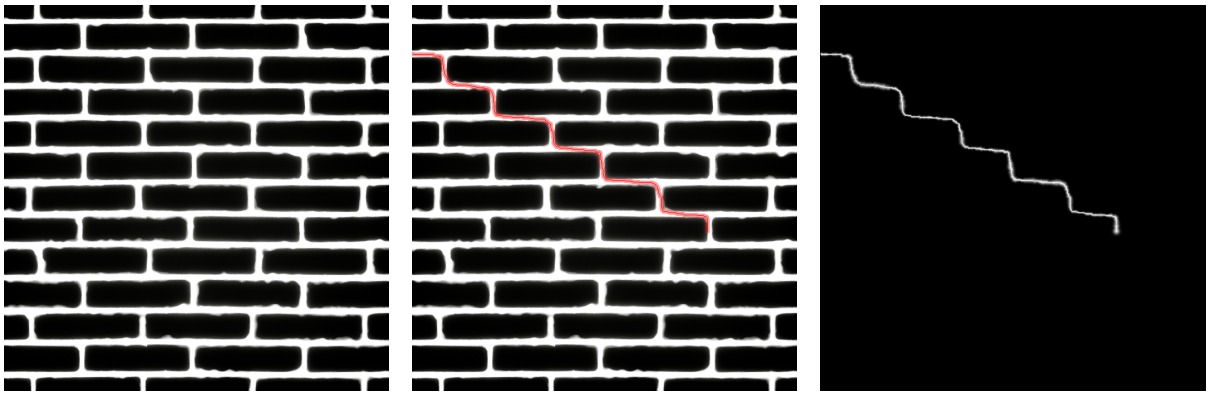

Figure 3.2: Overview of the crack generation steps: (left) captured mortar mask, (middle) captured mortar mask with the computed path in red, (right) final crack texture after Gaussian blur.

This results in a binary image where the crack appears as white on a black background. Once the label is captured, the original lighting and material properties are restored.

Finally, the camera checks whether the target number of images has been reached. If not, it issues a regeneration command, prompting the crack generator, PCG components, and the camera itself to reset and initiate a new iteration of the dataset generation loop. Figure 3.3 shows a generated image with its corresponding label.

3.4 Procedural environment generator

The Procedural Environment Generator is not implemented as a singular, standalone blueprint but rather as a collection of modular components embedded within other actors in the scene. These components collectively fulfill the role of introducing visual complexity, which is crucial for improving the environmental variability of the generated dataset.

At the core of each generator component is a plane mesh, typically subdivided into a grid of vertices that serve as a sampling surface for the PCG component. This component leverages a user-defined PCG Graph to control the spatial distribution of objects within the environment. During execution, the PCG component transforms the plane's vertices into world-space sampling points, which are then evaluated according to the logic defined in the PCG Graph. These candidate points can be filtered based on spatial constraints.

To prevent visual clutter or overlapping geometry, bounds-based conflict resolution can be applied to ensure that spawned elements maintain minimum spacing. Finally, the remaining valid points are used to instantiate actors or static meshes within the scene—such as debris, background elements, or occluders—which contribute to greater scene diversity and complexity.

Two distinct PCG graphs were developed to serve different purposes within the procedural environment generation framework. The first graph is dedicated to augmenting the brick wall surface and is responsible for placing visual noise elements such as windows, doors, and dirt decals. Elements that can be placed by the PCG components are meshes, actors and decals the sources of which are listed in Appendix C

Some of the objects placed by the wall-face PCG graph may partially overlap with the crack. To prevent these elements from occluding critical parts of the crack to an extent that would cause generated data to contain little to no crack surfaces, custom filtering logic is implemented. This logic evaluates object placement relative to the crack's location and enforces spatial constraints that ensure the crack remains sufficiently visible, while still allowing for the presence of realistic partial occlusions.

The second graph focuses on the broader environment surrounding the target surface. It procedurally distributes natural elements such as trees and bushes, which not only diversify the scene composition but also introduce complex dynamic shadows cast onto the crack surface.

