TRIBHUVAN UNIVERSITY
IOE PULCHOWK CAMPUS
Pulchowk, Lalitpur

Project Documentation
Minor Project on

Research Environment For Real-Time Visual Element Generation Using GANs

Bachelors in Electronics,
Communication, and Information Engineering

Submission Date: March 27, 2023

Submitted By:

Anju Chhetri (076BEI005)
Pramesh Shrestha (076BEI025)
Prashant Karn (076BEI026)
Tripti Sharma (076BEI047)

Submitted To:

Department of Electronics
& Computer Engineering

Acknowledgments

It has been a tremendous opportunity to learn and explore new ideas for us during the progress of our third-year minor project, which obviously wouldn't have been possible without the opportunity provided by the Department of Computer and Electronics Engineering, Pulchowk Campus, IOE. We extend our sincere gratitude to Assoc. Prof. Dr. Jyoti Tandukar, the Head of the department. We also present special thanks to Assistant Professor Santosh Giri, who along with Dr. Tandukar, helped guide our project on track. We are grateful for the trust placed in us to undertake this minor project as a group and have worked and will work diligently to deliver high-quality results. And last but not least, we would like to thank everyone else who has helped us directly or indirectly in the progress of this project.

Abstract

RESEARCH ENVIRONMENT FOR REAL-TIME VISUAL

ELEMENT GENERATION USING GANS

Anju Chhetri
Pramesh Shrestha
Prashant Karn
Tripti Sharma

Bachelor's in Electronics, Communication, and Information Engineering
IOE Pulchowk Campus, Tribhuvan University
2022-2023

Real-time visual element generation using GAN is a field gaining momentum in recent years. This project particularly explores the generation of grass in video games in the view-space, by using a Neural Rendering model. Since there is not much past work or data available related to this, the scope of this project includes specification of this model optimisation problem, creation of a mock game, highly efficient procedurally generated grass using traditional methods, a system to automatically generate datasets as required with a speed sufficient enough to supply data faster than can be used for training, a real-time testing environment using an integrated neural rendering engine within the render-pipeline, and research work conducting experiments using GAN. The target of this research work is to achieve subjectively good looking grass that can run at more than 10 fps. With the dataset generator and mock game working properly as verified by unit and relationship testing, the research work involved over a hundred experiments exploring five major neural network architectures seen in past work and created in accordance to insights gained from experiments in this project - UNet, Style Transfer, ResNet blocks, SPADE ResNet blocks, and Hybrid models. Many of these experiments produced models that successfully achieved the target of being real-time. Applications of this type of research include better rejection of unneeded information such as shape and identity of grass blades during their rendering, infinite LOD in procedural generation, and faster rendering of complex visual elements more resilient to factors such as number and density.

Table of Contents

List of Figures

(Figures in Appendix section are unnumbered, due to being too numerous)

CHAPTER 1

Introduction

1.1. Project Overview

Our sixth semester of the BEI course, IOE, requires us to work on a **minor project**, where we integrate our knowledge about concepts in software engineering, designing, and programming into a comprehensive project. This documentation provides an in-depth coverage of all the work done for the completion of this project.

This documentation is divided primarily into 4 chapters, beginning with the briefing of our problem statement and our objectives for the project in the first chapter. The second chapter, Project Workflow, covers team workflow and the different phases in which the project was completed throughout the semester. Chapters 3 and 4, Environment and Research developments, dive into more detailed explanations going through the specifics of this project. Each chapter is wrapped up with a concise overview of the sections covered in the summary of that chapter. We have also included many of our interesting findings at the end of the report, in the appendix section.

1.2. Background

Our sixth semester BEI syllabus included two courses that are highly relevant to this project. The first is Object Oriented Software Development, which has been critical for us in understanding how development of any software is carried out in a systematic and structured manner at the industry level or in large-scale research. The tools and techniques learnt in this subject have been helpful for us to understand how this project could be organised in a way that is more modular, reusable and maintainable for future extensions of this project and optimise it so that they suit our objectives better.

Second course that we studied was Project Management, where we learnt to plan, organise the project progression efficiently so that we meet the set goals of the project within the constraints of time, available resources and quality. A comprehensive detail to how this was achieved is covered in Chapter 2 of this documentation.

Other courses that provided us invaluable knowledge to work on this project include Data Structures and Algorithms (DSA), Computer Graphics, Object Oriented Programming, etc.

1.2.1. Motivation

There are two lines of thought to describe the motivation of this project. The first being the fascination with procedural generation, and the concept of neural rendering. Video games are getting more immersive day by day, but there will always be a performance difference between good and bad graphics. Grass is a notorious example of this, being a minor addition to the overall immersion of a game, but taking up massive amounts of resources. The nature of grass is to be numerous, and for computer graphics that means keeping track of all of them. How this is traditionally achieved is mentioned in section *1.2 - Background.*

There is obvious redundant information being stored in valuable memory space here. Grass is easy enough to forget about, and the exact location of every single grass blade does not matter. A different approach where unneeded information such as the shape, location and identity of every grass blade can be thrown away would revolutionise this aspect of computer graphics, and make gameplay faster and better.

Fig 1.1: Snapshots of the game Genshin Impact with grass enabled (left) running at 10 frames per second and disabled (right) running at over 60 frames per second.

Neural rendering seems like a promising approach for this. That served as one reason we were inspired to do this project.

The other line of thinking is that with machine learning projects becoming more and more prevalent especially in university settings such as ours, we noticed something about them. The projects themselves seemed to have a massive bottleneck - the dataset. A large part of these projects tended to be data collection and augmentation. We are interested in learning how machine learning models work, why they work well, and especially the limits of what they can do, for which having a dataset bottleneck really acts as an obstacle. Machine learning models are usually built to make the best of the available data, not make the best of the algorithms and ideas the models themselves operate on.

The design of this project reflects that thinking. We have made it what we call a good model optimisation problem, which sets out to remove such limitations and bottlenecks to truly allow exploration of new ideas.

1.2.2. Model Optimisation Problem

To set the project off, we first came up and specified the idea for the problem to solve. The elements of this include:

A. **Neural rendering of grass using GAN** - basically passing an image such as the one in *Fig 1.1* (left side), to the grassy one (right side). This needs to be done in the post processing phase of rendering, using a neural network taking that image as an input (with other useful inputs if found), and release an output.
B. **Real time** - the model should process and produce outputs really fast enough to be interactive live
C. **In the View space** - Unlike all traditional methods of generating grass in games, including procedural grass and billboard grass, we want to send the view itself, the pixel by pixel data, to our grass generator to basically draw it on top of the screen, instead of sending in 3D environment's data. There are a lot of advantages to this approach, which we have detailed in our documentation, and we will touch on them towards the end of the presentation, when we talk about future prospects and possibilities.

Considering this criteria, we could not find any solutions online. Thus, this is new territory for research.

A model optimisation problem in this context is a challenge involving getting the most out of the architecture of a machine learning model.

To turn these criteria into a **good model optimisation problem**, we defined the following characteristics for what an optimisation problem should have:

A. **It is not limited by the dataset:** Perhaps the most important characteristic of a model optimisation problem is that we aren't limited by the data. It might be easy to see how most machine learning projects struggle with data. The model might be very capable, but without enough data, those capabilities simply cannot be realised. Most of these projects sort of become data collection projects rather than model creation projects, since it is simply so difficult to gather it. They try to use the model to make the best of the data, but not the best of the model itself. In order to optimise a model, we need to test its limits - see exactly how capable it is. Only when we know how capable a machine learning model is, can we work to make it even more capable. And for that, we need abundant data - as much as the model needs to do its best. And that's what we did, we built a way to have an unlimited amount of data in our project.
B. **Chaos and unpredictability are minimised**: These are the images we want to draw grass on top of. You can see that they are very similar. We have designed this very specifically, to only have variation in the places we want variation. We don't have unexpected objects in the scene, or unexpected changes in the frame progression. We don't have chaotic lighting, or obstructions. Sure, a model needs to be able to tackle unexpected situations, but we have found that that is not a good approach to explore

new territory. To truly gauge the capabilities of brand new ideas, we want as few changing variables as possible, to see where the correlation lies, and why the new ideas are useful. Only when the ideas work in that context, can we improve it by adding more chaos. That is why we designed our model optimisation problem this way.

C. I**t is specific and well defined**: When we say we want to draw grass on top of an image, we only mean in this context. For example, our exact mock of a video game, with these rolling meadows and blue sky, and only grass present on the land, with 800 x 400 pixels of the view, the depth map, and the normals map, are examples of specific criteria. It must also be tested on a particular hardware - for our context it is the Nvidia GTX 1650. Only when we standardise these aspects, we can compare the capabilities of competing models where it truly counts - the performance.

This project has been designed to define this problem, attempting to meet the requirements of those characteristics.

1.2.3. Why Grass?

Grass for a long time has been symbolic of a simple, ordinary aspect in reality that has been impossibly difficult to portray realistically in video games. The common approach to this has been to simply use flat textures of grass on top of polygons on the screen, perhaps with normal mapping for some added depth. Vertically oriented polygons with alpha based textures and grass blades, sometimes animated, are often used for representation of taller grass.

Recently a new approach has become visible at the horizon, neural rendering, allowing for storage of the immense detail of instantiated grass only in the view-space, in one or multiple frame-buffers, which could be orders of magnitude cheaper than procedural grass.
This is a very active field of research, with a lot of potential, and a very real challenge in not only making it realistic and accurate, but also fast and as lightweight as possible. Grass is the ideal candidate to try to achieve real-time neural rendering, having complex elements difficult to emulate using traditional methods, but a simple overall picture that may be more easily picked up by a lightweight machine learning model.

1.3. Project Map

The following is a diagram of the project lifecycle model designed for this particular project.

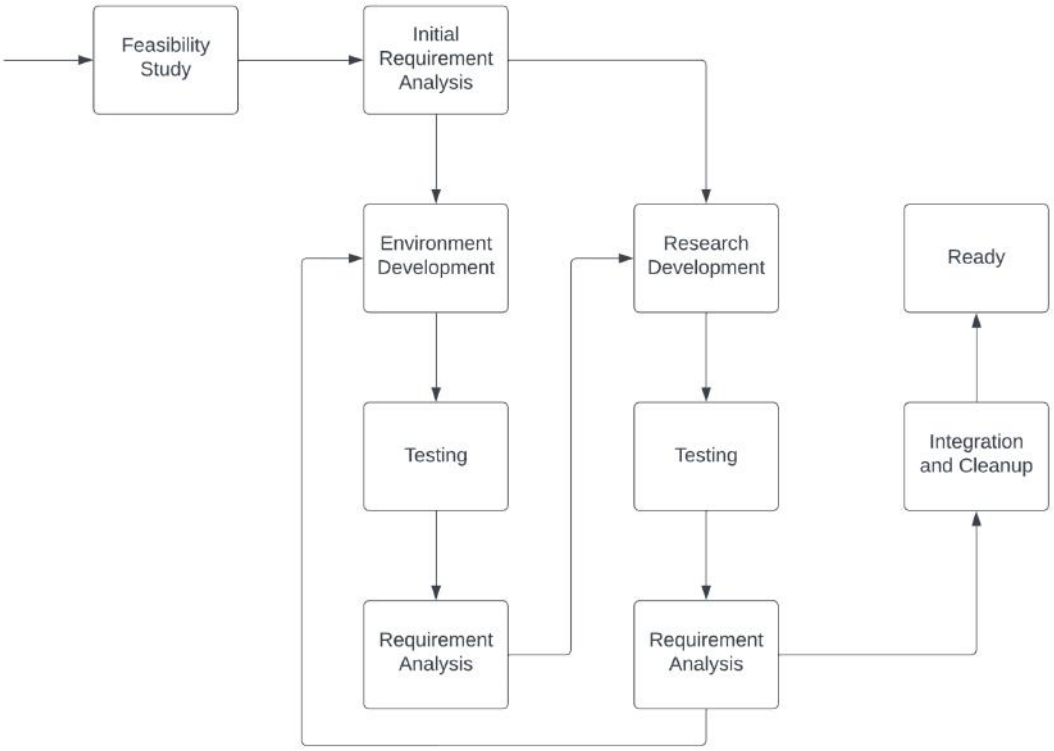


Fig 1.2: Project lifecycle model designed for this project

It makes use of a custom project development life cycle where we take the environment development and the research development parts of the project in parallel.

