A Novel Approach For Defect Detection Of Wind Turbine Blade Using Virtual Reality And Deep Learning

Md Fazle Rabbi
University of Texas at El Paso

A NOVEL APPROACH FOR DEFECT DETECTION OF WIND TURBINE BLADE USING VIRTUAL REALITY AND DEEP LEARNING

Md Fazle Rabbi

Master's Program in Industrial Engineering

Dedication

To my parents, teachers, and friends

Who always helps me in every ways possible.

They are the reason I am the way I am today.

A NOVEL APPROACH FOR DEFECT DETECTION OF WIND TURBINE BLADE USING VIRTUAL REALITY AND DEEP LEARNING

by

Md Fazle Rabbi

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso in

Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Industrial, Manufacturing and Systems Engineering Department

THE UNIVERSITY OF TEXAS AT EL PASO

August 2023

Acknowledgement

I would like to express my sincere gratitude to Dr. Tzu-Liang (Bill) Tseng, my thesis advisor for all his guidance, patience, and assistance. His continuous support and enthusiasm have motivated me to write this thesis. Thank you for giving me the opportunity, especially as an international student, to be involved in research activities. Special thanks as well to Dr. Md Fashiar Rahman, and Dr. Yirong Lin, my committee members, for all your suggestions and support.

Abstract

Wind turbines are subjected to continuous rotational stresses and unusual external forces such as storms, lightning, strikes by flying objects, etc., which may cause defects in turbine blades. Hence, it requires a periodical inspection to ensure proper functionality and avoid catastrophic failure. The task of inspection is challenging due to the remote location and inconvenient reachability by human inspection. Researchers used images with cropped defects from the wind turbine in the literature. They neglected possible background biases, which may hinder real-time and autonomous defect detection using aerial vehicles such as drones or others. To overcome such challenges, in this paper, we experiment with defect detection accuracy by having the defects with the background using a two-step deep-learning methodology. In the first step, we develop virtual models of wind turbines to synthesize the near-reality images for four types of common defects - cracks, leading edge erosion, bending, and light striking damage. The Unity® perception package is used to generate wind turbine blade defects images with variations in background, randomness, camera angle, and light effects. In the second step, a customized U-Net architecture is trained to classify and segment the defect in turbine blades. The outcomes of U-Net architecture have been thoroughly tested and compared with 5-fold validation datasets. The proposed methodology provides reasonable defect detection accuracy, making it suitable for autonomous and remote inspection through aerial vehicles.

Keywords: Defect detection, virtual reality, deep learning, U-Net, segmentation

Table of Contents

List of Figures

Chapter 1: Introduction

Wind Turbine plays a vital role in the energy generation of the United States of America. Wind turbines provided almost 8.4 % of total generated electricity in 2020 in the United States (According to US Energy Information Administration). According to the U.S Wind Turbine Database (2022), more than 700,800 wind turbines are running across 44 states resulting in the fourth number of sources for electric energy generation. Most importantly, it is the number one renewable energy source for the USA in power generation. For that reason, if the US wants to be a country of Green Energy, Wind Turbine will be their priority. As the European countries are now more focused on renewable energy for global warming and the US is lagging in this sector, the US government is now taking initiatives to grow their renewable energy source like wind turbines. Even recently, some states have set the target to maintain a percentage of energy that should come from renewable energy. So, to make the US a sustainable and low carbon emission country, wind turbine can contribute a lot.

1.1 Background

Wind Turbines are generally installed in remote and rural areas where there is a constant and strong wind. The constant and strong wind can be found in both onshore and offshore locations. Hills, open fields, or coastal areas can be a very good source of wind power for onshore settlements. On the other hand, oceans or lakes are the prime sources of wind for onshore settlement. Because of wind speed and the visual impact of nature, the onshore wind turbine is preferable for installation. But both locations are remote for proper inspections of the wind turbine. After installing the wind turbine, the blade defects generally start over time. Though blade defects can occur anytime, environmental factors like lightning, flying objects, and high wind plays an important part in it.

Along with that, poor inspections and maintenance are part of the gradual process of blade defects. For that reason, the detection of wind turbine defects in the initial phase is necessary for uninterrupted power generation.

1.2 Problem Statement

Wind turbines show different types of defects in its blade during their lifetime. But for generalizations and understanding, in this paper, four main types of defects have been studied such as 1) Blade Crack, 2) Leading Edge Erosion, 3) Delamination, and 4) Lighting Strike Damage. Blade cracks are a serious issue for wind turbines. If the crack is not identified in the initial phases of its origin, the blade will eventually fail to operate. So, the identification of Blade Crack in the primary phase is a must. In this case, the proper inspection of the blade by drone or watch tower is a measure towards safe operations of wind turbines. On the contrary, leading edge Erosion is another most common defect in wind turbines as the flowing air and dust continuously cause erosions in the edges. Over time, leading edge erosion reduces the efficiency of power generation. Similarly, the delamination of wind turbines is caused by extreme weather, humidity, or mechanical stresses. This is a significant concern for the power company as it has long downtime and safety issues. The manufacturing process and materials selections play a key factor in overcoming delamination defects. Ultrasonic testing and using adhesives are part of maintenance for delamination. The lighting strike damage in wind turbines results due to the lightning effect. It may damage the generator, control systems, power electronics, or blades. The blade may be affected by direct impact or through shock waves.

1.3 Thesis Objectives

This thesis paper addresses the problem of automated detection of the above-mentioned defects in wind turbine blades, which is a critical issue in the renewable energy industry. Traditionally, detecting blade defects is time-consuming and expensive and may not be effective in identifying all types of defects. To address this problem, the paper proposes a novel approach that combines virtual reality and deep learning techniques to develop an automated and efficient defect detection system for wind turbine blades. The proposed system aims to reduce inspection time and cost while improving the accuracy and effectiveness of defect detection from synthetic images.

Chapter 2: Literature Review

For working on defect detection of wind turbine using virtual reality and deep learning, it is needed to go through the literature about the synthetic images, wind turbine images, and deep learning. In recent years, the integration of VR based images and deep learning techniques has come out as a novel approach to increase the accuracy and efficiency of defect detection processes. Such synthetic images have achieved a significant attention in the field of deep learning due to their ability to generate large-scale, diverse, and labeled datasets. Synthetic image generation techniques help to create realistic virtual environments for wind turbine defect detection. The integration of virtual reality and deep learning gives a new opportunity to create immersive training environments for wind turbine defect detection. Synthetic images can help overcome the limitations of real-world datasets and the remove barrier for limited availability, annotation errors, and variability in environmental conditions.

There are various techniques available to generate synthetic wind turbine images, including 3D modeling, physics-based rendering, and generative adversarial networks (GANs). These approaches help to the create of virtual wind turbine environments with realistic appearance. Whereas, Procedural generation involves algorithms to generate virtual environments, including wind turbine models and surroundings environment, based on needs. This procedure allows for the creation of different customizable wind turbine scenes. Other technique usually used is data-driven synthesis, which uses existing real-world wind turbine images and data to generate new images. Both approach leverages machine learning algorithms to learn patterns and characteristics from the input data and then generates new images that closely visualize the real-world examples. With the help of images of the wind turbine blades, the inspections can be done automatically and quickly by deploying deep learning-based image segmentation and classification techniques [1,2].

Hence, in this work, the U-Net based semantic segmentation technique is used to detect the defects in turbine blades [3]. The U-Net architecture is selected because it can give an accurate and detailed segmentation map from a small image dataset. A multilevel convolutional recurrent neural network (MCRNN) is also good for detecting wind turbine blade icing [4]. MCRNN is a good choice because it combines CNNs and RNNs in a hierarchical manner. In real-time supervisory control [5] and Non-destructive testing [6], An Unmanned Aerial Vehicle (UAV) [7] is always preferable. An UAV takes images for segmentations with the help of the embedded camera. Then those images become a great source of datasets. On the other hand, the blade defect detection's accuracy can be increased by analyzing and identifying the distinctive spectral signature [8]. Again, there may not have enough real-world data available to train a deep learning model. Moreover, getting real-world data, in some cases, are expensive and time costuming. Therefore, it comes to the essence of using synthetic data [9]. As an example of using synthetic data, capturing the relationship among synthetic features [10] and pedestrian data can be used for Deep Convolutional Neural Networks (DCNN) [11]. And in defect detection, Fuzzy Failure Mode and Effect Analysis (FMEA) can also give a structured method for defect analysis [12,13] or any Android application [14]. Infrared thermography (IRT) [15] and continuous line laser thermography for damage visualizations for wind turbines [16] are other methods. However, after extensively evaluating various deep learning architectures, we have chosen U-Net as the most suitable one for this paper.

Chapter 3: Methodology

3.1 Designing the Wind Turbine

3.1.1 Selecting a CAD Software

To generate synthetic images, it is necessary to design a virtual model of wind turbines. The design of wind turbines can be done with the help of different Computer Aided Design (CAD) software like SolidWorks, Maya, or Blender. These software are commonly known for their ability to create 3D models. Among these options, SolidWorks 2021 was specifically selected for to its advanced assembly tools facilitating the assembly process of complex parts in the model. SolidWorks 2021 offers a comprehensive set of features and capabilities for designing various models as well as wind turbines models. Solidworks user-friendly interface and robust tools helps modeling and detailing the features. By utilizing SolidWorks, the wind turbine can be precisely designed in IPS (Inchi, Pound, Second) measurements, maintaining the model's desired specifications.
One of the benefits of using the SolidWorks 2021 is its ability to design complex assembly of models. Wind turbines consisting of numerous interconnected components, SolidWorks' assemble them together using tools. This capability helps simplifing the modeling process which allows fthe creation of complex and real-like wind turbine models. SolidWorks 2021's built in range of features including parametric modeling enables the creation of flexible and adaptable designs. This feature is particularly valuable designing wind turbines and it allows for easy modification of blade length, hub size, and tower height. Such flexibility ensures the generated synthetic images presenting different configurations of wind turbines.