Figure 3.3: Example pair of a generated RGB image with its corresponding label.

By leveraging PCG in this manner, the framework supports the rapid generation of varied and non-repetitive environments, essential for diverse dataset generation. While the PCG graph allows for rule-based control to enforce scene realism, the current implementation relies on largely random object placement, subject to basic constraints to prevent visually implausible configurations—such as floating doors or trees intersecting the wall geometry.

3.5 Experiments setup

To investigate the effectiveness of the synthetic dataset, three distinct training configurations were evaluated, each designed to quantify the impact of synthetic data when used alone or in combination with real data:

- **100% Synthetic Data:** The model was trained exclusively on a synthetic dataset composed of 2,000 training images and validated using 800 additional synthetic images. This configuration served to evaluate the baseline performance of models trained solely on procedurally generated data.
- **66% Real / 33% Synthetic Data:** A mixed dataset was created using 2,000 real training images supplemented with 1,000 synthetic training images. The validation set consisted of 800 real images, allowing for an assessment of how synthetic data augments real data during training without influencing validation performance directly.
- **50% Real / 50% Synthetic Data:** This configuration included an equal distribution of 2,000 real and 2,000 synthetic images in the training set. Validation was again performed using 800 real images to measure generalization to real-world data.

The validation sets used for the 66%/33% and 50%/50% experiments contain different real images.

All experiments were conducted under identical conditions, with training performed over 300 epochs using consistent model architecture, learning rate of 0.0005, Weighted Cross Entropy loss and optimization parameters Adam, with a batch size of 4. The model used is U-Net (Ros et al., 2016) with Mobilenet (Howard et al., 2017) as its backbone. These parameters have been chosen to be consistent with the results obtained running the model on the Blender dataset. It is important to note that, since the collection of the blender data in the study conducted by Boerema et al. (2025), the code-base has undergone changes, thus the following results should be compared with caution.

4 Results

The results of the experiments conducted in this study, as well as relevant results from the study conducted by Boerema et al. (2025), are presented in Table 4.1. Due to technical issues and time limitations, it was not possible to retrain the model on the Blender dataset. As a result, the Blender-based performance metrics presented here are those previously published in Boerema et al. (2025), which allows for a relative evaluation of the performance differences between the Unreal Engine and Blender frameworks. However, it is important to note that

Table 4.1: Results of training the neural network on datasets composed of real, synthetic (Unreal Engine / UE or Blender), or mixed data. Results from Blender datasets as well as on the Original dataset taken from Boerema et al. (2025). Cells highlighted in gray indicate the better score between the two frameworks within each experimental pairing.

Dataset	Real : Synthetic	F1 (%)	Precision (%)	Recall (%)	F1 Diff
Real-Only Baseline					
Original	2434:0	70.40	71.71	71.84	—
Synthetic-Only Datasets					
(UE) Synthetic	0:2000	88.22	81.36	96.59	—
(Blender) Synthetic	0:2434	88.77	89.52	88.58	—
Mixed Datasets – Approx. 66/33 Split					
(UE) Mix-66/33	2000:1000	63.35	63.71	69.32	-7.05
(Blender) Mixed-1000	2434:1000	68.54	78.06	63.76	-1.87
Mixed Datasets – Approx. 50/50 Split					
(UE) Mix-50/50	2000:2000	63.05	59.85	75.21	-7.35
(Blender) Mixed-2000	2434:2000	28.84	92.09	18.41	-41.56