1.4. Problem Statements

The following are the problem statements explored in the concept of this project:

A. Traditional methods of generating grass in games are either flat with very little depth (normal mapping), or computationally very expensive (procedural generation).
B. Fine details like grass-blades do not need to be remembered since it is very hard to notice inconsistencies in them during normal play, as they are mostly peripheral elements. In traditional methods, such information cannot easily be removed while preserving the collective appearance.
C. Neural Rendering so far in games has always been very large-scope, trying to generate everything at once, with varying quality but almost always slow result.

1.5. General Objectives

In order to tackle those problems, our objectives are as follows:

A. A mock game designed to perform this type of research, able to provide a natural gameplay experience especially regarding navigating a grassy landscape
B. A dataset generator, which can emulate natural gameplay in the mock game and save results of various render passes including the grassless view, the depth map, the optical flow map, the normals map, etc. as required by the research progress, in a time-scaled frame-by-frame progression
C. A real-time testing environment, which can run a neural network model as part of the rendering pipeline of the game, provide it the required inputs, and display the output
D. Experimentation with neural network models, improving them to the point where they can generate fairly good looking grass in real time

1.6. Specific Objectives

After performing the feasibility study of the project, and analysing the problem in accordance to the model optimisation problem criteria, we set the following as our specific targets for this project

A. A mock game with an infinitely generated terrain with no repeating features, with the environment fully navigable through common controls for a 3rd person view game
B. An active and movement-filled character to represent our 3rd person view's protagonist, to act as the aspect to ignore during grass generation processing
C. A system to generate grass using traditional methods, realistically portraying the shape and behaviour of grass, making efficient use of the GPU
D. A dataset generator that can save images in a reasonable amount of time and can be left to do it on its own. We suspect we need about 20,000 training data items to train the model, so this must be able to generate that within a day's time
E. A real-time testing environment with minimal overhead for the render passes for the inputs and displaying the output - about 0.05 ms limit to accomplish everything to be done in the frame other than the neural rendering
F. A neural network processing pipeline integrated to work with the GPU in the real-time testing environment
G. Exploration of models used in related work, as described in section *2.2 - Literature Review*, as starting points, and improvement of those models or combination of ideas presented in those models to generate subjectively good looking grass independently for each frame, at the rate of at least 10 frames per second.

1.7. Project Scope

The scope of the project is summarised below in general terms. Since the specific aspects of this are dynamic, it has been described as required in the detailed parts of the report.

A. Definition of a model optimisation problem for view-space neural rendering of grass
B. Determination of the appropriate specifications and dataset for such a problem
C. A system to generate such a dataset and of an appropriate size in a reasonable amount of time
D. A real-time interactable environment that can quickly provide inputs as specified in the dataset
E. A GPU based tensor processing pipeline that can run those inputs through a neural rendering model, and provide an output in real-time, in the aforementioned environment
F. A neural rendering model development environment to create and refine the aforementioned neural rendering model
G. A software process model and a custom workflow model designed for the development of the neural rendering development environment and the real-time interactable testing environment in parallel
H. A feedback system for the two development processes to help with each other's progress
I. An environment to be able to easily train machine learning models and save its output and progress
J. An organisational system to keep track of experiments regarding the machine learning models
K. Experimentation of machine learning models to reach the specified objectives

CHAPTER 2

PROJECT WORKFLOW

2.1. Feasibility Study

The feasibility of the project was assessed for the two constituents of the project: the environment development and the research development.

The environment was to be developed in the Unity engine, which was well within the expertise of at least one of our team members, and thus the feasibility analysis was done by them. Examples of potentially required programs were found in either past projects, or online. Parts of this work that would be new included possibly managing inter-process data transmission or communication at very high speeds, optimising calculations using the GPU, and writing compute shaders. Enough examples scattered throughout online tutorials were found to consider these aspects feasible.

The research development feasibility study was done by studying past research found online through sites such as Google Scholar. Examples include the paper by Mittermueller et al. [8] , the one by Richter et al. [7], and the famous Pix2PixHD paper by Wang et al. [9]. These papers were not studied for the methodology, but for the result they achieved, to analyse the similarity of those results to what we want, and thus assess our project's feasibility. See *2.2 - Literature Review* for further details about that.

In none of the past work we found, however, was the generation done in real-time. This is new territory, and thus the feasibility would have been unknown without more information for judgement.

This judgement came from the existence of real-time image generation in applications such as face changing filters in Instagram. These can be seen running neural network models complex enough to produce those effects, but fast enough to be real-time in mobile hardware. Further information about this however was not found, and it was suggested that research and development for these were done behind closed doors, and not revealed to the public. Talking to an expert, we were given an estimate that these models would have around two million trainable parameters, which is only one factor that could affect the speed, but provided us further information for the judgement of this project's feasibility.

It was clear however, that this would become a research project, where new territory will have to be explored, and new ideas tested and implemented. Talking to experts specialising in machine learning algorithms used in video games, this type of advancement in the speed and

efficiency of this type of task was suggested to be a research project that could take years to produce solid results actually useful in games. We thus pulled back our scope and set our target to the first milestone among the ones described in section *6 - Conclusions and Future Work*, which was seen to be feasible enough.

2.2. Literature Review

Upon reading about related work and observing the methods used in games, it has become evident that the common approach to the generation of grass has been to simply use flat textures of grass on top of polygons on the screen, perhaps with normal mapping for some added depth. Vertically oriented polygons with alpha based textures and grass blades, sometimes animated, are often used for representation of taller grass. The industry standard uses such techniques paired with procedural grass blade generation, along with procedural animation for realistic movement, a technique pioneered as far back as 2001 [1], and vigorously improved over the years, such as moving the animation component of the pipeline within the shaders [2]. However, this notorious impression of grass rendering comes down to the disparity between its simplicity and where the complexity lies - the incredibly large number of blades it constitutes, while the shape and whereabouts of each of them is not crucial information to track as long as the collective is realised. Procedural generation of 3D grass in the game environment solves half the problem by only making it necessary to store the information of nearby grass in primary memory rather than all grass in secondary memory, allowing arbitrarily large environments with O(1) space complexity for the grass. It is still, however, very computationally expensive.

We discovered that recently a new approach has become visible at the horizon, one allowing for storage of the immense detail of instantiated grass only in the view-space, in one or multiple frame-buffers, which could be orders of magnitude cheaper than procedural grass. This is possible using Neural Rendering, which has taken the graphics world by storm, presenting countless use cases that used to be impossible [3]. Generative Adversarial Networks (GANs) are a big part of this, being able to synthesise complex imagery without having to explicitly store information about the instantiated details. GANs have reached the stage of aiding procedural generation in 3D environments already [4]. Stable Diffusion [5] and similar image-to-image generation models can already take a simple, flat rendition of grass and add detail with depth to it. Wang et al. [6] present a way to replace all of the lighting and most of the rendering pipeline with GANs, to generate every visual element on the screen using Neural Rendering. Richter et al. [7] take a different approach, style-transferring realistic visuals on top of fully rendered video game scenes. Mittermueller et al. [8] provide some lighting information, and various maps, including untextured or lightly textured albedo, normal and depth map of a scene along with semantic labels on view-space segments to their CycleGAN based model to generate a realistic scene, combining the two approaches into a realistic, but crucially not yet real-time, application. The natural next step is

of course real-time application of Neural Rendering, and grass, having its long-time symbolic stance as an impossibility, is the perfect visual element to start on.

2.3. Requirement Analysis

The initial requirement analysis was done mostly through educated guesswork. Since the exact requirements of the project could not be immediately known without some exploration and experimentation, this served to provide only a starting point. These are the initial functional requirements for the project:

Playground Environment Development

A. Infinitely navigable environment, using a first-person camera view
B. Randomly generated terrain
C. Procedural grass generation for reference
D. Live creation of training data
E. Ability to load neural rendering models
F. Ability to run neural rendering models in the GPU
G. Display of the output of the neural rendering model on the screen
H. Real-time running of developed neural rendering model

Research Development

A. Loading of a dataset
B. Batching of a dataset for GPU based optimisation
C. Logging of required changeable values
D. Visualisation of logged values in form of graphs or charts
E. Ability to run neural network models for both forward and back propagation
F. Ability to package and save neural network parameters
G. Ability to export neural network models and saved to a format recognisable by the environment

The requirement analyses done during the parallel development of the environment and the research - the feedback provided by them - have been much more ubiquitous in the development of this project. Each of these feedback requirement analysis phases was conducted based on observations of the experiments done in case of the research development, and limitations found in the implementation of the environment aspects in case of the environment development. We have presented three of these requirement analysis results in the appendix section of this documentation (*see section 7.2 Appendix B).*

Retrospective Analysis of the Requirement Analyses

We went through about 10 requirement analysis phases, with a lot of nuanced changes with the requirements as per the feedback provided by both the environment development process and the research development process to each other. A simplified view of this is present in the three-way simplification presented in the Appendix section *(see section 7.3 Appendix B)*. Examples of these are also included in the Appendix section *(see section 7.3 Appendix C)*.

2.4. Development Cycle Methodology

Our development cycle has been design with the following objectives in mind

A. Dividing the work into tasks that can be done by any team member available with the required skills. Tasks are assigned to skills rather than to team members.
B. Efficient division of tasks to maximise parallel execution of independent tasks
C. Making all types of tasks available and understandable to the team members, in order to ensure overall learning for everyone.

2.4.1. Custom Development Cycle

We developed our own project development lifecycle, as shown in the Project Map in section *1.3 - Project Map*

Here are the reasons behind the particular design we ended up with:

a. Since our project contains two constituents, the environment development and the research development, we thought it would be most efficient to take those two in parallel
b. We assessed the importance of environment-research feedback, as discussed in section *2.4.2 - Environment-Research Feedback*, and integrated the idea into the project development lifecycle
c. We decided to use a Task based workflow (as described below), which allowed for such parallel tasks

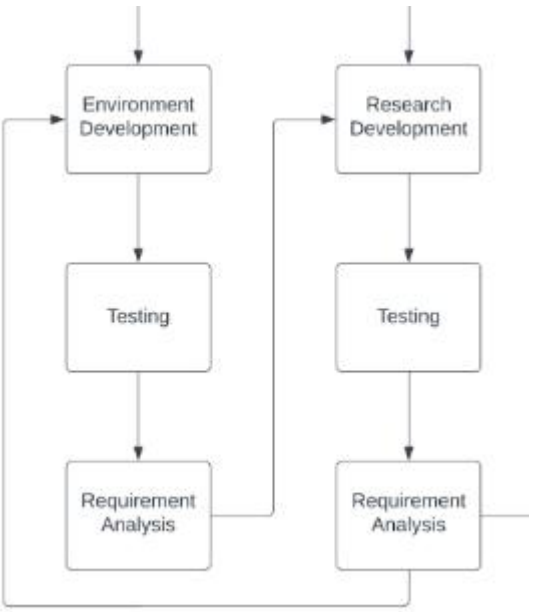


Fig 2.1: The parallel development and feedback section of the project map

During the development phase in each of these flow paths, we build the systems as required. In case of research development, we build the models to be used in the experiment, and come up with the configurations to be explored.