Again, SolidWorks 2021 provides simulation capabilities enabling designers to analyze the structural integrity and performance of the wind turbine model. This feature helps in optimizing the design to identify potential issues and allow for necessary adjustments before physical production. The accuracy and precision provided by SolidWorks 2021 makes it an ideal choice for wind turbine design. Furthermore, the software's advanced rendering capabilities contribute creating visually beautiful and realistic virtual models. These models can be utilized for various purposes including visual simulations and presentations. In conclusion, SolidWorks 2021's selection for wind turbine modeling in IPS measurements is driven by its assembly tools, parametric modeling capabilities, simulation features, and advanced rendering options.

3.1.2 Designing the Wind Turbine in SOLIDWORKS 2021

The base of the wind turbine is constructed exclusively using the boss extrude tool. For facilitating the modeling process, the structural component called mast has been divided into two sections, each of which has created using the revolve tool. Then, the motor housing and rotor were designed using a combination of the boss extrude tool and the shell tool. The distance between the base and the center of the rotor was set at Z=380 inches. After that, surface modeling techniques have employed to design a single blad and then it has replicated using the circular pattern tool to create two additional blades. These blades have been evenly distributed across a 360-degree span, with the rotor serving as the central point. The resulting circle created by the blade measures 200 inches in diameter.

To visualize various possibilities of delamination in the blades, the flex tools has bent and twisted the blades in ten different ways. Additionally, the extrude cut function has deployed to simulate lightning strike damage and blade cracks in twenty different scenarios. Another defect type, leading edge erosion has been created by using surfacing tools to remove specific sections of the blades, with ten different erosion patterns created. In short, the wind turbine design created in a a step-by-step process using various modeling tools such as boss extrude, revolve, shell, circular pattern, surface modeling, flex, and extrude cut. These techniques allow for the creation of a base, mast, motor housing, rotor, and blades, as well as the simulation of delamination, lightning strike damage, blade cracks, and leading edge erosion in different configurations.

Finally, Keyshot 10 has been utilized to incorporate dirt, rust materials, and corrosion textures, which enhances the visual realism of the models. Through the advanced capabilities of Keyshot 10, all the models have been seamlessly exported into the FBX format. This format ensures quality and facility of their seamless integration into Unity, a widely recognized cross-platform game engine known for creating immersive virtual environments. Unity's capabilities makes it an ultimate choice for further simulations and interactive experiences. By using Keyshot 10 and Unity in conjunction, the models' aesthetics and functionality can be synergistically increased, providing users with an engaging and visually appealing virtual environment. The integration of dirt, rust, and corrosion textures adds an authentic touch, lending a sense of realism and depth to the virtual scenes. The FBX format's versatility facilitate transition between Keyshot 10 and Unity, streamlining the workflow for developers and artists alike. The combination of these cutting-edge tools gives designers confident to deliver high-quality virtual experiences and captivate audiences across multiple platforms.

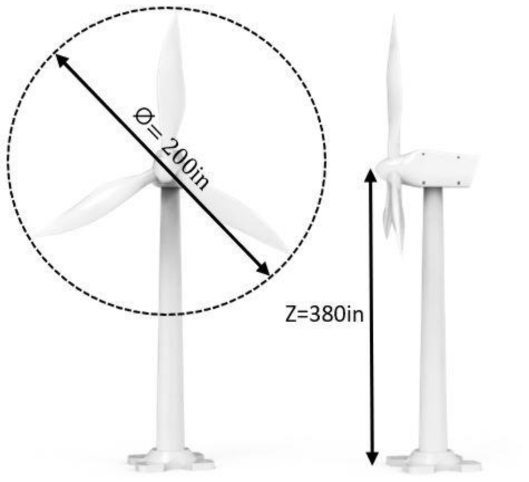


Figure 1 Designing the Wind Turbine

3.1.3 Deciding the Amount of Wind Turbine Needed to Generate

Determining the optimal number of wind turbine models is a critical task as it directly impacts the assessment of environmental randomness. A greater number of models, each with variations, introduces a higher degree of randomness in the testing environment. This heightened level of randomness serves to thoroughly challenge and evaluate the deep learning model's robustness. However, generating a substantial number of models tends to be time-consuming and complex, particularly due to the requirement for distinct defect shapes in each model. After careful consideration, it has been decided to proceed with a total of 40 different wind turbine models, with 10 models allocated to each of the four blade defect types. This approach aims to strike a balance between generating a diverse set of models and managing the practical constraints of the task. Each of the 40 models will exhibit unique characteristics pertaining to their respective defect type, ensuring a comprehensive representation of potential scenarios.

By implementing this strategy, the testing process can visualize a broad range of defect shapes, providing valuable insights into the deep learning model's performance across various realistic scenarios. Although the generation of 40 distinct models presents difficulties, the resulting dataset will serve as a valuable resource for training and validating the deep learning model, ultimately contributing to more reliable and efficient wind turbine operations.

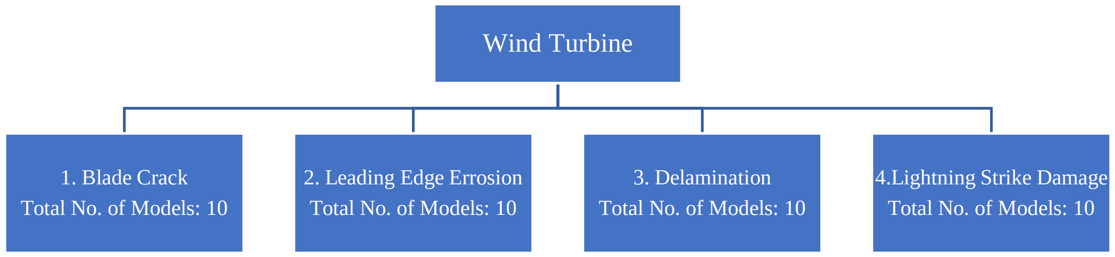


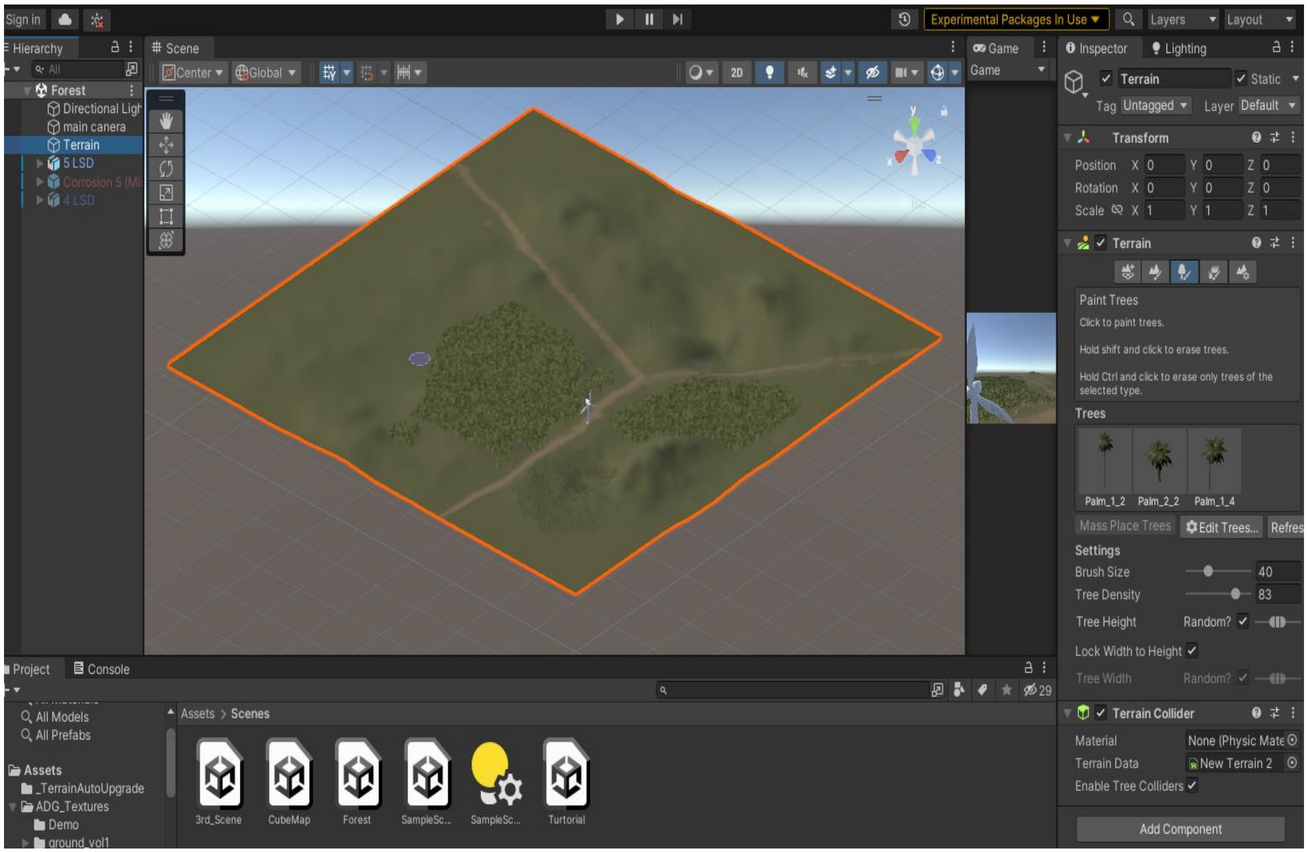
Figure 2 Schematic Overview of Wind Turbine Defect Type And Generated Model Amount

3.2 Creating the Virtual Environment in Unity

In this paper, the virtual world has been made inside the Unity in two different scenes: a. Terrain, b. Cube Map.