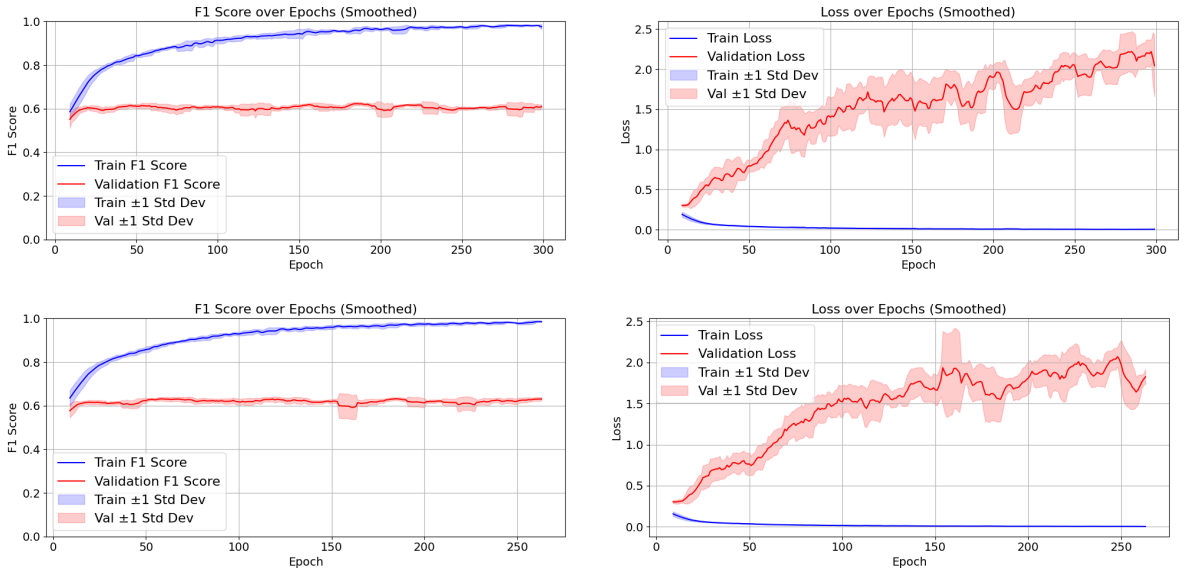

Figure 4.1: F1 score and Loss of the 66%/33% (top row) and 50%/50% (bottom row) datasets over training and validation

this is not a direct comparison, and therefore, the findings should be interpreted with caution.

The synthetic-only results show that both the Unreal Engine (UE) and Blender frameworks can generate high-quality data. Blender achieves slightly better F1 (88.77%) and precision (89.52%), while UE leads in recall (96.59% vs. 88.58%). This suggests that UE-based data results in models that are more sensitive to cracks, whereas Blender produces more conservative but precise predictions.

In the 66% real / 33% synthetic configuration, Blender again outperforms UE in F1 score (68.54% vs. 63.35%) and precision, while UE maintains higher recall.

In the 50/50 mixed setup, the performance gap widens. Blender's F1 score drops significantly (28.84%), with very low recall (18.41%), despite maintaining high precision. UE, on the other hand, preserves a higher F1 (63.05%) and recall (75.21%), showing greater robustness at high synthetic data ratios.

Figure 4.1 shows that the training F1 score steadily increases toward 1.0, while the training loss—computed using weighted cross-entropy—consistently decreases. This indicates effective learning on the training set.

For both the 66%/33% and 50%/50% mixed datasets, the validation F1 score remains stable around 0.6 around epoch 25, suggesting that model generalization does not improve beyond this point. Interestingly, the validation loss increases throughout training, reaching values around 2.0.

Given the use of weighted cross-entropy, this trend suggests that the model becomes increasingly confident in its predictions, including incorrect ones on underrepresented crack pixels. These misclassifications contribute disproportionately to the validation loss, despite stable segmentation performance as reflected by the F1 score.

5 Discussion and Conclusions

This section begins by assessing the extent to which the research questions outlined in Section 1 have been addressed. It then discusses key observations and limitations encountered during the study, and concludes with suggested directions for future work.