During the testing phase, we perform unit testing and relationship testing in case of the environment development path, and we conduct the listed experiments in the research development path.

During the requirement analysis phase, we observe the results of the testing and make a list of any new requirements from the other path. If there are none, we simply return to the development phase for the next aspect of the system.

2.4.2. Environment-Research Feedback

One of the most important aspects of the project development dynamic was determined to be the Environment-Research feedback. The goal of our project first and foremost is to create an appropriate environment with necessary tools to help with the research task. We however did not initially know all the details and requirements of the type of research. To determine that, we performed everything that will be required for such a research, and designed, trained and fine tuned many machine learning models, conducted experiments, analysed and verified results, and made conclusions about what would be needed to improve results in the next run, especially regarding the input and output information in the dataset.

2.5. Project Management and Approach

We developed a workflow system that we call the “Task based workflow”, based on past experiences particularly with Prashant Karn’s game development career working with small teams.

2.5.1. Tasks

The tasks can be described as follows:

a. The work is divided into tasks, each task being accomplishable within a day’s time, in a few sittings
b. Any given task is characterised by its goals, directions on acquiring required resources and submitting its result, and a deadline, and also a parameter called skill, which will be explained below
c. Tasks are connected in a graph form, in parallel and series combinations, reminiscent of resistors in a circuit diagram
d. Parallel tasks can be done independently from one another, while series tasks require the previous tasks to be done

2.5.2. Skill List

Each task also has a skill list parameter, which we determined is more useful for assigning tasks than having dedicated members with individual roles. Thus, any team member with the required skills can take up a task which is to be done, and has all previous tasks already completed.

In this way any idle team member can find a free task to complete it. No task is assigned to team members when made. Tasks themselves can be taken up by multiple team members, if it is clear that would be more efficient.

The tasks were kept track of in a kanban style board chart in Notion. When required, more complex graphs were visualised using task-flow diagrams, an example of which is given below.

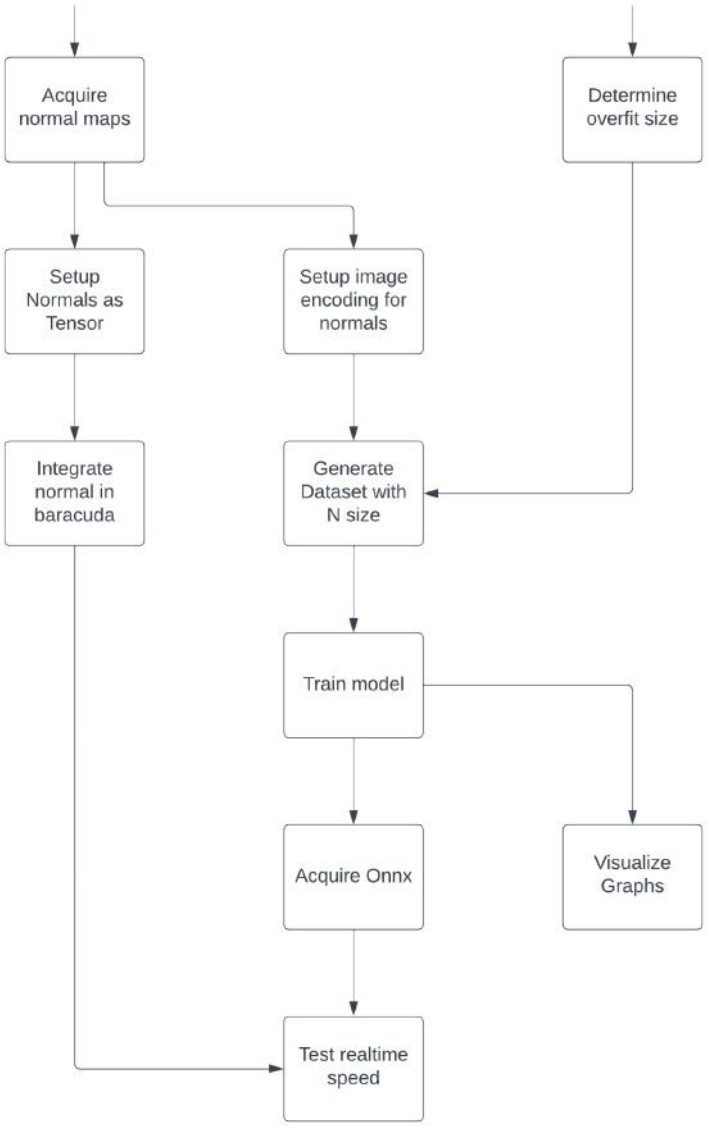


Fig 2.2: Sample Visualisations for Task Flow Diagrams

2.5.3. Background Tasks

Certain tasks such as model training could be supervised while running in the background, while doing a different task in parallel.

2.5.4. Learning and Efficiency Balance

As important as efficiency is for projects such as this one, we decided to prioritise one factor more: learning, since it is important for all team members to fully understand the project. Since all of us may be involved in any subsequent research projects built upon it, we set the

skill parameter loosely, opening up the option to learn the skill while working on the task, either from online resources, or with another team member as a teacher. We also made sure to let every team member have a hand in a variety of tasks, including parallel tasks spanning across the environment development as well as the research development.

2.6. Chapter Summary

The feasibility study of our project was done mostly by reading preceding work to our project similar in the scope of both GANs and generation of grass graphics. After extensive reading, we assessed the requirements of our project in several phases. A major initial design aspect worth mentioning was having a customised development cycle so that a feedback-based, parallel workflow could be achieved between environment and research development since such a project still remains unexplored territory and requirements were not known until our experiments were run.

For the project management, larger tasks were first broken down and parallelised as far as possible and then assigned and then assigned to the team members according to their skills on a Kanban board in Notion. Although efficiency has remained the priority of the project throughout, the key takeaway of the project was to learn and build our skillsets for future projects and this remained the core motivation behind doing this project.

Relevant coursework studied in this semester, namely Object-oriented Software Engineering and Project Management has been very helpful in designing this workflow and breaking down the projects into different manageable and maintainable sections. We were able to set our objectives clearly and manage resources, time and quality efficiently using the tools and techniques that we learnt in these subjects.

The changes made throughout the development cycle of the project strongly signify the importance of feedback between the two project constituents, and it can be concluded that the designed workflow performed well for this mechanism and is apt for this type of project. Examples that make this evident can be seen in the Appendix section *(see section 7.3 Appendix C)*.

CHAPTER 3

ENVIRONMENT DEVELOPMENT

The Environment in the context of our project refers to both the dataset generator and the neural rendering playground. Both of those rely on a simple game interface to work, which includes a third-person view of a playable character navigating in an infinitely generated landscape. The design of this landscape, the constituents of the game interface, and the abilities of the player were constructed to keep the model optimisation problem of grass generation simple enough to be feasible, but complex enough to require new ideas to be researched. An initial design of this was constructed, which was subsequently improved by observing the results from the research development work to fit that goal. Examples of this can be seen in the Appendix section A (*7.1 - Experiments Conducted and Architectures Implemented*) and Appendix section C (*7.3 - Examples of Environment-Research Feedback*)

3.1. Development Engine and Methodology

The playground environment and the real-time running of the neural rendering model have been set up through Unity's barracuda library, and are working fully as intended, being tested with lightweight models working just about within the definitions of "real-time". Since barracuda is incomplete, many of the problems faced were worked around either in the engine or in the neural rendering model, and are thus open to further optimisation.

3.1.1. Game Environment and Interface Design

The Game environment consists of an infinitely generated terrain, with a controllable player character in third-person view. In the initial requirement analysis of the project, it was thought that the visual of the player would be part of the inputs sent to the neural rendering model, and thus it was programmed with elaborate movements to provide better quality learning from the model.

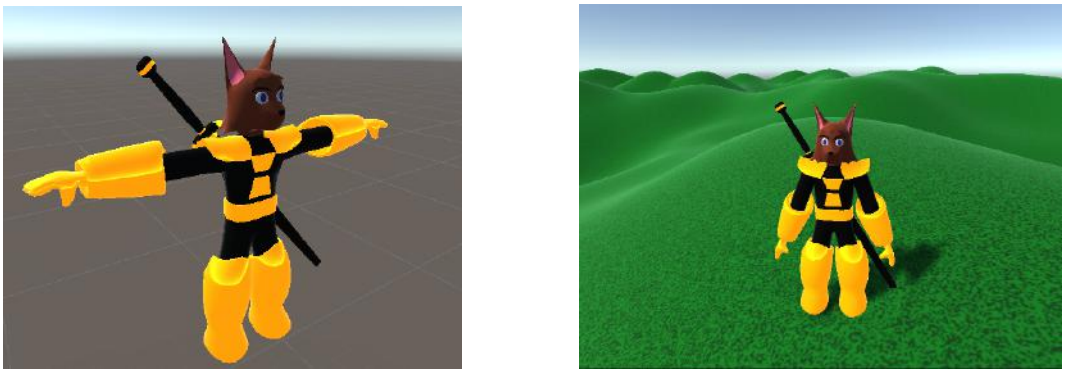


Fig 3.1: Player (left) and the landscape the player is in (right)

The player can move forwards with two speeds - walking and running. The player can also jump, do back-flips, and get into a fighting position with lots of movement. The controls for this are as follows

- 'W' or Up Arrow key moves the player forward with the walking speed
- Holding 'Shift' while walking makes the player run, with about three times more speed than walking
- 'Space bar' key pressed while the player is grounded makes it jump
- Horizontal movement of the mouse turns the player left or right
- Vertical movement of the mouse turns only the head of the player up and down
- 'Right click' with the mouse makes the player take a fighting stance, during which rotation movements are slowed down
- 'Left click' with the mouse uses the current weapon to attack in the direction of the mouse pointer from the centre of the screen
- 'E' key changes the weapon (from fists to sword and vice versa)

These controls were programmed using a priority decoder implemented in C#, effectively being a state machine. For every state, we also created a corresponding animation, which was built using Unity's animation graph state machine. This state machine can act as a visual representation of both the animation states and the control states.