3.2.1 Creating the Virtual Environment in Unity Terrain

The Unity terrain system facilitate designers a wide range of options to create diverse rural environments. By working on the vertices of the terrain, developers can shape hills, valleys, and lowlands, giving the terrain a natural and realistic appearance. Moreover, in the heightmap option, it allows for the addition of different surface textures, further enhancing the visual appeal of the terrain. In this particular case, the terrain has been textured with painted green grass which gives it a lush and vibrant look. To add even more variety, three types of ground textures have been incorporated, creating a more immersive and believable environment. The inclusion of a downloaded tree asset from the Unity asset store adds an extra layer of realistic looks, as the trees are seamlessly integrated into the terrain. Furthermore, a rural road has been crafted using various brushes, adding a sense of ruggedness and practicality to the environment. Finally, to complete the scene, a defective wind turbine has been imported, adding an intriguing element and capturing attention.



Terrain ✓ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Terrain
Paint Trees
Click to paint trees.
Hold shift and click to erase trees.
Hold Ctrl and click to erase only trees of the selected type.
Trees
Palm_1_2 Palm_2_2 Palm_1_4
Mass Place Trees Edit Trees... Refres
Settings
Brush Size 40
Tree Density 83
Tree Height Random? ✓
Lock Width to Height ✓
Tree Width Random? ✓

Terrain Collider
Material None (Physic Mate
Terrain Data New Terrain 2
Enable Tree Colliders ✓

Figure 3 Setting up Unity Terrain

The Unity terrain system gives the developers a powerful tool for creating the appealing environment for their games or simulations. Through careful assuages of vertexes, texture painting, and asset integration, developers can visualize the stunning and immersive landscapes. The combination of painted grass, ground textures, tree assets, and a muddy road in this specific terrain adds depth and authenticity to the scene. The addition of the defective wind gives us the environment we need to capture images realistic look-like. For simulation image capturing purposes, the Unity terrain system provides an accessible and effective means of creating captivating environments.

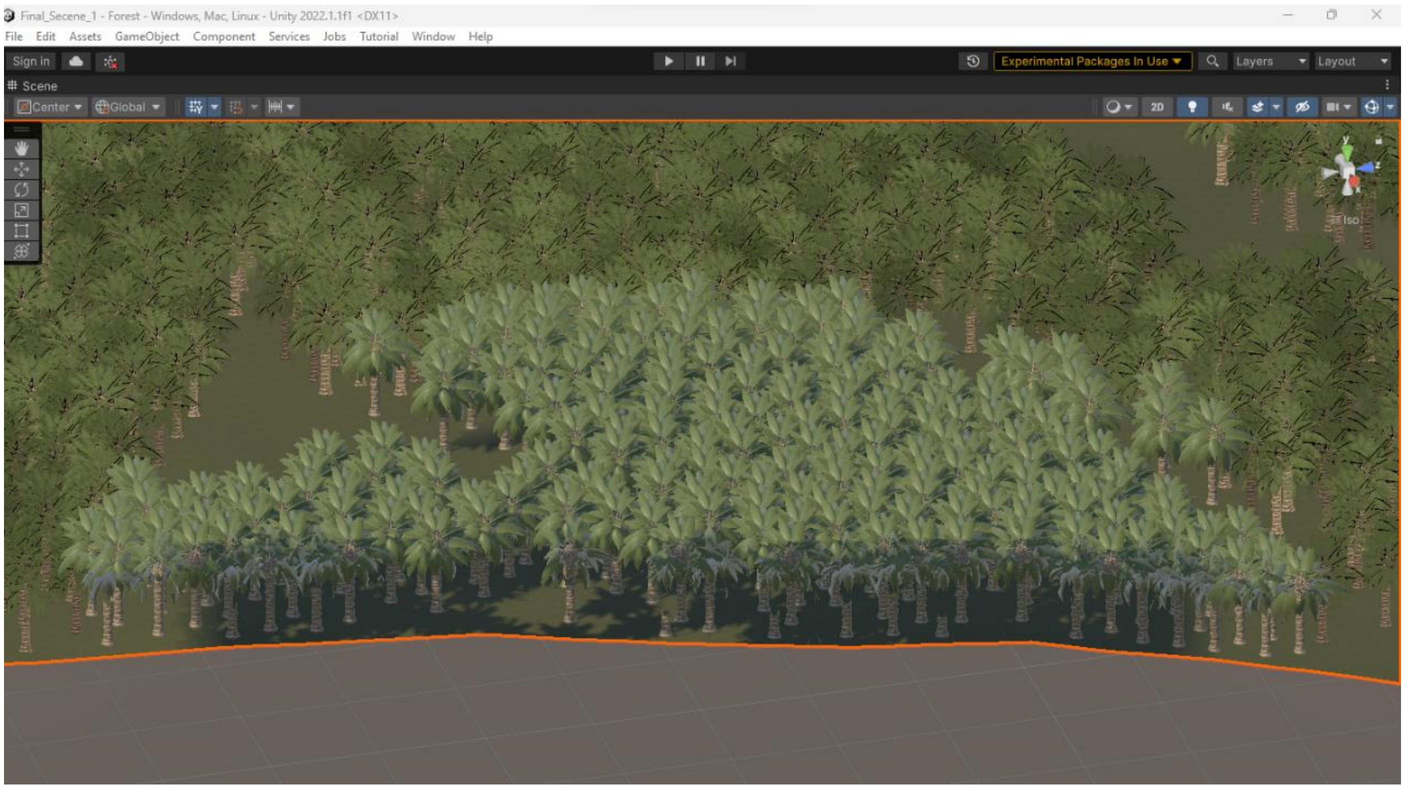
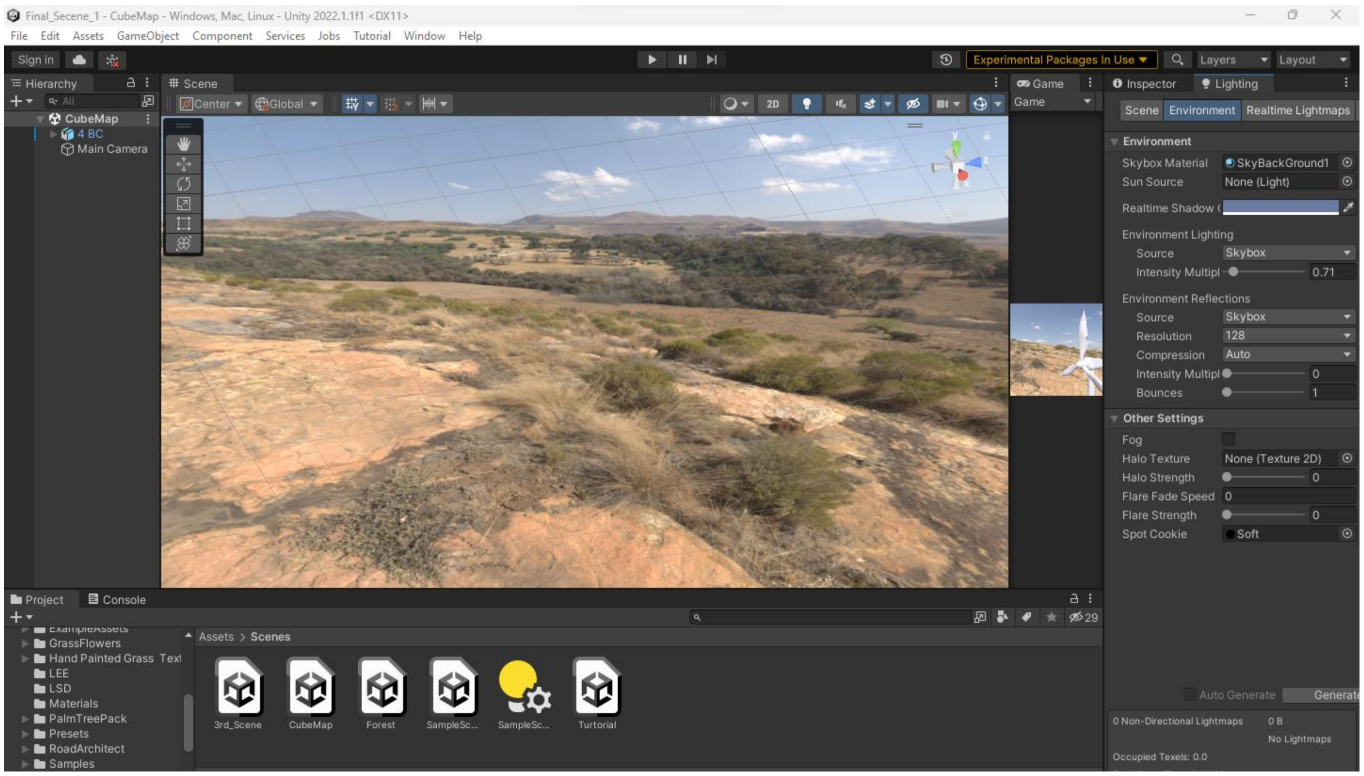

Final_Secene_1 - Forest - Windows, Mac, Linux - Unity 2022.1.1f1 <DX11>
File Edit Assets GameObject Component Services Jobs Tutorial Window Help
Sign in
Experimental Packages In Use
Layers
Layout
Scene
Center
Global
2D
Iso

Figure 4 Inserting Trees in Terrain

3.2.2 Creating the Virtual Environment in Unity Cube Map

The second virtual scene used is Unity Cube Map technology, which is commonly found in 3D graphics and 360-degree environments. This mapping technique allows for the creation of three-dimensional spaces by combining multiple images of the top, bottom, front, back, left, and right perspectives. Where you can see the 360 degree by rotating the images. With Unity Cube Map, the creates the 3D geometry seamlessly integrates into the overall environment. To incorporate the cube map into the project, users can easily import the free version available from the Unity Asset Store. There are numerous cube map images are available in the Unity Asset Store. You can buy the cube images or even can get it free. In our case, it has been downloaded free from there. After that, the map has been uploaded in the unity projects. Then a material has been created in the asset store. Because for applying the cube map, you need to create material first.