5.1 Research question assessment

Can Unreal Engine produce synthetic crack images that are comparable to or exceed the visual and functional quality of those generated in Blender? Based on the experimental results presented in Section 4, synthetic images generated using the Unreal Engine framework do not yet achieve the same level of effectiveness as those produced in Blender, particularly when combined with real data in mixed datasets. Models trained on Blender-generated data yielded higher F1 scores and better precision across all tested configurations. Additionally, the increasing validation loss observed when training with Unreal Engine data suggests issues with generalization, likely due to overconfident predictions on weighted crack pixels. However, an important finding is that the F1 score for Unreal-based data remains relatively stable even as the proportion of synthetic data increases, suggesting a degree of robustness that merits further exploration. Future improvements to the visual realism and diversity of cracks generated in Unreal may help close this performance gap.

It is important to note that the Blender-based results used for comparison were obtained from previously published experiments and were not reproduced in parallel with the Unreal Engine evaluations. As such, differences in training conditions may introduce inconsistencies that limit the validity of direct, one-to-one comparisons.

Additionally, while the stability of the F1 score observed in the Unreal Engine experiments may initially appear to reflect robustness, it may also be attributed to the limited variability in the generated dataset caused by lack of diverse crack behavior and limited selection of textures and models used. This lack of diversity may result in a plateau in model performance, where additional synthetic data neither significantly improves nor degrades accuracy, due to redundancy in the training signals. Therefore, while the framework shows potential, its capacity to support effective learning at scale is contingent on addressing these variability limitations.

5.2 Limitations

As this study focused on developing a functional proof of concept using relatively new tools and technologies, several components of the framework

remain limited in scope and refinement. The development followed a "minimum viable product" approach, prioritizing core functionality over completeness or optimization. Consequently, the following areas require further development to enhance the framework's performance, realism, and scalability:

- **Crack points selection:** The current method for selecting crack start and end points is deterministic and simplistic, leading to repetitive and predictable crack patterns. The starting point is always positioned in the upper-left quadrant relative to the end point, reducing geometric diversity. A proposed improvement—postponed due to time constraints—involves integrating point selection directly within the crack generation script. This would enable dynamic selection of points based on the render target content, potentially allowing more natural and varied crack trajectories.

- **Crack generating algorithm:** The crack path is generated using a modified A* algorithm, where noise is introduced during pathfinding to avoid overly linear and unnatural paths. While this introduces some variation, the method remains limited in complexity. It currently lacks features such as branching, curvature control, or adaptive trajectory logic. Additionally, the Gaussian blur applied to the path—used to simulate crack depth and width—could benefit from improved control, particularly in how the sigma value evolves along the path. Enhancing both the path generation and post-processing steps would lead to more realistic and diverse crack patterns.

- **Limited Brick Textures:** The framework currently supports only a single brick texture and its associated surface mask, due to limitations in image processing experience. This restricts variability across generated scenes. While random hue shifts, brightness adjustments, and surface decals (e.g., dust or grime) are used to simulate variation, these techniques are not a full substitute for genuinely different surface types. However, the system is designed for extensibility: once additional texture sets and masks are created, they can be integrated easily by instantiating new materials based on the master brick shader.

- **PCG conflicts:** Issues have been observed within the Procedural Content Generation (PCG) system, particularly in the uneven distribution of objects. Certain meshes appear disproportionately often due to the current overlap-avoidance logic, which prevents objects from being placed in close proximity. This logic is implemented per actor and does not consider the global distribution of placements. A more robust solution would involve first identifying all valid spawn locations, then distributing actors across these points using either uniform or weighted sampling strategies to achieve more balanced scene compositions.

- **Occlusion avoidance:** The framework includes logic to prevent occlusion of the crack by PCG-placed objects. However, the exclusion zone around the crack's center is currently a static value defined in the editor, regardless of the crack's actual size. As a result, smaller cracks are disproportionately protected from occlusion, often leaving the surrounding area unnaturally empty. This limits scene realism and reduces the likelihood of generating challenging training samples with partial crack occlusion. An adaptive approach that scales the occlusion buffer based on the crack's length or area would mitigate this issue and promote greater variability in object placement near cracks.