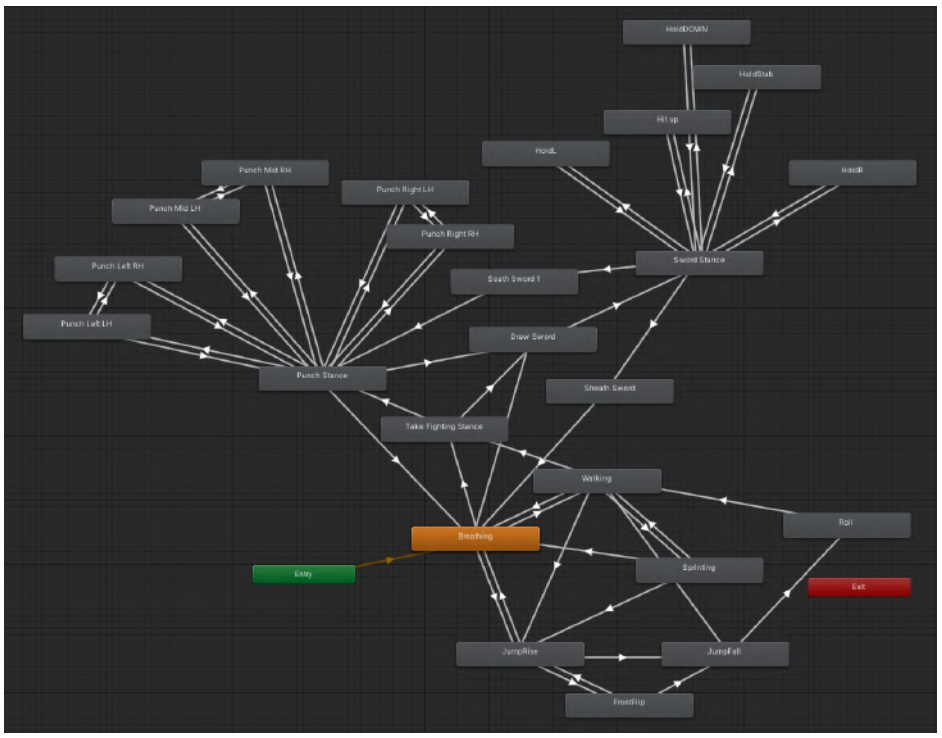


Fig 3.2: The state diagram describing the states controlling (and animating) the player

This game environment is used for both the dataset generator and the neural rendering playground. The interface of the dataset generator contains buttons for capturing frame, automatic frame capturing, automatic random movement, and teleportation. It also contains toggles specifying what types of images to save. It gives us crucial information about the current state of the program, such as the frame rate, elapsed time, number of frames captured, etc.

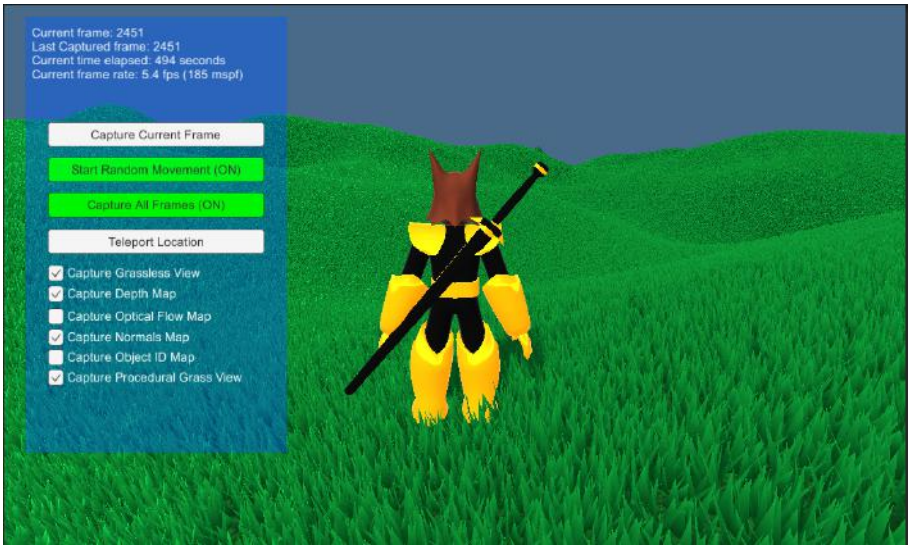


Fig 3.3. Interface for Dataset Generator (with procedural grass enabled)

Each of these buttons also has a corresponding hot-key, along with hot-keys for enabling or disabling the grass in the current view.

The neural rendering playground consists of two configurations in the context of the interface: the output view and the quad view. The output view displayed only the output of the neural rendering model, while the quad view can view up to three of the inputs provided. Information about the frame-rate and so on has so far not been implemented in a custom UI, since unity provides a stats window to gauge them, which has been sufficient.

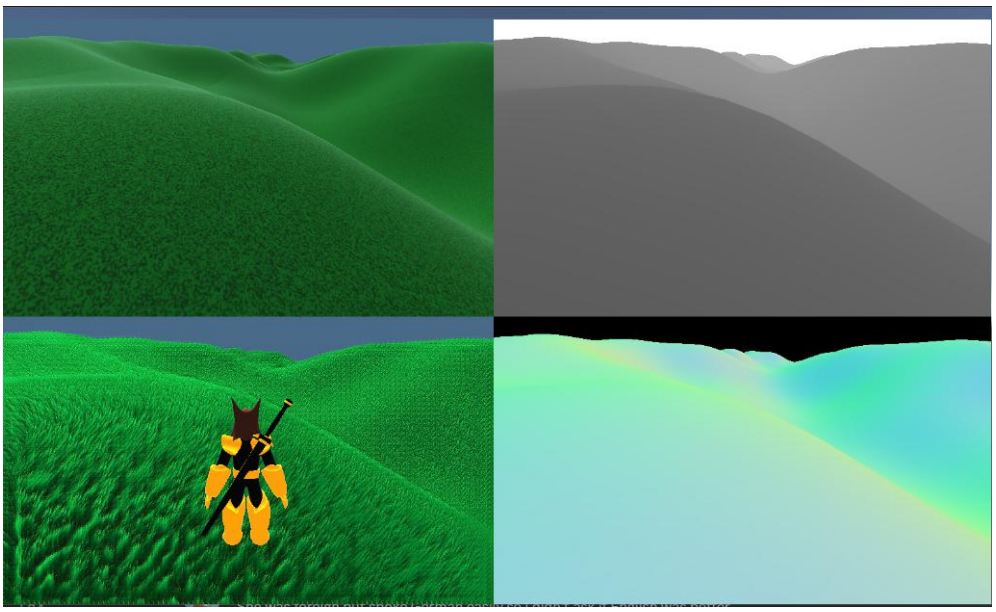


Fig 3.4: The quad-view of the Neural Renderer Playground, containing the input view (top-left), depth map (top-right), normals map (bottom-right) and output view (bottom-left)

The NNmodel to be tested can simply be dragged into the inspector field of the Barracuda singleton object, and the navigation of the player can then happen normally. Unlike the dataset generator, which displays the direct output of the render pipeline of the Unity engine (but uses render textures to save the view), the Neural Renderer Playground's quad display consists of four rectangular meshes, the UV coordinates of which are filled using materials controlled by Render Textures.

3.1.2. Render Textures

One very helpful component in Unity is the Render Texture, which can display a camera view onto a UV-mapped texture in the game environment. We used this to create the quad view for the Neural Renderer Playground, but more importantly, these views are saved in Render Textures as pixel-defined textures, which can be turned into tensors required for neural rendering models' inputs.

The major Render Textures are used in the project in its current state are as follows

1. The ***Save-Helper Render Texture***, used by the Dataset Generator to save snapshots of the current frame
2. The ***Current View Render Texture***, holding pixel data of the landscape before processing, and without the character model

3. The ***Player Render Texture***, holding pixel data of only the character model
4. The ***Depth Map Render Texture***, holding pixel data of the depth map, provided by a camera set up with a depth map shader
5. The ***Normals Map Render Texture***, holding pixel data of the view normals, provided by a camera set up with a normals map shader
6. The ***Output Render Texture***, holding pixel data provided by the Neural Rendering model

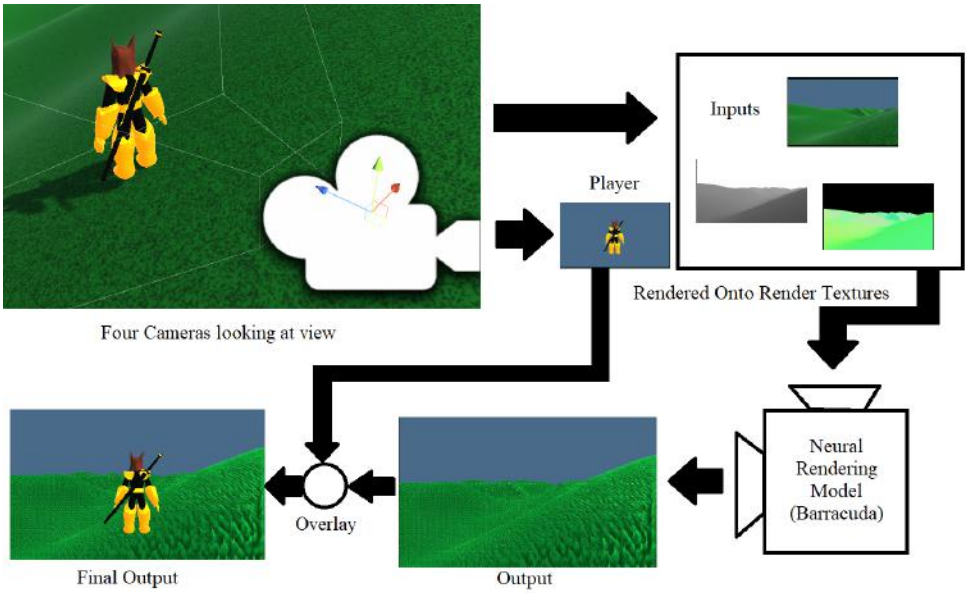


Fig 3.5: The rendering process, using render textures

In the Neural Rendering Playground, for the currently finalised set of inputs, four Render Textures are used to hold the view being looked at by the four cameras watching an identical scene, but set up with different culling masks and shaders. Three of those (the defined set of inputs) is then sent to the Neural Rendering Model where the pixel data is translated into Tensors. The Neural Rendering Model then outputs a tensor, which is converted into pixel data and saved onto another Render Texture. The Render Texture holding the pixel data of only the player is then overlaid on top of the Neural Rendering Model's output rendering texture, to form the final view displayed. One drawback of this method is that player to world interaction visuals such as shadows are lost. This method is however considered acceptable for our current goals, since the player visual itself is not part of the Neural Rendering Model's processing.

3.1.3. ZMQ

ZMQ is a C# and a Python library useful for inter-process communication. It is not used in the current state of our project. However, it provided vital information in the initial stages of the project, and thus was considered worth a mention. As mentioned in section *2.3 - Requirement Analysis*, the initial idea of the project did not include a saved dataset, but instead a live transfer of the inputs involved in the dataset to the python program training the

model. These inputs were supposed to train the model during the training phase, and display the output of the model during testing phase, acting as the Neural Renderer Playground see section *3.4 - Neural Renderer Playground*. It was however considered too slow and thus a significant limiting factor to the speed of the operations it would be involved in. It would have provided a greater amount of flexibility, but not enough to overcome its disadvantages. We learned in detail its working, and are highly confident it will be extremely valuable in future projects where a smaller amount of data transfer is being done between the machine learning model and Unity, such as text data in NLP applications.

3.1.4. Barracuda

Unity's Barracuda library provides the necessary tools to run neural network models defined by the ONNX standard. After the failure of the ZMQ based system to meet the required speeds, Barracuda was considered the next option with the highest prospects. Barracuda contains definitions of tensors, tensor operations, and GPU optimisation for those operations. It can also be used to set up a GPU pipeline to run a neural network model (defined as NNModel) into which an input can be passed, and an output retrieved. This is crucial in the working of the Neural Rendering Playground in the current state of the system.