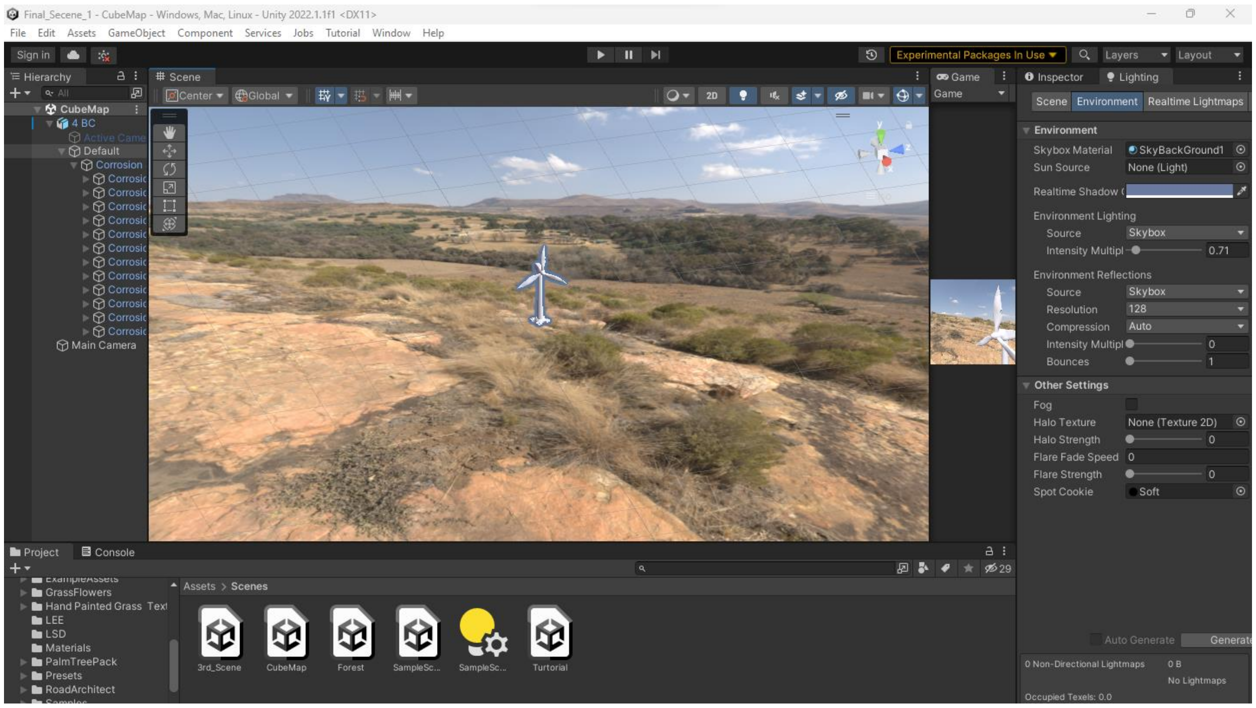


Environment
Skybox Material SkyBackGround1
Sun Source None (Light)
Realtime Shadow C
Environment Lighting
Source Skybox
Intensity Multipl 0.71
Environment Reflections
Source Skybox
Resolution 128
Compression Auto
Intensity Multipl 0
Bounces 1
Other Settings
Fog
Halo Texture None (Texture 2D)
Halo Strength 0
Flare Fade Speed 0
Flare Strength 0
Spot Cookie Soft

Figure 5 Setting Up Unity Cube Map

Then the downloaded cube map has been applied to the material created. In our case, there are several types of cube map in the asset store with different background environment. But we select the cube map with semi desert hilly area because wind turbines are generally installed area like that. Once imported, the cube map is applied as a material within the scene, providing a visually stunning backdrop against which the defective model can be placed.

In this particular scenario, the defective model was added into the scene to be observed and captured images from various angles. The cube map's contribution shows apparent as it envelops the model, creating a lifelike setting that accurately reflects its surroundings. By leveraging Unity's Cube Map technology, designers can obtain a high level of fidelity and detail in their virtual environments, enhancing the overall visual experience. The usages of the cube map and the defective model provides engaging visual representation and users can examine and analyze the model's intricacies from different perspectives and lighting conditions. Most importantly, the perception packages can work in such in environment.


Final_Secene_1 - CubeMap - Windows, Mac, Linux - Unity 2022.1.1f1 <DX11>
File Edit Assets GameObject Component Services Jobs Tutorial Window Help
Experimental Packages In Use
Hierarchy
Scene
Game
Inspector
Lighting
Project
Console
3rd_Scene
CubeMap
Forest
Turtorial

Scene | Environment | Realtime Lightmaps

Environment
Skybox Material SkyBackGround1
Sun Source None (Light)
Realtime Shadow (
Environment Lighting
Source Skybox
Intensity Multipl 0.71
Environment Reflections
Source Skybox
Resolution 128
Compression Auto
Intensity Multipl 0
Bounces 1
Other Settings
Fog
Halo Texture None (Texture 2D)
Halo Strength 0
Flare Fade Speed 0
Flare Strength 0
Spot Cookie Soft

Figure 6 Inserting Wind Turbine in Unity Cube Map

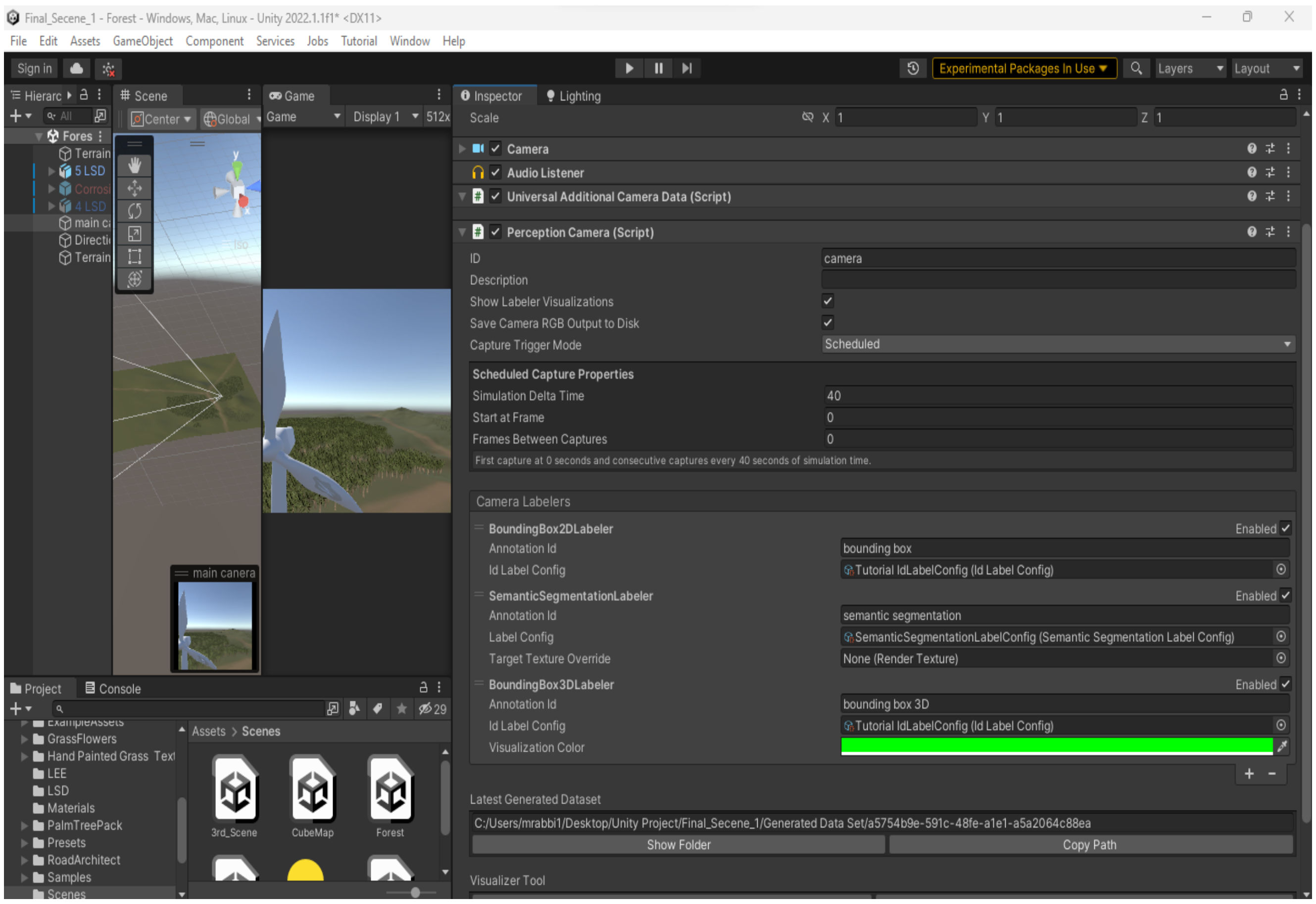
3.3 Generating Synthetic Images Using Unity Perception Package

The Unity Perception Package is a useful tool for capturing images from a Unity environment. The perception package is still in experimental stage. But the package can be downloaded from Unity Package Manager. To use it, you can download the package from there and it would be stored in the Unity project. The perception package has five components and it can be seen in the project windows: forward renderers, sample scene, universal RP – high quality, universal RP – low quality, and universal RP – medium quality.