Labeling observation: The original dataset of real crack images includes labels that were manually annotated by human experts. While generally accurate, these annotations are not always precise at the pixel level, as illustrated in Figure 5.1. In contrast, the Unreal Engine framework produces pixel-perfect labels procedurally at runtime, ensuring exact alignment between crack geometry and its corresponding label. This discrepancy in annotation quality introduces a source of inconsistency when real and synthetic data are combined during training. The model is exposed to two different standards of label precision, which may lead to confusion during learning and potentially hinder convergence or generalization. The unreal engine

material designed to show the crack can be modified to allow for errors in labeling that, if modeled correctly, can be similar to human made error.

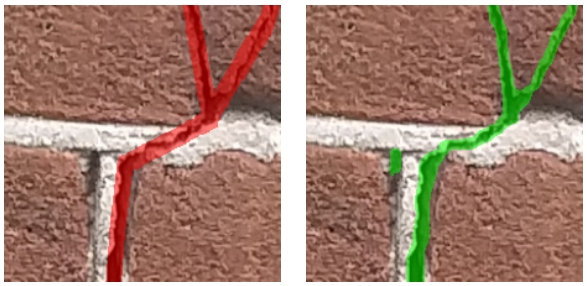

Figure 5.1: The same real crack image with its original label overlayed over it with red (left) and with the model prediction overlayed with green (right)

5.3 Future Direction

As outlined in Subsection 5.2, several components of the framework remain underdeveloped and offer opportunities for further enhancement. In addition to those previously identified limitations, the following directions are proposed to expand the capabilities and applicability of the framework:

- **Increased Scene Diversity:** The Procedural Content Generation (PCG) system can be further leveraged to introduce greater environmental variability. This includes incorporating a wider range of noise objects such as vents, signs, and surface attachments; applying additional decals to simulate surface imperfections like stains or paint erosion; and populating the surrounding environment with larger contextual assets, such as vehicles, streetlights, and vegetation.

- **Advanced Lighting Scenarios:** The framework can be extended to support variable lighting conditions, including different times of day, weather phenomena (e.g., overcast, fog, rain), and artificial light sources such as streetlamps or flash lighting. These additions would help simulate realistic and diverse illumination contexts, thereby improving model robustness to lighting variability.

- **Enhanced Camera Simulation:** Unreal Engine's CineCamera component allows detailed control over parameters such as focal length, aperture, sensor size, and lens distortion. This functionality can be exploited to emulate specific real-world camera and lens configurations, which would improve the realism of synthetic images and better align them with the characteristics of real-world datasets.

5.4 Conclusions

This study explored the development and evaluation of a synthetic dataset generation framework for masonry crack detection using Unreal Engine 5. The framework was designed to replicate key components of a previously established Blender-based pipeline, including crack generation, scene composition, and automated rendering. Despite platform-specific challenges and limitations in available time and domain expertise, a functional proof of concept was achieved.

Experimental results demonstrated that while Unreal Engine-generated images offer a viable alternative to Blender for scenarios requiring large-scale synthetic data generation—particularly when real data is scarce or unavailable—they currently do not match Blender's performance in mixed real-synthetic training configurations. Models trained with Blender data achieved higher F1 scores and precision, particularly in configurations with moderate synthetic input. However, the Unreal-based framework showed greater robustness at higher synthetic data ratios and produced stable F1 scores across training configurations, suggesting its potential for large-scale synthetic dataset generation in low-data settings.

Moreover, the integration of Unreal Engine's Procedural Content Generation (PCG) system significantly improved workflow scalability and scene diversity, reducing the need for manual scene construction. This procedural flexibility, combined with real-time label rendering and camera configuration, positions Unreal Engine as a promising tool for further development in the domain of synthetic image generation for visual inspection tasks.

The framework, while functional, presents several limitations—including simplistic crack path logic, limited surface texture variety, and rigid occlusion avoidance—that impact the realism and di-

versity of the generated data. Nonetheless, these constraints also identify clear directions for future improvement. By incorporating more advanced crack modeling techniques, expanding scene variation, and simulating real-world environmental and camera conditions, the framework could be significantly enhanced in both visual realism and utility.