This was however a particularly difficult task due to this being a new library with examples few and far between and incomplete documentation. Many common operations such as concatenation were painstakingly found to be unsupported and thus had to be worked around either within the rest of the unity engine or within the neural rendering model itself.

Research Development, as described in the methodology in section *4.1 - Development Methodolgy*, primarily using pytorch can publish the trained models, with defined inputs, outputs, layer definitions, etc. into an ONNX file, which can directly be read by the functions provided by Barracuda to convert it into its native NNModel class, which is used to create that aforementioned GPU pipeline.

Pytorch, ONNX and Barracuda all use different orders of dimensions (batch, channel, width and height) to define images and tensors used in image processing CNN's. The translation of these was a particularly difficult hurdle to cross, requiring debug screens and displaying of tensors as numbers, but was eventually achieved - providing valuable skills for solving and debugging such problems.

3.1.5. ONNX

The ONNX standard can be used to define neural network models, providing both definitions of the layers, operations, inputs, outputs, etc. in terms of tensors, as well as the learned parameters used in the neural network model.

3.2. Static Design

The Static Design of the system refers to the aggregate of all components working together and interacting with each other for both the Dataset Generator, and the Neural Rendering Playground. Both share a large number of classes, especially singleton classes, and bigger units such as prefabs. The structure of the scene with the tree organisation of parent and child objects as well as the singletons are shown in the respective sections for those scenes, in sections *(see sections 3.3 Dataset Generation and 3.4 Neural Rendering Playground).*

The class diagram in figure 3.6 covers all important classes made and interfaces realised for infinite terrain generation with procedural grass for generating the datasets, corresponding image maps production, mesh generation for the terrains, and automated navigation of the animated character in a 3D playable environment with required camera rotations.

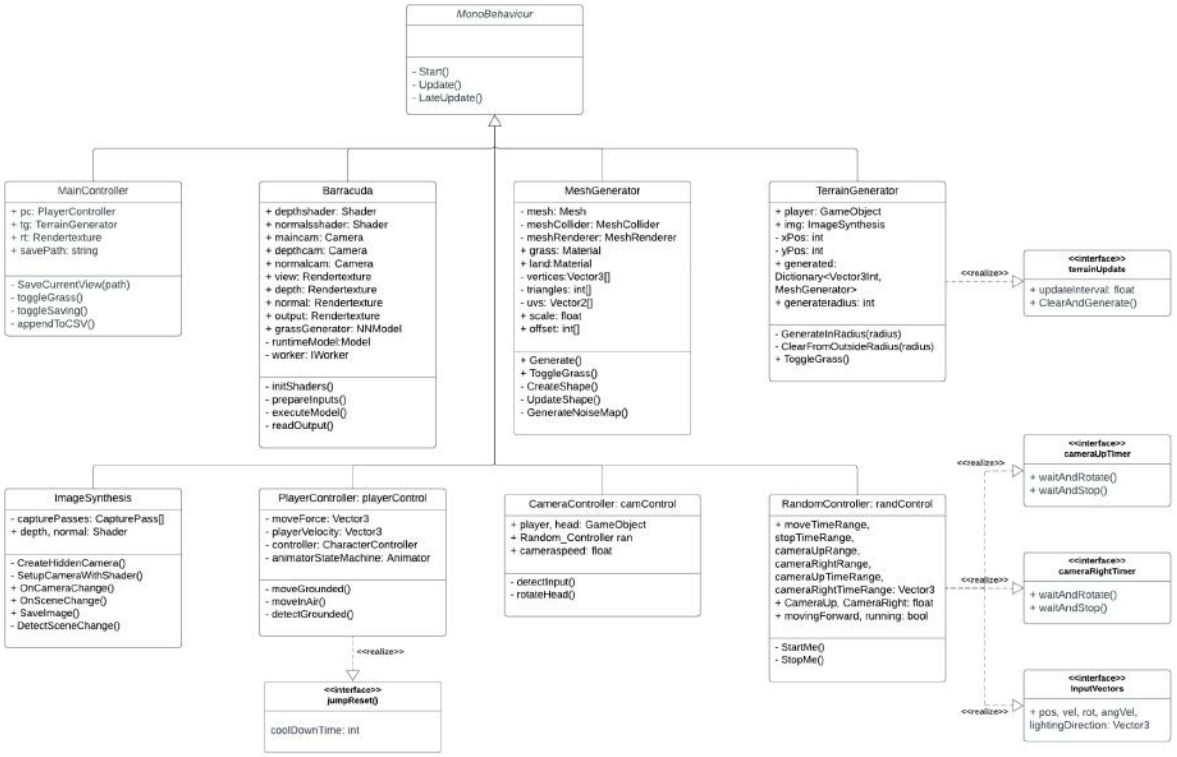


Fig 3.6: UML Class Diagram for Playable Environment Development and Rendering

All classes inherit from the Unity parent class MonoBehavior which controls the start and is responsible for updates within the gameplay loop. Class Barracuda is used to interface the python-based neural network model in onnx file format with the playable game environment in Unity for rendering purposes.

All of these classes are implemented in C# to be compatible with Unity. Some changes had to be made from that original design for further optimisation, but it stays true to the diagram for all intents and purposes.

3.2.1. Infinite Terrain Generation

Terrain Generation was achieved using vertical displacement of the vertices in a plane-subdivided grid.

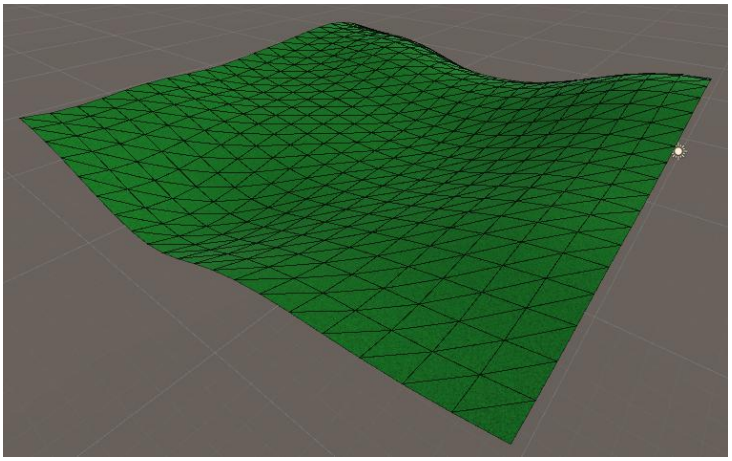


Fig 3.7: Terrain generated using subdivided grids on planes, displaced vertically

The vertical displacement is determined by two-dimensional Perlin noise, provided by C#'s Mathf library. Multiple instances of the noise with different scales were layered to achieve a more detailed look.

The world is divided into square chunks, and only the chunks around a certain region of the player are generated. This is done by running a coroutine that occasionally performs checks to make sure of this fact. The region around the player is defined as a circle characterised by a maximum square distance from the player to any given chunk coordinate.

The coroutine is part of a system with a dictionary containing all loaded chunks, with their coordinate vectors as keys. The coroutine performs checks every few seconds for two things, and performs the required action when the condition is true:

- Whether there are chunks in the dictionary that are outside the radius to be loaded around the player; delete those chunks if there are
- Whether there are coordinate vector keys within the radius to be loaded around the player which does not exist in the dictionary; generate chunks for those keys if there are

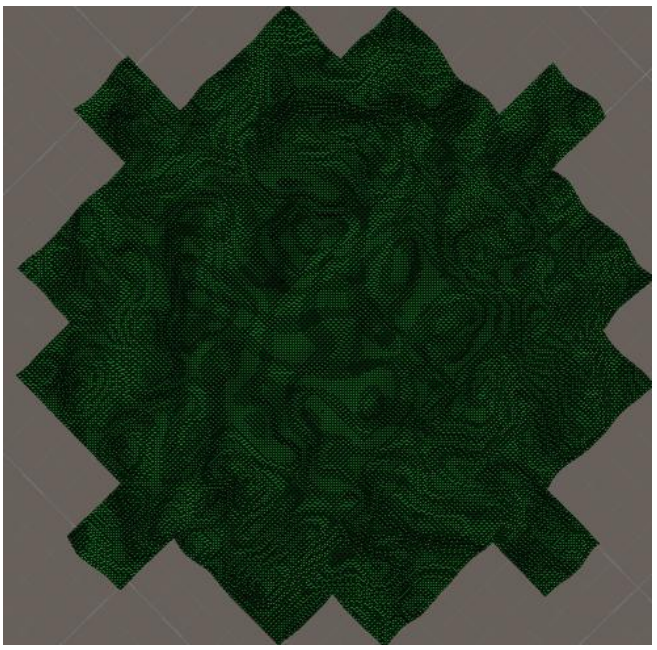


Fig 3.8: square chunks loaded within a radius around the player

Once the coroutine decides to generate the terrain chunk in the newly introduced area inside the radius around the player, it calls the MessGenerator class to create an instance of a mess class for that chunk. The coroutine passes it the x and z coordinates and then the MessGenerator uses layered perlin noise to create its y coordinate.

3.2.2. Procedural Grass Generation

3.2.2.1. Design and Requirements

To provide a baseline of the ideal grass to be generated by the neural rendering model, a traditional method of creating grass was required. Many methods were reviewed, including textured grass, billboard grass and generation of individual grass blades. Textured grass was considered too low quality and fast enough to not require a neural rendering alternative. Billboard grass is the industry standard and the most commonly used grass, but it was not chosen for our application for the following reasons:

1. Billboard grass causes grass blades to line up unrealistically. This is not an issue if it is dense enough, but we would not want the ML model to pick up on patterns such as that which do not exist in the real world
2. Shadows in Billboard grass may not be realistic
3. The movement and orientation of the grass blades is limited within or with the plane onto which they are present, which is another pattern we would not want the ML model to pick up

We finally simply chose to render every single blade of grass separately, to make the curving, movement and shadow dynamics as realistically as possible. This method is much slower than Billboard grass, but we considered it a worthwhile compromise.

In retrospect, the Neural Rendering models that we have trained so far have not been able to pick up enough detail for it to matter if we had used Billboard grass in the first place. The research is not complete, however, and those problems could arise in future iterations of the research development. We are open to replacing the individually generated grass with Billboard grass in the future if such pressure for optimisation arises.

The generation of this procedural grass was done on the basis of the tutorial provided by Roystan [L1] Roystan's implementation was not meant for the large environment of our usage however, so optimisations were done in that regard, with help from shader tutorials provided by CatLikeCoding [L1]. Roystan's tessellation implementation was also not appropriate for this large scale, creating repeating grid-like patterns, and thus Voronoi tessellation was implemented from scratch to achieve better random distribution.