Before applying the perception packages, it is needed to select forward renderer. There it is needed to add ground truth renderer feature. Renderer pipeline is the feature to generate images from all the world in the Unity. It also allows to generate the segmentations data. Firstly, we need to create a empty scene. There we need to rename the scene too. And in the scene window, we will create the scene with the unity terrain and unity cube map. And there needs to upload the defective wind turbine too. Then we need to select the camera from the hierarchy window.

In the project window, the wind turbine model has different parts combination. All the parts of it like turbine, blade, engines, are made differently and then combine to make it. The defect part is also different, and the defect part has been given add label config. After that, the main camera has been set in the direction of the wind turbine. The main camera needs to set the perception package setting in a set by set way. Firstly, the perception camera is added with the main camera of the scene. Then, it is needed to make sure the camera is in a scheduled mode. That means the images will be captured automatically from the camera. There is a simulation delta time in the setting. This is actually simulations time; the camera will capture the images in a rate. By changing the simulation delta time you can change the time needed to capture the images. In our case, we used the simulation delta time 40.

The resolution of the images can also be controlled from the game window free aspect setting. It has been used here 512*512 binary images. So that means, all the images generate here will be 512*512 binary images. Then it is needed to include bounding box 2D labeler in the camera. Then in the asset package, it is needed to find out the tutorial id label config, then needed to set the defect label name in the id label config. Similarly, it is needed to add semantic segmentation labeler in the asset and set the label from defect part. And finally set the semantic segmentation labeler in the main camera. The other labeler name is bounding box 3D labeler.



Scale X 1 Y 1 Z 1

Camera

Audio Listener

Universal Additional Camera Data (Script)

Perception Camera (Script)

ID	camera
Description	
Show Labeler Visualizations	✓
Save Camera RGB Output to Disk	✓
Capture Trigger Mode	Scheduled

Scheduled Capture Properties

Simulation Delta Time	40
Start at Frame	0
Frames Between Captures	0

First capture at 0 seconds and consecutive captures every 40 seconds of simulation time.

Camera Labelers

BoundingBox2DLabeler — Enabled ✓

Annotation Id	bounding box
Id Label Config	Tutorial IdLabelConfig (Id Label Config)

SemanticSegmentationLabeler — Enabled ✓

Annotation Id	semantic segmentation
Label Config	SemanticSegmentationLabelConfig (Semantic Segmentation Label Config)
Target Texture Override	None (Render Texture)

BoundingBox3DLabeler — Enabled ✓

Annotation Id	bounding box 3D
Id Label Config	Tutorial IdLabelConfig (Id Label Config)
Visualization Color	🟩

Latest Generated Dataset

C:/Users/mrabbi1/Desktop/Unity Project/Final_Secene_1/Generated Data Set/a5754b9e-591c-48fe-a1e1-a5a2064c88ea

Show Folder — Copy Path

Visualizer Tool

Figure 7 Adding Perception Package in Main Camera in Unity

After that, it is needed to select the folder to store the all the images. As it creates huge amount of images in a single click, it needs a big storage to keep it. Then, the wind turbine needs to focus on the game window to see how images will be captured from the scene. After refocusing the wind turbine, it is needed to hit the play button of the scene. In this set up the perception camera will capture the same images according of the simulation delta time. But the problem is that the images will not be identical. It is needed to have variation images. For creating variations, the images are moved in different directions to have variations. So, when the play button is played the simulations is started, and the whole wind turbine model is then moved along different direction to have different images. From the different images and their perspective semantic segmentation can be found like the following images.