Overall, this work provides a foundation for further exploration into the use of modern game engines for automated dataset generation, contributing to the broader goal of scalable, reproducible, and realistic training data for computer vision tasks in structural health monitoring.

6 Acknowledgments

The framework project used to generate the full-resolution images, along with the data used to create patches from those images, is available on HuggingFace.

The model architecture used in this study is available on GitHub.

We thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Hábrók high performance computing cluster.

All assets used within this project are listed in Appendix C.

References

Boerema, D. H., Bal, I. E., Smyrou, E., & Kosinka, J. (2025). *Towards Enhancing AI-Based Crack Segmentation for Masonry Surfaces Through 3D Data Set Synthesis.* (Unpublished manuscript)

Dais, D., İhsan Engin Bal, Smyrou, E., & Sarhosis, V. (2021). Automatic crack classification and segmentation on masonry surfaces using convolutional neural networks and transfer learning. *Automation in Construction*, *125*, 103606. Retrieved from https://www.sciencedirect.com/science/article/pii/S0926580521000571 doi: https://doi.org/10.1016/j.autcon.2021.103606

Epic Games. (2025). *Unreal engine features.* Retrieved from https://www.unrealengine.com/en-US/features (Accessed: 2025-06-18)

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. doi: 10.1109/TSSC.1968.300136

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, *abs/1704.04861*. Retrieved from http://arxiv.org/abs/1704.04861

Laefer, D. F., Gannon, J., & Deely, E. (2010). Reliability of crack detection methods for baseline condition assessments. *Journal of Infrastructure Systems*, *16*(2), 129-137. Retrieved from https://ascelibrary.org/doi/abs/10.1061/%28ASCE%291076-0342%282010%2916%3A2%28129%29 doi: 10.1061/(ASCE)1076-0342(2010)16:2(129)

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature pyramid networks for object detection. In *2017 ieee conference on computer vision and pattern recognition (cvpr)* (p. 936-944). doi: 10.1109/CVPR.2017.106

Phares, B. M., Washer, G. A., Rolander, D. D., Graybeal, B. A., & Moore, M. (2004). Routine highway bridge inspection condition documentation accuracy and reliability. *Journal of Bridge Engineering*, *9*(4), 403-413. Retrieved from https://ascelibrary.org/doi/abs/10.1061/%28ASCE%291084-0702%282004%299%3A4%28403%29 doi: 10.1061/(ASCE)1084-0702(2004)9:4(403)

Ros, G., Sellart, L., Materzynska, J., Vazquez, D., & Lopez, A. M. (2016). The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *2016 ieee conference on computer vision and pattern recognition (cvpr)* (p. 3234-3243). doi: 10.1109/CVPR.2016.352

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *2016 ieee conference on computer vision and pattern*

recognition (cvpr) (p. 2818-2826). doi: 10.1109/CVPR.2016.308

VG Insights. (2025). *The big game engines report of 2025.* https://vginsights.com/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf. (Accessed: 2025-06-18)

Zhang, A., Wang, K. C. P., Fei, Y., Liu, Y., Tao, S., Chen, C., ... Li, B. (2018). Deep learning–based fully automated pavement crack detection on 3d asphalt surfaces with an improved cracknet. *Journal of Computing in Civil Engineering*, *32*(5), 04018041. Retrieved from https://ascelibrary.org/doi/abs/10.1061/%28ASCE%29CP.1943-5487.0000775 doi: 10.1061/(ASCE)CP.1943-5487.0000775

A AI usage

This study used Large Language Models for general rephrasing and restructuring of general and rough ideas into academic text. The model used is the free version of GPT-4o. The outputs provided by the model have been closely inspected and corrected if factual errors were observed. Bellow, examples of how the model has been used are provided:

Prompt: Rephrase the following paragraph to fit into an academic report: The developed Unreal Engine (UE) framework is composed of multiple components which interact with each-other. In the following subsections, an overview of each important component developed for the UE framework as well as brief introductions to the tools used for their development will be provided.