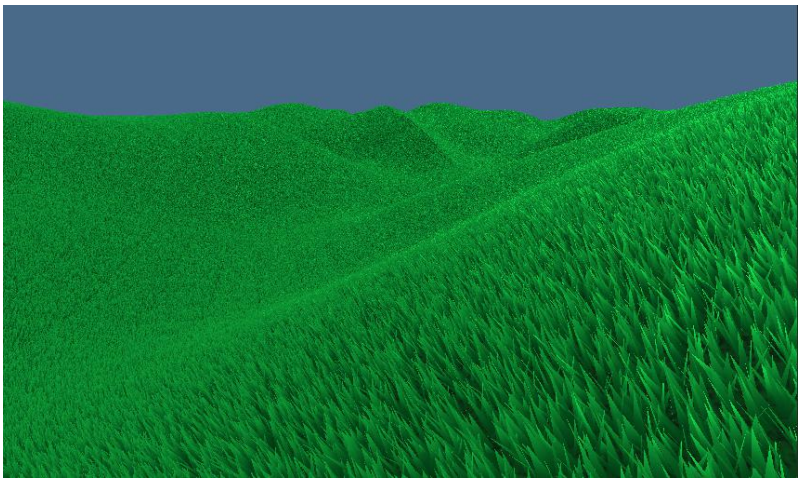


Fig 3.9: Final image generated with procedural grass

3.2.2.2. Creation of a single blade of grass

Every individual grass blade is a mesh made up of five triangles as shown in the figure 3.10. The parameters defining the grass blades include its width, its height, its curvature, and the position of its root in the environment.

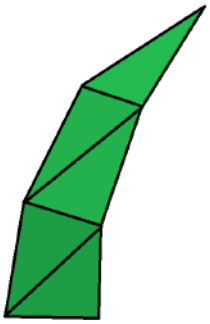


Fig 3.10: Structure of an individual blade of grass

Characteristics of it include its movement speed and vertex colouring. The vertex colouring is performed by using interpolation of the tip colour and the root colour. The root colour is made darker than the tip colour to simulate light loss in depths of a grassy covering. The movement speed is defined by a wind-texture. This wind turned out to be too high-level of a feature for the Neural Rendering model to learn about. However, in future research focussing on temporal consistency, it might become relevant.

3.2.2.3. Tessellation for Randomisation

The roots of the grass are defined to be every vertex present in the landscape. Due to the limitations of the terrain generation speed however, such a densely defined landscape was not considered. Instead every square (with an area of 1 square metre) was further subdivided into a number of polygons. This was done by selecting random points on the surface, and marking the lines and points equal distances from those points as edges and vertices - the Voronoi pattern. A better method could have been to simply have those random points be the roots of the grass instead of performing those computations, but as it turned out, Unity is optimised to perform such computations, and keeping track of all of those points on the GPU was a much harder task, requiring detailing with compute buffers. It will be considered for future optimisation, however. This tessellation is performed using the GPU to take advantage of parallel processing.

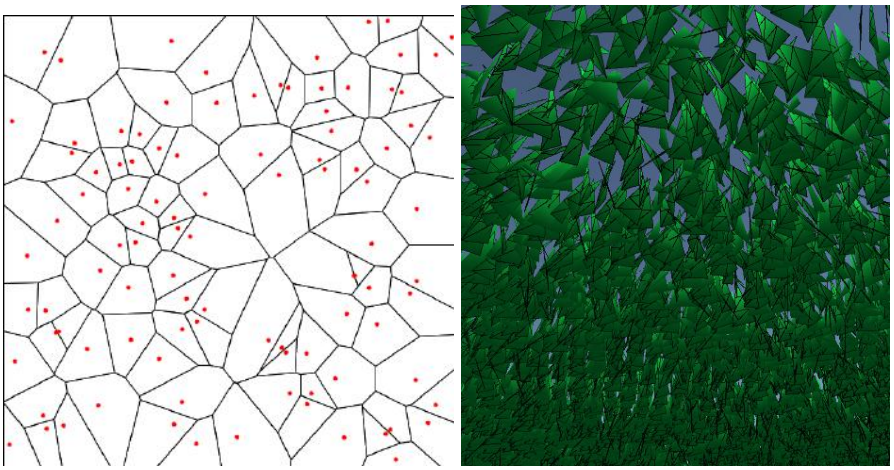


Fig 3.11: Voronoi diagram for tessellation (left) and view of grass from below, revealing the randomisation and constituent polygons (right)

3.2.2.4. GPU Optimisation

Since over 32,000 squares of one square metre area each are present in the landscape, each of which are subdivided into about 300 vertices each, for each of which a new grass blade is created, this kind of computation is simply impossible to perform in a reasonable amount of time using a CPU. Thus, all of this was done in the GPU, taking advantage of the parallel processing capabilities. Unity provides the possibility of using compute shaders, and geometry shaders (a special type of compute shaders) to perform tasks such as this, which was fully taken advantage of. A regular shader was written to perform the vertex and

fragment shading operations, and another geometry shader was written for the tessellation operation. The shaders are written in the HLSL language, customised to be understood by Unity's API to perform higher level tasks than HLSL would normally provide. The resulting aggregate shader is designed to be used with Unity's materials and mesh filters. The fact that mesh filters can render multiple materials at once is taken advantage of by providing only the grass blades in one material, and the land itself in another one.

3.3. Dataset Generation

Dataset Generation is a large part of the project, requiring multiple optimisations. Efficient code was written for actions such as saving files. With three different cameras involved, doing four total rendering passes in each frame. The goals of the Dataset Generation part of the project can be listed as follows:

1. To provide a simple interface that can be used to capture snapshots either individually or automatically every frame
2. To capture various types of inputs, including the grassless view and the procedurally generated grassy view compulsorily, and views such as the depth map and normals map as required, all for any given frame
3. To produce a dataset in a reasonable amount of time
4. To emulate semi-natural movement of the player automatically during the automatic capture of frame-by-frame data

Fig 3.12: Snapshots of datasets generated

3.3.1. Design and Processes

The activity diagram for the dataset generation was created detailing the components and actions involved detailing the components and actions involved during a single frame,and went through multiple iterations of revision. This is it as described in the latest revision:

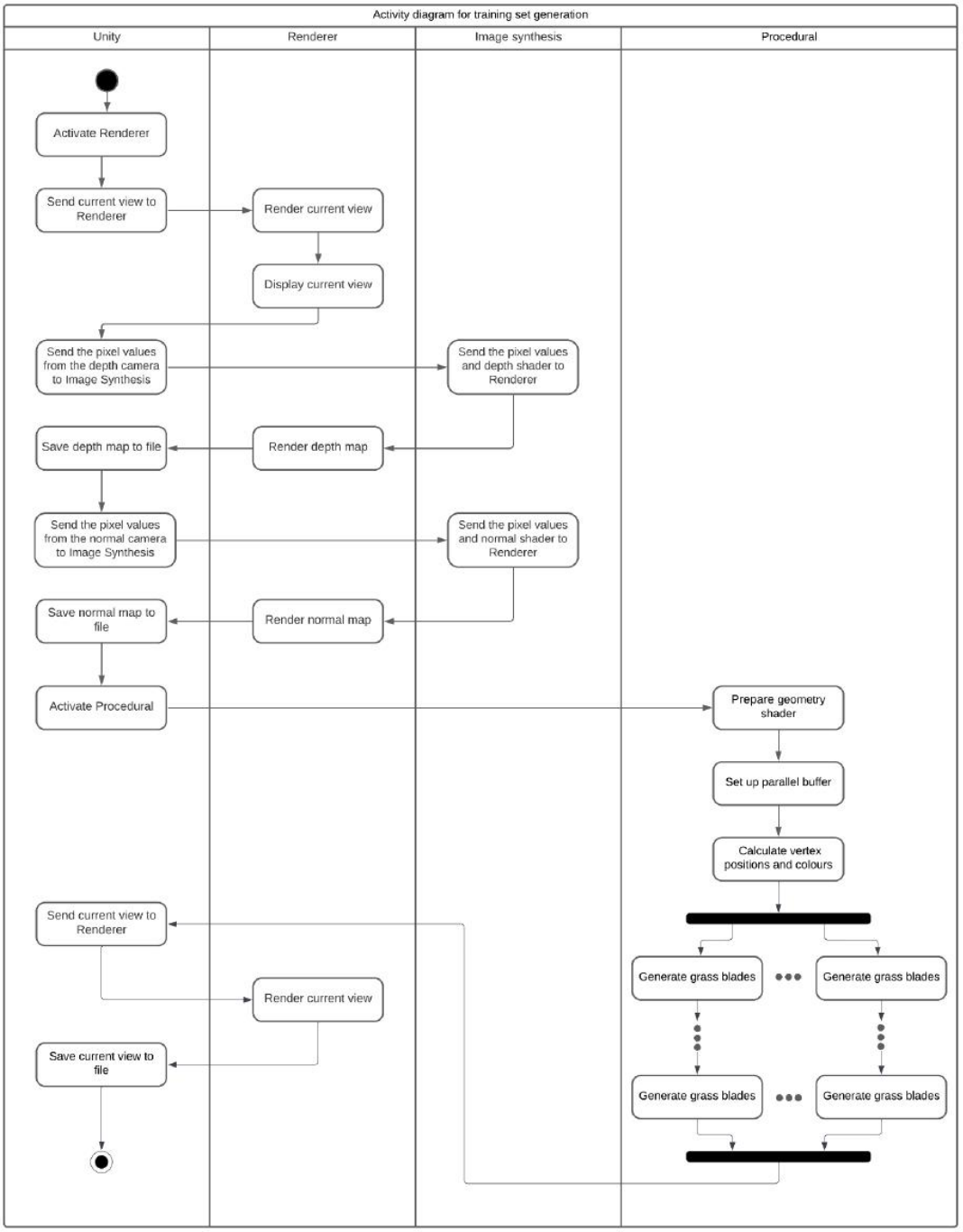


Fig 3.13: Activity Diagram for Training Set Generation for single frame

In a single frame of operation, the camera renders the scene four times, each time with a different replacement shader, and saves the produced output to disk.

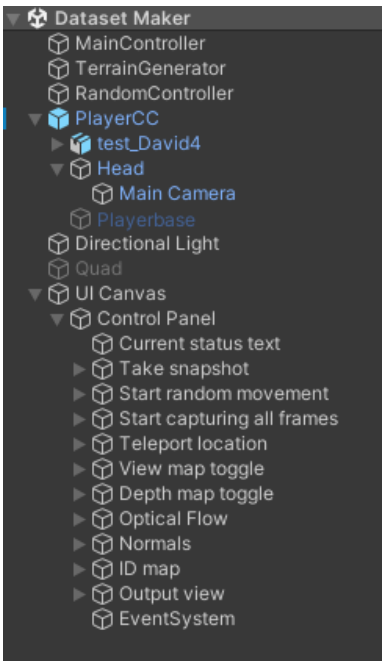


Fig 3.14: Structure of game objects in the dataset generator scene

3.3.2. Random Movement

The datasets were generated automatically, with no active involvement of the user during the generation. This was required to be able to run the generator for a long period of time, in case that is required by the Research Development experiments. This was only possible by having an agent emulate natural movement of the character during normal gameplay.

This movement was performed by using coroutines and timers to randomly virtually trigger the control events - in other words, virtually press the arrow keys to move, space-bar to jump, move the mouse to look around, etc.

The duration of these timers was fine tuned to best emulate natural gameplay. A future change to this could be using a neural network model to learn to do this more naturally, but this method was found to be perfectly adequate for our purposes.

3.3.3. Performance

The fastest time achieved to create the dataset turned out to be about 5 frames per second. Many datasets were created for the project, as per the requirement analysis phase of the research development, among which four were the biggest and most important, which shall be described in more detail in the final documentation of the project. Each of those biggest datasets required about two hours to create, which in the end easily fit the reasonable time requirements.

3.3.4. Major Datasets Created

As per the requirement analyses and insights gained from the experimentation (as described in the appendix section of this documentation), we made five major datasets in the duration of this project.