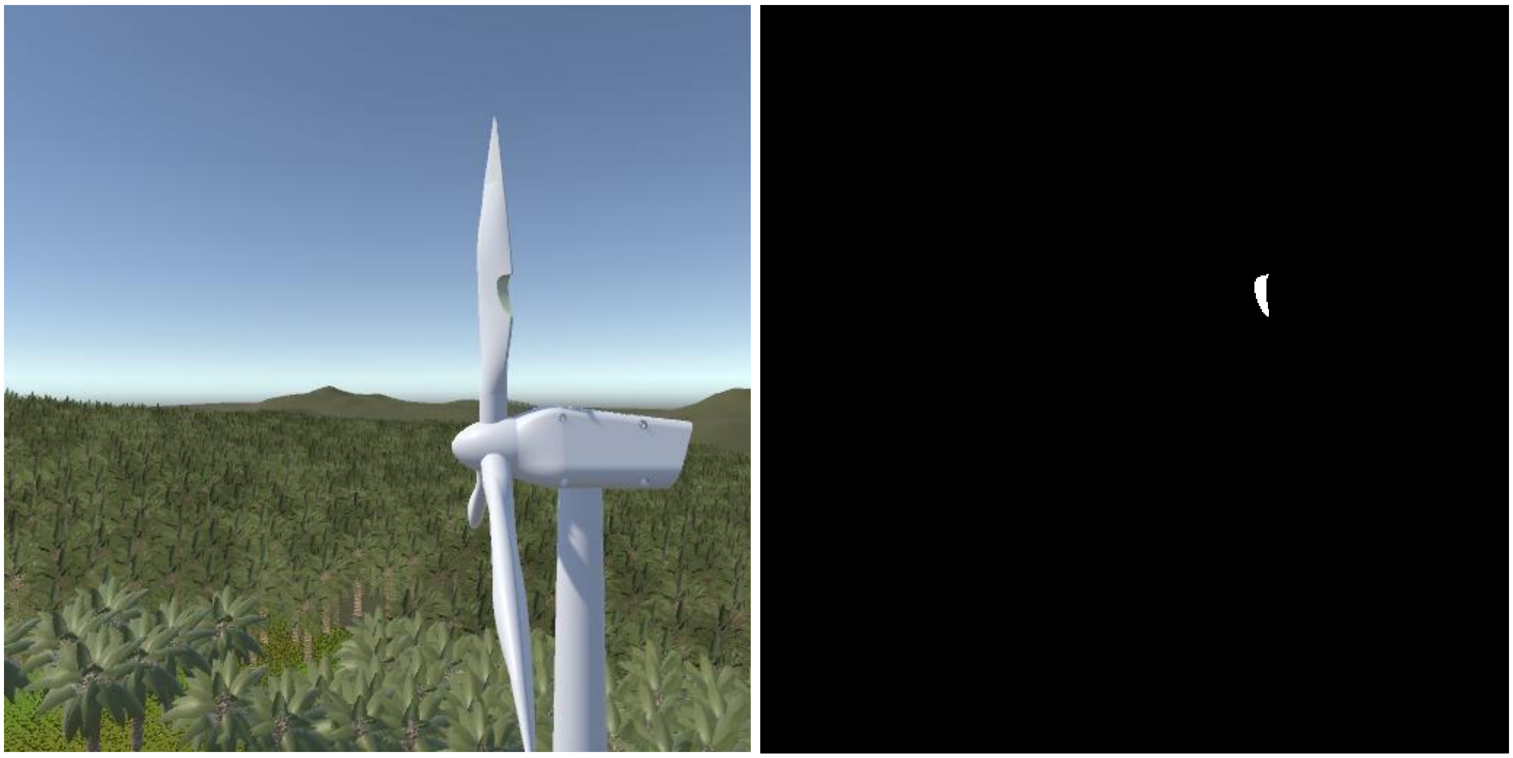


Figure 8 Blade Crack Image and Its Perspective Semantic Segmentation

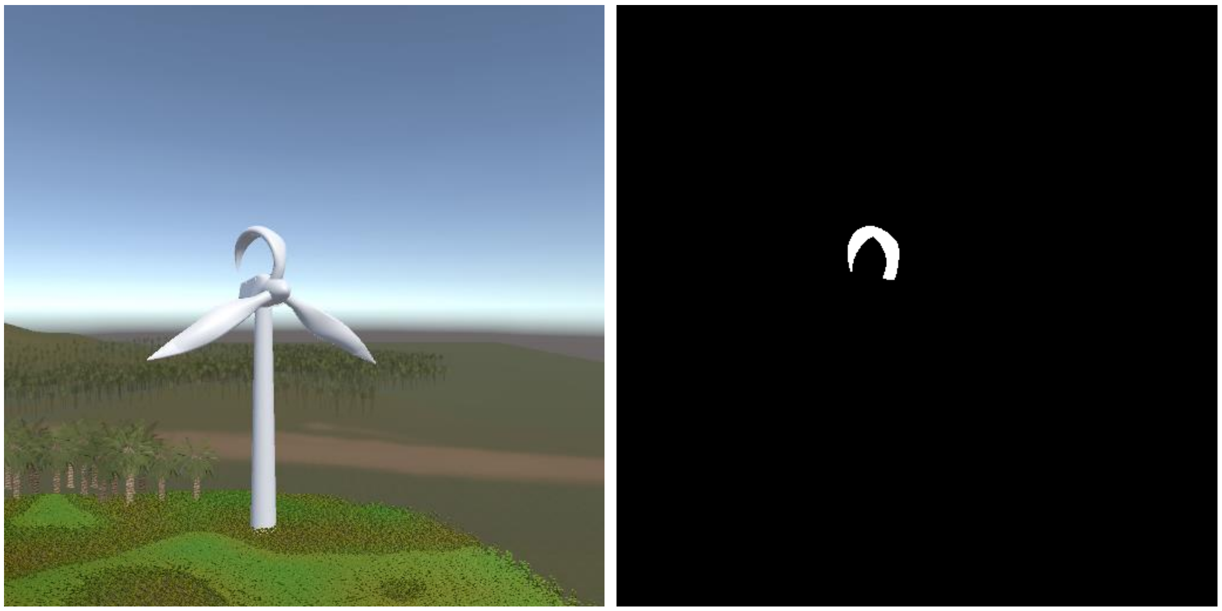


Figure 9 Delamination Image and Its Perspective Semantic Segmentation

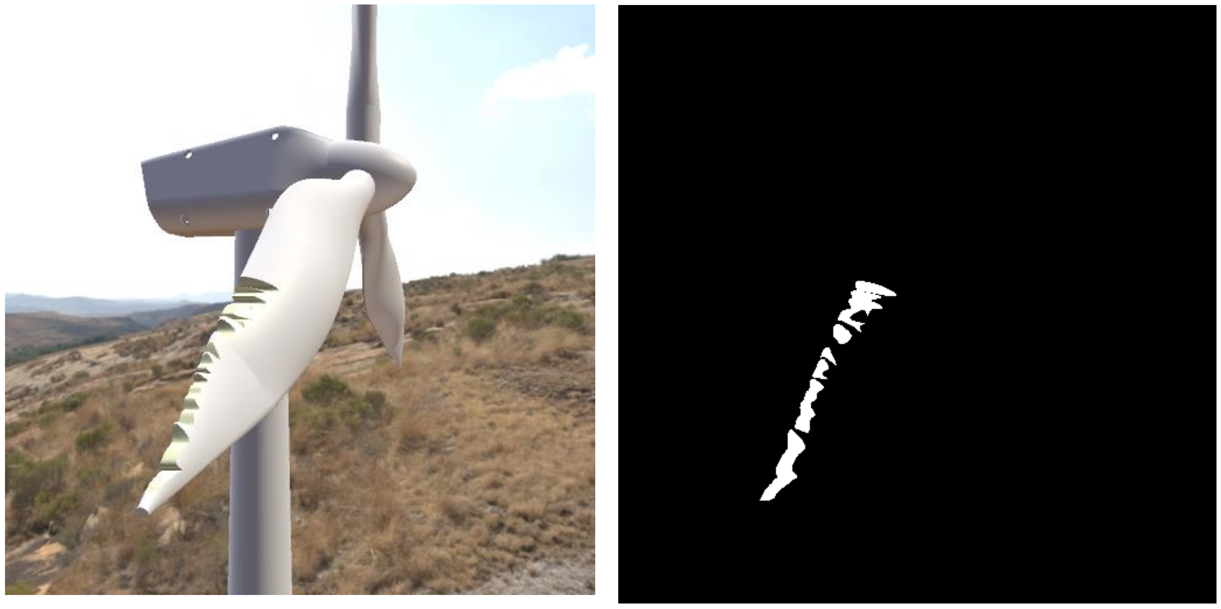


Figure 10 Leading Edge Erosion Image and Its Perspective Semantic Segmentation

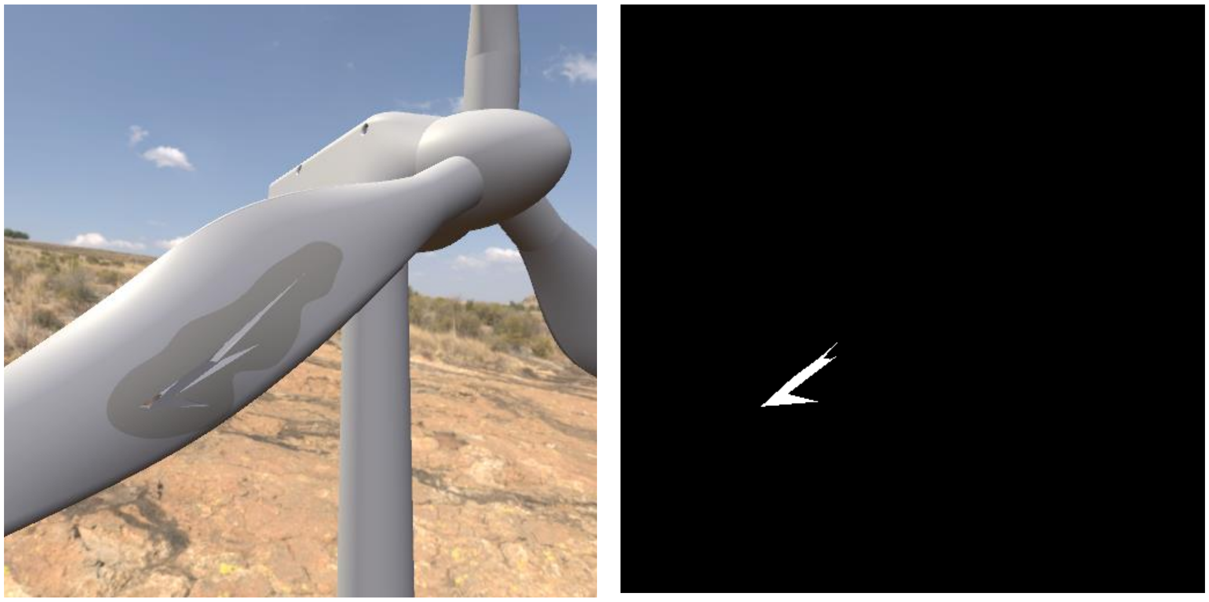


Figure 11 Lightning Strike Damage Image and Its Perspective Semantic Segmentation

This process was repeated for all 40 models, that results in a total of 642 images. Additionally, all images have corresponding semantic segmentation/ground truth in the desired folders. Even, the images have its corresponding text data file too which in dot json format. The text file can also be read through Sublime Text software. Overall, this approach provides a way to capture images from a Unity scene and generate corresponding semantic segmentation/ground truth labels for further analysis.

3.4 Setting Up Basic U-Net Network

After generating the image datasets from Unity, it is needed to select a better deep learning model to detect the defect. Deep learning models like U-Net, faster R-CNN, SSD (Single Shot MultiBox Detector), DeepLab, Mask R-CNN are available to use for the synthetic dataset generated. But we used the basic U-Net architecture [17] for image segmentation. Because U-Net has higher level of accuracy. The network consists of encoding and decoding parts that are connected by the bottleneck. The convolutional layers create the contracting paths. One or more convolutional layers make up the bottleneck, which joins the contracting and expanding paths. Its goal is to portray the most significant aspects of the input image in a compressed manner. Here, the input images are 512*512 Pixels in size. Figure 5 shows the architecture of the U-Net which is employed for our purposes. In the experimental settings, Google Colaboratory is utilized for model training and testing.

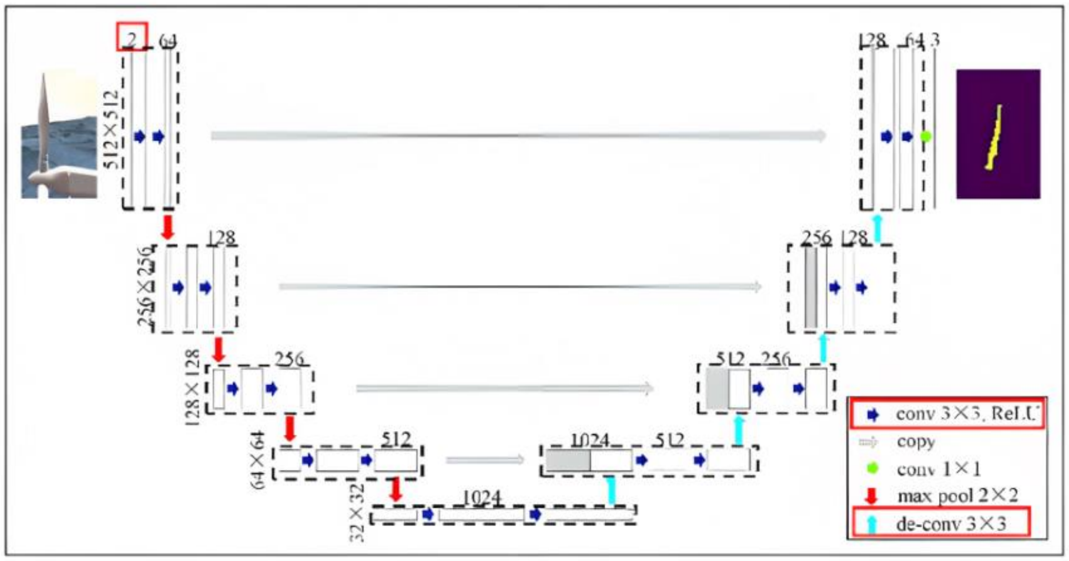


Figure 12 Illustration of U-Net Convolutional Network Structure

To evaluate the model outcomes, Jaccard Index is generally used to measure the similarity between predicted masks and ground truth masks. It measures the overlap between the ground truth and segmentation outcomes.

$$J(W_1, W_2) = \frac{|W_1 \cap W_2|}{|W_1 \cup W_2|} \tag{1}$$

Where, W_1 and W_2 are the ground truth and segmentation outcome, respectively. The Jaccard Index ranges from 0 to 1, with 0 denoting no overlap and 1 indicating complete overlap between the sets. In the basic cross entropy loss, per-pixel loss is calculated discretely, measuring as the average of per-pixel loss. It takes into account loss in a micro sense as opposed to taking it into account globally. For handling this, we have used Dice Loss which originates from the Dice Coefficient (DSC). Jaccard Loss is calculated from the coefficient value by applying $1 - Dice\ Coefficient\ (DSC)$.

$$Dice\ Coeffieient\ (DSC) = \frac{2 \times |A \cap B|}{|A| + |B|} \tag{2}$$

$$Jaccard\ Loss = 1\ - Dice\ Coefficient\ (DSC) \tag{3}$$

Where, A represents the predicted mask set and B represents the ground truth mask set.