Response: The Unreal Engine (UE) framework developed in this study consists of several interconnected components. The following subsections provide an overview of each key component, along with brief descriptions of the tools and technologies employed in their implementation.

Prompt: Rephrase the following paragraph: The original dataset containing real images of cracks also contains human annotated labels. Upon close inspection, despite being accurate, the labels are not always perfect as shown in Figure5.1. The fact that the unreal engine framework generates pixel perfect labels at runtime, there defiantly exists a discrepancy between the accuracy of the human annotated labels compared to unreal's labels. this can cause the model to struggle while training as it receives different mixed signals.

Response: The original dataset of real crack images includes labels that were manually annotated by human experts. While generally accurate, these annotations are not always precise at the pixel level, as illustrated in Figure 5.1. In contrast, the Unreal Engine framework produces pixel-perfect labels procedurally at runtime, ensuring exact alignment between crack geometry and its corresponding label. This discrepancy in annotation quality introduces a source of inconsistency when real and synthetic data are combined during training. The model is exposed to two different standards of label precision, which may lead to confusion during learning and potentially hinder convergence or generalization. This highlights the importance of accounting for label noise and structural alignment when mixing datasets from different sources.

B Unreal engine workflow and implementation details

The development of the framework relies on several core Unreal Engine tools, which are briefly introduced bellow to contextualize their application within this work.

B.1 Brief introduction to Unreal Engine tools

Blueprints Blueprints are Unreal Engine's visual scripting system, enabling the creation of object-oriented logic through a node-based interface. Functionally equivalent to traditional scripting languages, Blueprints offer full access to engine-level operations, allowing non-programmers to design complex behaviors. In this framework, Blueprints were used to implement dynamic scene logic, camera control, and actor coordination, forming the backbone of all major components.

Materials Materials define the visual and physical surface properties of objects in the scene. Using a node-based system, they describe interactions with light such as color, roughness, and displacement. In this project, Materials are employed not only for surface appearance but also for simulating geometry deformation through Nanite-compatible displacement mapping, enabling cracks to appear as physically embedded features rather than texture overlays.

Procedural Content Generator The PCG framework supports rule-based generation of scene elements such as assets, geometry, or entire biomes. Within this framework, PCG is used to procedurally populate scenes with occluding or contextual meshes—such as debris, clutter, or secondary structures—to enhance variability in the synthetic dataset. This ensures that crack detection models

are exposed to a wide range of visual scenarios and potential obstructions.

C Meshes and texture sources

The following is a list of the sources of models and textures used within this study:

- Hinged door:https://www.fab.com/listings/630481aa-f86c-439b-9aca-3e6dcfbdee2f
- Gray wooden door: https://www.fab.com/listings/e056d79e-e98b-4226-a953-47af2271057a
- Window: https://sketchfab.com/3d-models/window-2811eb28fa7a49b4adb86b948f75ff37#download
- Brick facade: https://www.fab.com/listings/efa35b96-cb00-48b1-9402-fd35ede9dcf6
- Fake interior cubemap: https://www.fab.com/listings/62e0fe0f-3fd7-4d40-993a-cae13e8199f4
- Diverse foliage: https://www.fab.com/listings/11cc2abb-126c-4452-9fe4-6f2381d96544
- Misc noise items: https://www.fab.com/listings/2e5835c8-6e4b-4cae-aab9-85396e68401c
- Mud decal texture: https://www.fab.com/listings/0a461105-d696-4e2a-834d-a271f9ce67f9
- Mud decal texture:https://www.fab.com/listings/2719f132-39b3-4156-a1d5-953c64ea048c
- Mud decal texture: https://www.fab.com/listings/243aaaab-02e3-4424-a74c-c4f54d689d19
- Mud decal texture: https://www.fab.com/listings/d943a805-9e06-435c-9691-06890fc8cb12