Dataset version 1: textured inputs, with depth maps, optical flow maps, and object ID maps
Dataset version 2: textured inputs, with depth maps; removed character from view to only have the grassy landscape
Dataset version 3: textured inputs, with depth maps and normals maps
Dataset version 4: smooth inputs, with depth maps and normals maps
Dataset version 5: noisy inputs, with depth maps and normals maps

The current dataset consists of these four images for each training item: the input, the depth, the normals map, and the ideal output (the procedural grass).

A CSV file mentions the paths to all of these images, as well as some important vector data such as the speed and velocity of the camera, and the lighting direction.

An automatic random movement system was created to let the camera randomly navigate around the environment while taking pictures five times every second.

3.4. Neural Renderer Playground

The Neural Renderer Playground is simply the mock game where natural gameplay can be performed, but it is capable of running a neural network taking inputs from the game and displaying it into a quad view, as seen in *figure 3.4 in section 3.1.1 - Game Environment and Interface Design*

Two activity diagrams were created for the design of this playground. The initial setup, and the operations performed in a single frame.

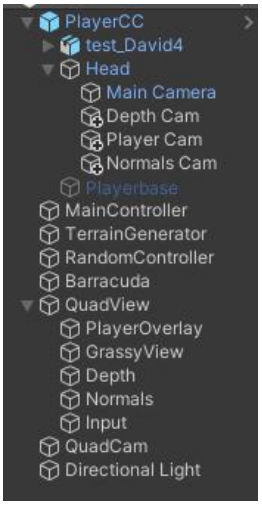


Fig: Game Objects in the Neural Renderer Playground scene

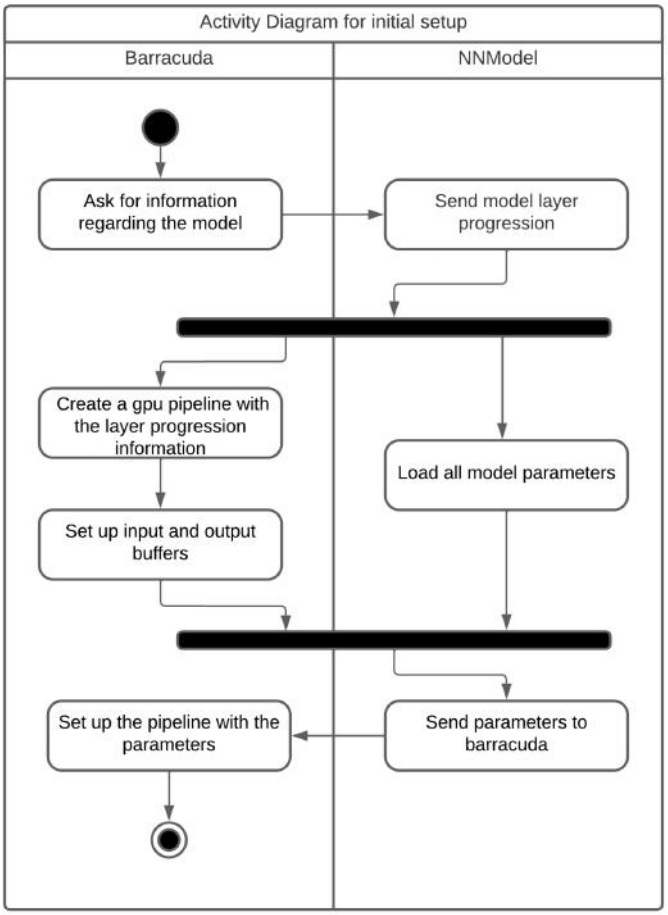


Fig 3.15: Initial setup of the Neural Rendering Playground

Fig 3.16: Operations in a single frame of the neural rendering playground

The playground made use of Barracuda to run the neural rendering model in the GPU. Render textures were used to provide the inputs to it, and receive the output. Both of these are described in sections *3.3 - Dataset Generation* and *3.4 - Neural Renderer Playground*

The neural rendering model was only fully completed halfway through the development of the project, due to difficulties understanding and working with the GPU, and the lack of good documentation resources for the libraries used, with them being quite new libraries. Despite that, it proved to be an extremely useful tool to test the real-time performance, and external-data performance of the neural rendering models used in experiments.

3.5. Chapter Summary

The 3D environment setup for dataset generation can be divided into three parts: Terrain, Procedural grass, and corresponding image map generations. Terrains are generated in Unity using 2D Perlin noise for the vertical displacement to form landscapes, with only tiles around the player updated using a coroutine. Procedural grass is generated by chunking planes into smaller divisions using Voronoi tessellations with every vertex treated as seed points for grass blades. Image maps, although ideally extracted from rendering pipelines, required using camera replacement shaders due to access restrictions. Potential areas for optimization exist that we aim to explore in the future to achieve higher-quality real-time performances.

Iterative fine-tuning has been done to generate good quality datasets from initially 10,000 images, to then generated 20,000 images, to a few more versions, and finally 8192 items in our dataset, as the requirements changed over time through performing experiments. Extensive experimentation was done to investigate the image maps that best served as the input, like segmentation maps, optical flow maps, normals maps, and depth maps. In the end, normals and depth maps were found to be the most useful input for our model. Similarly, it took several trials to figure out which person-view is best received by our model and we finally switched to a third-person camera view with an animated character for natural movement only in the forward direction and rotation of a game player simulated using two components, player and head.

The final dataset generated has a size of 8192 items, containing the view map with added noise, depth map, and normals map, along with some vector data, including positions, velocities, and lighting direction. The output of the neural rendering model is displayed on the screen, along with a split-screen view of all the maps mentioned and live windows to the vector values. All datasets were generated in reasonable amounts of time, which was a factor that itself required lots of optimisation. It is working fast and completely as intended.

The Neural Renderer Playground includes a rendering pipeline capable of providing the input and getting the output out of a neural network model, as well as optimising its processing in the GPU. This tool was invaluable for testing the models produced during the experiments.

CHAPTER 4

RESEARCH DEVELOPMENT

4.1.1. Objectives

The objectives of the research development part of the project are as follows:

- To train the machine learning model, fine-tune the hyperparameters and find the suitable objective function for our network.
- To design the machine learning models for the neural rendering of grass.
- To experiment with the state of the art architectures and perform inference based on the observed data.
- Perform quantitative and qualitative analysis of the observed data for performing various hypothesis testing.
- To create a list of requirements for the environment dataset generation for further improvements.
- To provide feedback to the environment regarding the transformations used, neural rendering model format, input format, etc.

4.1.2. Methodology

We used torch vision utils to save image in the Google Drive.

Tools used:
Platform: Google Colab, Kaggle
Framework: PyTorch, PyTorch Lightning

We conducted a majority of our research and development on Google Colab, utilising its free GPU for training. However, due to occasional usage limitations, we also conducted some of our model training on Kaggle, which provided us with 30 hours of free GPU access per week. Initially, we attempted to load our data directly from Google Drive, but we found that the process was seven times slower compared to loading the data from Colab. We also experimented with putting the training images directly in the RAM and loading from there, which was about 1.5 times faster than having it in the colab context storage but due to RAM limitation we were not able to put all the images in the RAM.

For training we utilised the PyTorch Lightning framework, which provides a high level interface for PyTorch. Using the feature provided by PyTorch Lightning we were able to save all our progress(checkpoints) in the specified path on Google Drive. As we were

experimenting with numerous models and hyper-parameters, we created the following directory tree to keep track of all the changes and their respectives outputs.

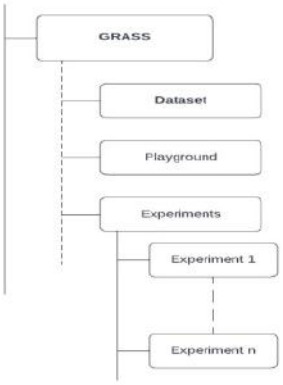


Fig 4.1: Directory tree tracking the progress

Dataset folder was used to store our dataset. All the experiments with each having a unique name were stored in the experiments folder. Similarly, a Config dictionary of hyper-parameters provided us an efficient way of storing all the hyper-parameters used in the program.

4.2. Training Environment

4.2.1. The System

Path of the generated data using the Unity game engine is written in the csv file. This csv file is processed using the pandas library in python to create the dataset. The preprocessed dataset is then given to the training pipeline which uses the supervised learning.

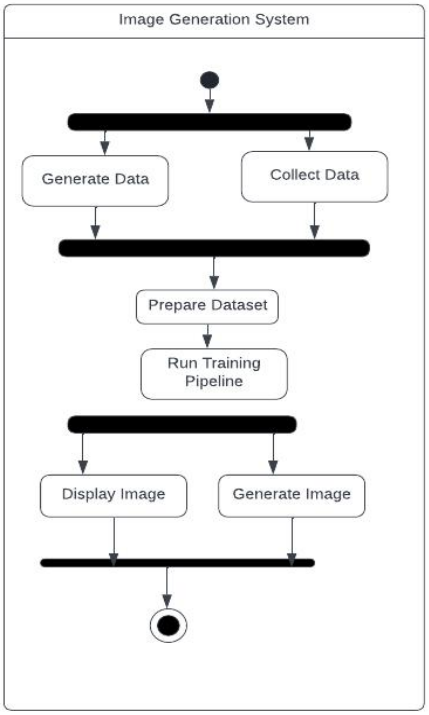


Fig 4.2: Activity Diagram of the Image Generation System

4.2.2. Training Model

We are using GAN (Generative Adversarial Network) to train our machine learning model. This model consists of two networks i.e Generator and Discriminator which work on zero sum rule.