The coding for the dataset can be visualized as follows (taken from Google Colab)

```
from google.colab import drive

drive.mount('/content/drive')

%cd /content/drive/My Drive/Unet-Rabbi


#import all the libraries

import numpy as np

from keras.preprocessing.image import ImageDataGenerator
```

```
import os
import glob
import skimage.io as io
from PIL import Image
from skimage.viewer import ImageViewer
import matplotlib.pyplot as plt
def trainGenerator(image_path,mask_path,flag_multi_class = False,num_class = 2,image_prefix
= "images",mask_prefix = "masks",image_as_gray = True,mask_as_gray = True):
    image_name_arr = glob.glob(os.path.join(image_path,"*.PNG"))
    mask_name_arr = glob.glob(os.path.join(mask_path, "*.PNG"))
    image_arr = []
    mask_arr = []
    for index,item in enumerate(image_name_arr):
        img = io.imread(item,as_gray = image_as_gray)
        img = np.reshape(img,img.shape + (1,)) if image_as_gray else img
        img = img / 255
        image_arr.append(img)

    for index, item in enumerate(mask_name_arr):
        mask = io.imread(item, as_gray = image_as_gray)
        mask = np.reshape(mask,mask.shape + (1,)) if mask_as_gray else mask
        mask = mask/255
        mask[mask>0.5] = 1
        mask[mask<=0.5] = 0
        mask_arr.append(mask)
    image_arr = np.array(image_arr)
    mask_arr = np.array(mask_arr)
    return image_arr,mask_arr
def testGenerator(test_path, image_num):
    for i in range(image_num):
        #idx = str(i+69).zfill(5) + ".png"
        idx = 'rgb_'+ str(i+70) + ".png"
        img = io.imread(os.path.join(test_path, idx), as_gray=True)
        img = img / 255
        img = np.reshape(img, img.shape + (1,))
        img = np.reshape(img,(1,)+img.shape)
        yield img
def testMask(test_path, image_num):
    Y_test = np.zeros((image_num, 512, 512, 1), dtype=np.bool)
    for i in range(image_num):
        #idx = str(i+69).zfill(5) + ".png"
        idx = 'segmentation_'+ str(i+70) + ".png"
        msk = io.imread(os.path.join(test_path, idx), as_gray=True)
        msk = msk / 255
        msk = np.reshape(msk, msk.shape + (1,))
        msk = np.reshape(msk,(1,)+msk.shape)
        Y_test[i] = msk
    return Y_test
def getItem(img_path, item):
    idx = str(item+1).zfill(5) + ".png"
    img = io.imread(os.path.join(img_path, idx), as_gray=True)
    img = img/255.0
    img = np.reshape(img, img.shape + (1,))
    img = np.reshape(img,(1,)+img.shape)
    return img
def saveResult(save_path, files):
    for i, item in enumerate(files):
        img = item[:,:,0]
        idx = str(i+1).zfill(5) + ".png"
        io.imsave(os.path.join(save_path, idx), img)
def viewImage(array1, array2, i):
    img = np.reshape(array1[i], (512, 512))
    mask = np.reshape(array2[i], (512, 512))
    f, axarr = plt.subplots(1,2)
    axarr[0].imshow(img, cmap=plt.cm.gray)
    axarr[1].imshow(mask, cmap=plt.cm.gray)
    plt.show()
os.getcwd()
# Load the train data and their corresponding labels
# Visualize a sample image

imgs_train,imgs_mask_train= trainGenerator("train/images/","train/masks/")

print(imgs_train.shape)

print(imgs_mask_train.shape)

index = np.random.randint(1, 50)


#viewImage(imgs_train, imgs_mask_train, index)

!pip install -U segmentation-models

import os

os.environ["SM_FRAMEWORK"] = "tf.keras"

from segmentation_models import Unet, FPN

from segmentation_models import  get_preprocessing # this line has an error in the docs

from segmentation_models.losses import bce_jaccard_loss

from segmentation_models.metrics import iou_score

from segmentation_models.losses import dice_loss


#from segmentation_models.metrics import dice_score



from segmentation_models.utils import set_trainable

from keras import backend as K

def iou_coef(y_true, y_pred, smooth=1):

  intersection = K.sum(K.abs(y_true * y_pred), axis=[1,2,3])

  union = K.sum(y_true,[1,2,3])+K.sum(y_pred,[1,2,3])-intersection
  iou = K.mean((intersection + smooth) / (union + smooth), axis=0)
  return iou
def dice_coef(y_true, y_pred, smooth=1):
  intersection = K.sum(y_true * y_pred, axis=[1,2,3])
  union = K.sum(y_true, axis=[1,2,3]) + K.sum(y_pred, axis=[1,2,3])
  dice = K.mean((2. * intersection + smooth)/(union + smooth), axis=0)
  return dice
from keras import backend as K
def jaccard_loss(y_true, y_pred, smooth=100):
    intersection = K.sum(K.abs(y_true * y_pred), axis=-1)
    sum_ = K.sum(K.abs(y_true) + K.abs(y_pred), axis=-1)
    jac = (intersection + smooth) / (sum_ - intersection + smooth)
    return (1 - jac) * smooth
import numpy as np
import os
import skimage.io as io
import skimage.transform as trans
import numpy as np
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras import backend as keras
```

```
def unet(input_size = (512, 512,1)):

    inputs = Input(input_size)

    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(inputs)

    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv1)

    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(pool1)

    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv2)

    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(pool2)

    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv3)

    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(pool3)

    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv4)

    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(pool4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv5)
    drop5 = Dropout(0.5)(conv5)
    up6 = Conv2D(512, 2, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(UpSampling2D(size = (2,2))(drop5))
    merge6 = concatenate([drop4,up6], axis = 3)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(merge6)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv6)
    up7 = Conv2D(256, 2, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(UpSampling2D(size = (2,2))(conv6))
    merge7 = concatenate([conv3,up7], axis = 3)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(merge7)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv7)
    up8 = Conv2D(128, 2, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(UpSampling2D(size = (2,2))(conv7))
    merge8 = concatenate([conv2,up8], axis = 3)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(merge8)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv8)
    up9 = Conv2D(64, 2, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(UpSampling2D(size = (2,2))(conv8))
    merge9 = concatenate([conv1,up9], axis = 3)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(merge9)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv9)
    conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same', kernel_initializer =
'he_normal')(conv9)
    conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)
    model = Model(inputs, conv10)
    model.compile(optimizer = Adam(lr = 1e-4), loss = [dice_loss], metrics = [iou_score])
    return model
# laod model and show the model summary
model = unet()
model_checkpoint = ModelCheckpoint('windTurbine_segmentation.hdf5',
monitor='loss',verbose=1, save_best_only=True)
model.summary()
# train the model
```

```
hm_epochs = 100

batch_size = 4

model.fit(imgs_train, imgs_mask_train, batch_size=batch_size, epochs=hm_epochs, verbose=1,

        validation_split=0.2, shuffle=True, callbacks=[model_checkpoint])

test_path = 'test/images/'

test_image_arr = glob.glob(os.path.join(test_path, "*.png"))

num_of_test_images = len(test_image_arr)

testGene = testGenerator(test_path, num_of_test_images)

results = model.predict(testGene, num_of_test_images, verbose=1)

# print(results)

print(results[0])

results = results >= 0.1

print(results)

#plt.imshow(results[0], cmap='Reds', alpha=0.3)

plt.imshow(np.squeeze(results[0]), cmap='Reds', alpha=0.3)

test_path = 'test/masks/'

test_image_arr = glob.glob(os.path.join(test_path, "*.png"))

num_of_test_images = len(test_image_arr)

testMsk = testMask(test_path, num_of_test_images)

print(testMsk[0])

plt.imshow(np.squeeze(testMsk[1]), cmap='Reds', alpha=0.3)

plt.imshow(np.squeeze(results[1]), cmap='Reds', alpha=0.3)

idx = 0
```

```
true = testMsk[idx]
predicted = results[idx]
intersection = np.logical_and(predicted, true)
union = np.logical_or(predicted, true)
iou_score = np.sum(intersection) / np.sum(union)
print('IoU is %s' % iou_score)
def Jaccard_img(y_true, y_pred): #https://www.jeremyjordan.me/evaluating-image-
segmentation-models/
    iou_score=0
    counter=0
    for i in range(y_true.shape[0]):
        if np.sum(y_true[i])>0:#Considering only the slices that have hemorrhage regions, if y_true
is all zeros -> iou_score=nan.
            im1 = np.asarray(y_true[i]).astype(np.bool)
            im2 = np.asarray(y_pred[i]).astype(np.bool)
            intersection = np.logical_and(im1, im2)
            union = np.logical_or(im1, im2)
            iou_score+= np.sum(intersection) / np.sum(union)
            counter+=1
    if counter>0:
        return iou_score/counter
    else:
        return np.nan
```

```
jac_indx = Jaccard_img(predicted, true)

print(jac_indx)

print('*_------------------------------------------')
```

The output of the above basic U-Net code has been discussed in discussion and conclusion chapter.

Chapter 4: Results and Discussions

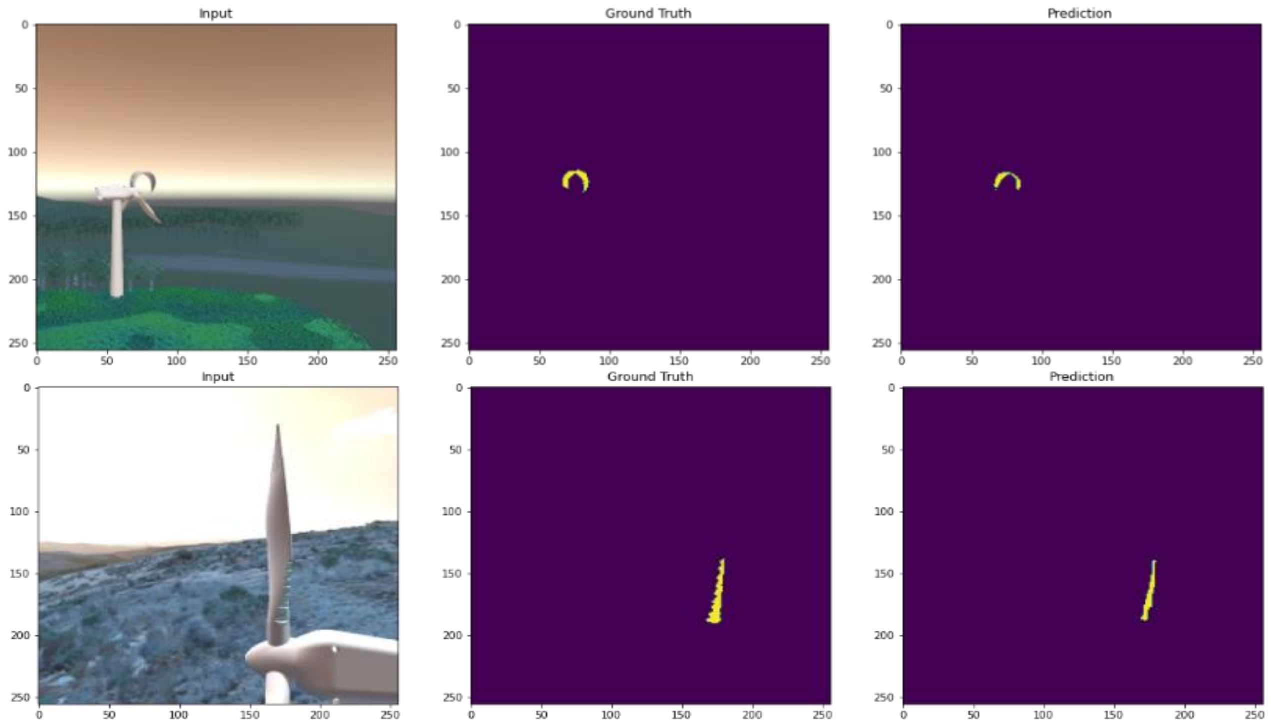


Figure 13 Visualization of Model's Output

Unity generates images automatically based on the setup as discussed above. During the simulation, unity can generate tons of images within a very short period with variations with precise annotation of semantic segmentation. We use the simulated images and their corresponding annotation (ground truth) to train the U-Net model. The Adam optimizer [18] is used to train the network. Adam is a well-known optimization algorithm for deep neural network training which gives adaptive learning rate.

As the Dice coefficient gives the similarity between the predicted segmentation map and the ground truth, Dice coefficient is used here. Then, Jaccard loss has been used as loss function here because Jaccard loss function is complementary to Dice coefficient. They are related but not interchangeable to each other. To calculate the Jaccard loss, one needs to subtract the Dice coefficient from 1. In this synthetic image segmentation in U-Net, the goal is to partition an image into different regions or objects of interest. The Dice coefficient gives a measure of how well the algorithm's segmentation result aligns with the ground truth segmentation. It is particularly useful when dealing with binary segmentation tasks, where each pixel is classified as either foreground (object) or background. So, the dice coefficient selection is perfect here. However, the total number of datasets used here are 642 with four main defect types: blade crack, leading edge erosion, delamination, and lightning strike damage. And each defect has 10 different types of variations positioning the defect in a different area and in size.

The dataset was divided into the train set (75%) and test set (25%) and the model was trained for 100 epochs. The improvement rate in each epoch is moderate while training the model. After training and testing the basic U-Net model, we obtain the Dice coefficient of 0.658. A Dice coefficient of 0.658 suggests that the U-Net model has achieved moderate segmentation accuracy, but there is still scope for improvement. In other words, the model is correctly segmenting 65.8% of the pixels in the image. On the other hand, Dice coefficient of 0.658 in the model means the Jaccard loss of the model is 0.342. The model fails to visualize 34.2% of the ground truth in the predicted mask.

As the classes are more balanced in our experiment, the Dice Loss indicates that the model needs further refinement. Figure 6 also supports the finding of our experiment. As we can see in Figure 6 (both delamination and blade crack defects), the predicted masks from U-Net are close to the Unity-generated ground truth. But notice that the segmented result is still missing some of the pixels compared to the ground truth. Despite the limitation, the proposed methodology can segment the defects with reasonable accuracy. Table 1 provides a summary of trained model and evaluation metrics.

Table 1: Summary of Trained Model and Evaluation Metrics.

Dataset	642 (images & masks)
Defects	4 types
Segmentation Architecture	U-Net
Optimizer	Adam
Loss Function	Jaccard Loss
Dice Coefficient (Test)	0.658

Chapter 5: Conclusion

This thesis paper focuses on data synthesis using Unity and its perception package. The proposed work can successfully generate tons of images in the virtual settings. The real-like images have been used for the training of the U-Net. It is known to all that training the deep learning model needs voluminous images, where image data generation techniques can be useful. The synthetic data were used to train the U-Net architecture. After training of 100 epochs, we obtained the Dice coefficient of 0.658. The model shows reasonable performance in detecting the defects in wind turbine blades. The dice coefficient value could have been better if all blade defects can be identified well in the U-Net model. Leading edge erosion defect is most vulnerable for detection for its shape. If you see the generated images and its respective masks, it is slight straight line in the edge of the blade. Even camera angle from Unity sometimes make it more tough to detect. The slim shape and camera angle make it harder to detect them in U-Net. So, those Leading edge erosion images give very bad dice coefficient results and overall affects the models output. In future, we can improve the model for better segmentation and add the classification module. In addition, we will investigate the work using wide range of defects and real wind turbine images.

References:

[1] Rahman, Md Fashiar, et al. "A deep learning-based approach to extraction of filler morphology in SEM images with the application of automated quality inspection." *AI EDAM* 36 (2022).

[2] Rahman, Md Fashiar, et al. "Improving lung region segmentation accuracy in chest X-ray images using a two- model deep learning ensemble approach." *Journal of Visual Communication and Image Representation* 85 (2022): 103521.

[3] Yu, Junfeng, et al. "An Improved U-Net Model for Infrared Image Segmentation of Wind Turbine Blade." *IEEE Sensors Journal* (2022).

[4] Tian, Weiwei, et al. "A multilevel convolutional recurrent neural network for blade icing detection of wind turbine." *IEEE Sensors Journal* 21.18 (2021): 20311-20323.

[5] Rezamand, Milad, et al. "A new hybrid fault detection method for wind turbine blades using recursive PCA and wavelet-based PDF." *IEEE Sensors journal* 20.4 (2019): 2023-2033.

[6] Márquez, Fausto Pedro García, and Ana María Peco Chacón. "A review of non-destructive testing on wind turbines blades." *Renewable Energy* 161 (2020): 998-1010.

[7] Barker, Jack, Neelanjan Bhowmik, and Toby Breckon. "Semi-Supervised Surface Anomaly Detection of Composite Wind Turbine Blades From Drone Imagery." *arXiv preprint arXiv:2112.00556* (2021).

[8] Rizk, Patrick, et al. "Wind turbine blade defect detection using hyperspectral imaging." *Remote Sensing Applications: Society and Environment* 22 (2021): 100522.

[9] Rahman, Md Fashiar, Jianguo Wu, and Tzu Liang Bill Tseng. "Automatic morphological extraction of fibers from SEM images for quality control of short fiber-reinforced composites

manufacturing." *CIRP Journal of Manufacturing Science and Technology* 33 (2021): 176-187.

[10] Abay, Nazmiye Ceren, et al. "Privacy preserving synthetic data release using deep learning." *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2018, Dublin, Ireland, September 10–14, 2018, Proceedings, Part I 18*. Springer International Publishing, 2019.

[11] Ekbatani, Hadi Keivan, Oriol Pujol, and Santi Segui. "Synthetic Data Generation for Deep Learning in Counting Pedestrians." *ICPRAM*. 2017.

[12] Rabbi, Fazle. "Assessment of fuzzy failure mode and effect analysis (FMEA) for reach stacker crane (RST): A case study." *International journal of research in industrial engineering* 7.3 (2018): 336-348.

[13] Rabbi, Md Fazle, and Bangladesh Khulna. "Assessment of Fuzzy Failure Mode and Effect Analysis (FMEA)…· 2020-06-26· FMEA (Failure..." *Int. J. Res. Ind. Eng. Vol* 7.3 (2018): 336-348.

[14] Kamal, Tamanna, Fabiha Islam, and Mobasshira Zaman. "Designing a Warehouse with RFID and Firebase Based Android Application." *Journal of Industrial Mechanics* 4.1 (2019): 11-19.

[15] Galleguillos, C., et al. "Thermographic non-destructive inspection of wind turbine blades using unmanned aerial systems." *Plastics, Rubber and Composites* 44.3 (2015): 98-103.

[16] Hwang, Soonkyu, Yun-Kyu An, and Hoon Sohn. "Continuous line laser thermography for damage imaging of rotating wind turbine blades." *Procedia Engineering* 188 (2017): 225-232.

[17] Fan, Qiancong, et al. "Ship detection using a fully convolutional network with compact polarimetric SAR images." *Remote Sensing* 11.18 (2019): 2171.

[18] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

Vita

Md Fazle Rabbi is a Research Assistant in the Department of Industrial, Manufacturing and Systems Engineering (IMSE) at the University of Texas at El Paso. Presently, he is completing his M.S. degree in Industrial Engineering and he has completed a B.Sc. degree in Industrial and Production Engineering. Moreover, he is working in the field of Augmented Reality (AR), Virtual Reality (VR), Deep Learning, and Smart Manufacturing Systems for Industrial Applications. Previously, he has experience in the field of Product Development, Computer-Aided Design (CAD), Advanced Quality Management, and Reliability.