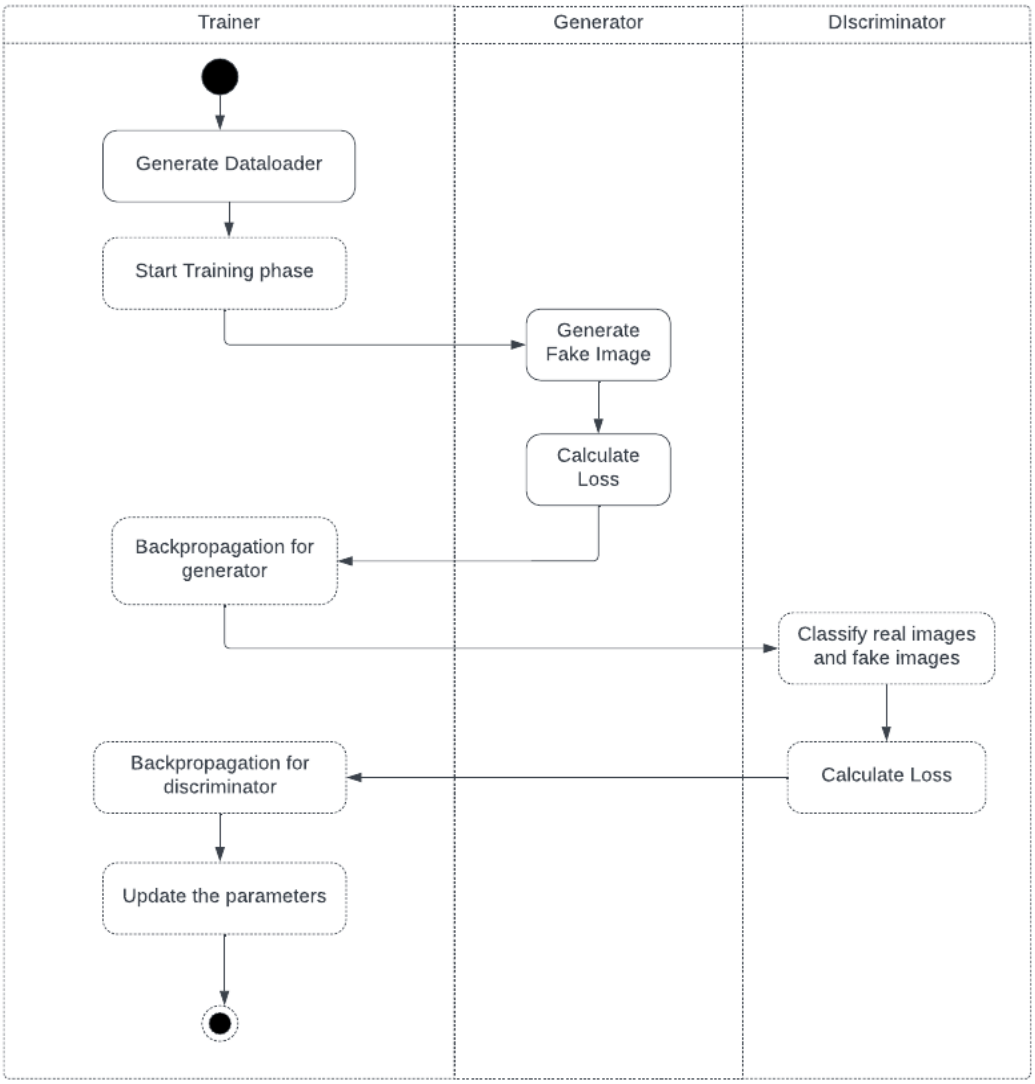


Fig 4.3: Activity Diagram of the training model

The proposed system at minimum consists of four classes. The trainer class is responsible for the entire training loop. The Generator and the discriminator networks are optimised based on the loss value obtained from loss class.

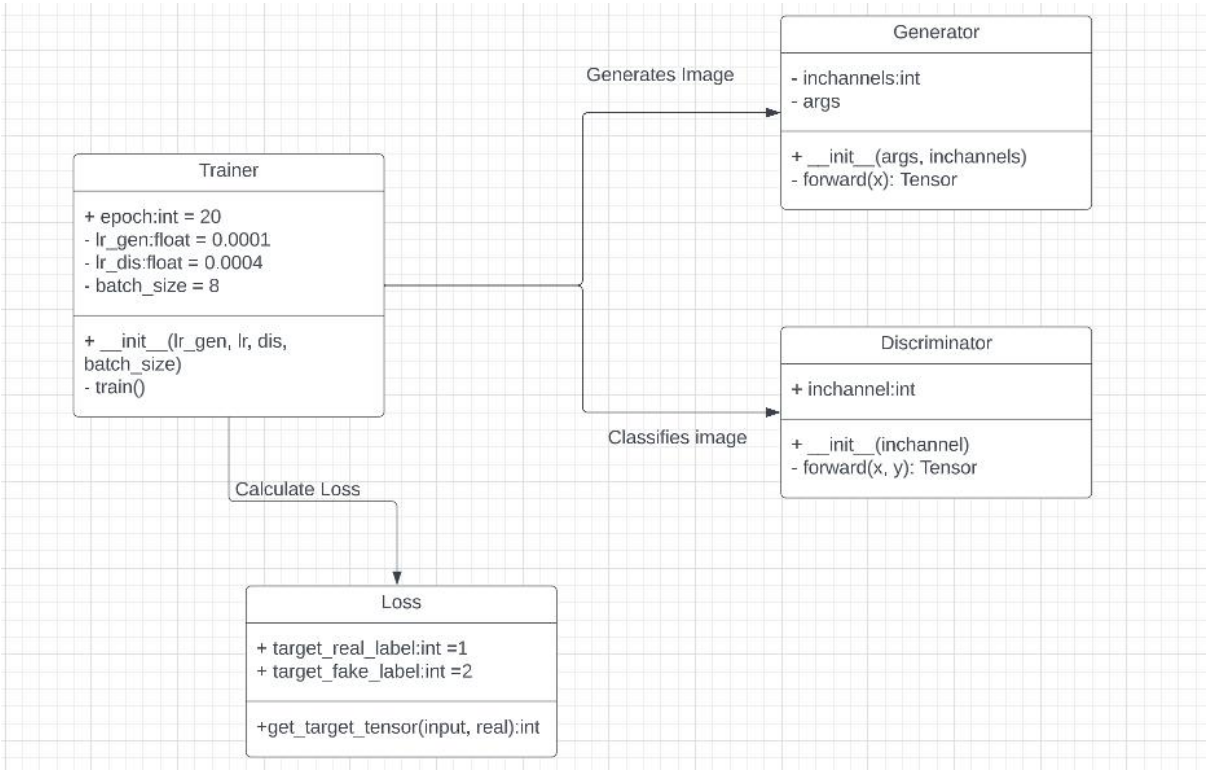


Fig 4.4: UML class diagram for GAN

4.3. Development Map and Explored Ideas

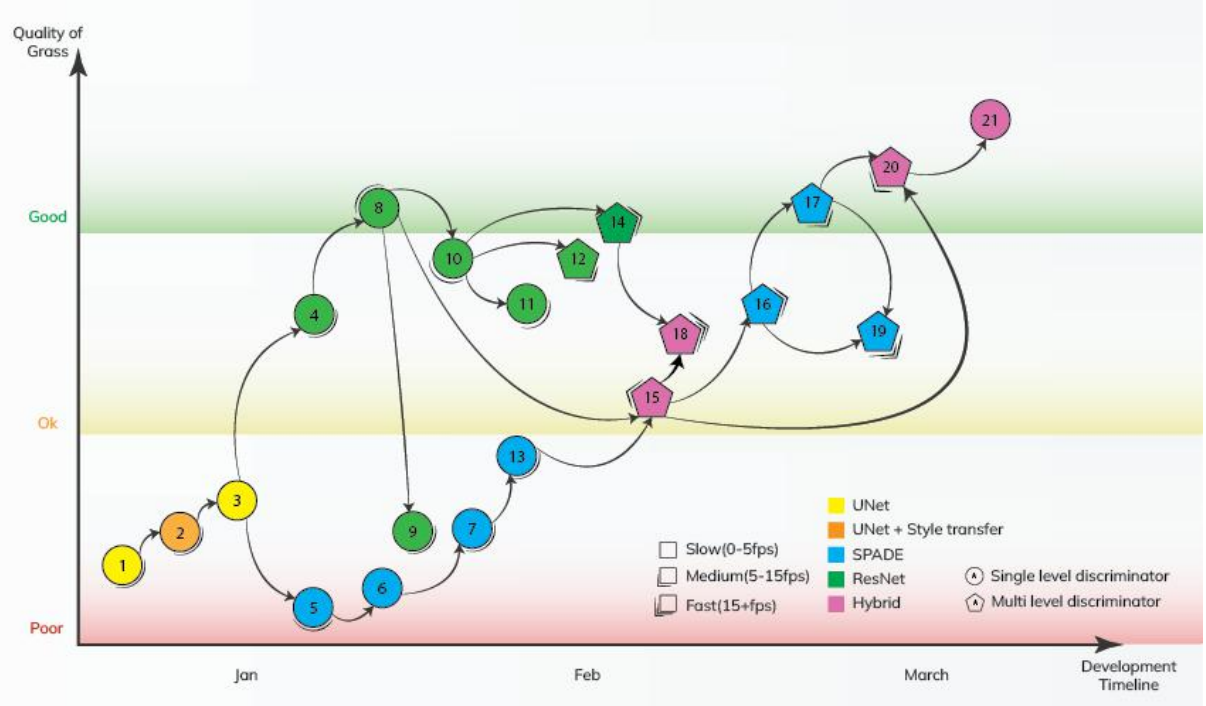


Fig 4.5: Progression of the research development work

We opted to use the U-NET architecture as our baseline model due to its simplicity, efficiency, and accuracy. However, we found that the U-NET architecture alone was insufficient in learning about the grass.

To address this, we combined the U-NET architecture with a style-transfer network, but the results were still unsatisfactory. As a result, we conducted further research and discovered SPADE (Spatially Adaptive Normalisation)

As the approach looked promising we decided to work parallelly, one team working on SPADE based decoder architectures while the other team working on regular ResNet based decoder-encoder architectures.

The quality of images produced from ResNet architectures were far better in comparison to SPADE. However, the speed of learning in SPADE appeared to be much higher. Seeing the advantage of both ResNet and SPADE, we decided to make a hybrid model. The hybrid model used Half-SPADE and produced reasonably good results in terms of quality. To further improve the performance, we experimented with using a Multi-level PatchGAN as a discriminator, which showed better results in terms of both image quality and speed.

4.4. Generative Adversarial Network (GAN)

GANs are a type of machine learning model that uses neural networks in a zero-sum game format. This involves a generator neural network that creates output intended to deceive a discriminator neural network, which aims to accurately classify the generated output. GANs are the new state-of-the-art neural network that can generate realistic looking output.

We performed an experiment using the GAN model and a simple UNet architecture. The results showed us that without a discriminator the model was not able to learn the nature of the grass blades. Nevertheless, it was able to learn the colours and generated basic blank green images of grass as a result (*see Appendix section 7.1.9 - Experiment set 9 - codename: Oyster)*

One problem that we often encountered across a number of models that we trained was mode collapse. Mode collapse is a condition when the generator produces similar kinds of images. This happens when the generator learns that it can fool the discriminator using a specific type of data and thus keeps on generating that kind of data as it has no incentive to vary the output.

4.5. Experimented Models and Outcomes

4.5.1. Generators

4.5.1.1. Conv blocks

Upsample conv block:

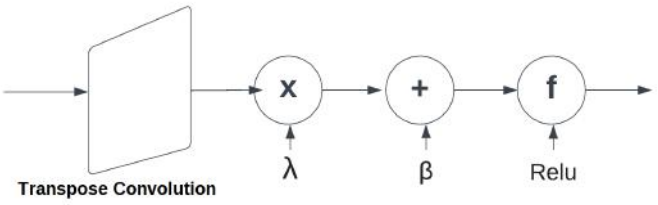


Fig 4.6: Up-sample block

We performed the upscaling of the input data using transposed convolution operation with the specified number of kernel size, stride, and padding. During the operation we also changed the number of outchannels.

Downsample conv block:

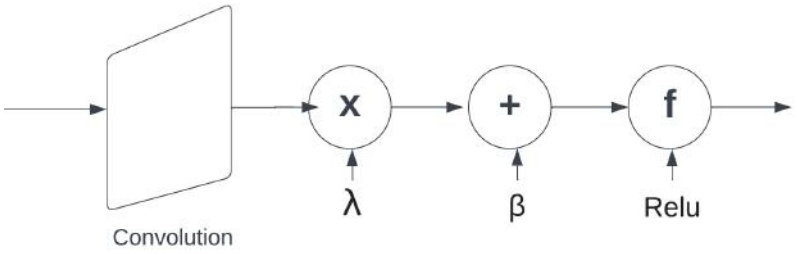


Fig 4.7: down-sampling convolution block

We downscaled the input image using a 2D convolution over an input image composed of multiple planes. The factor for down scaling was specified using the kernel, stride, and padding size. We also perform batch normalisation based on the passed argument.

4.5.1.2. The UNet Architecture

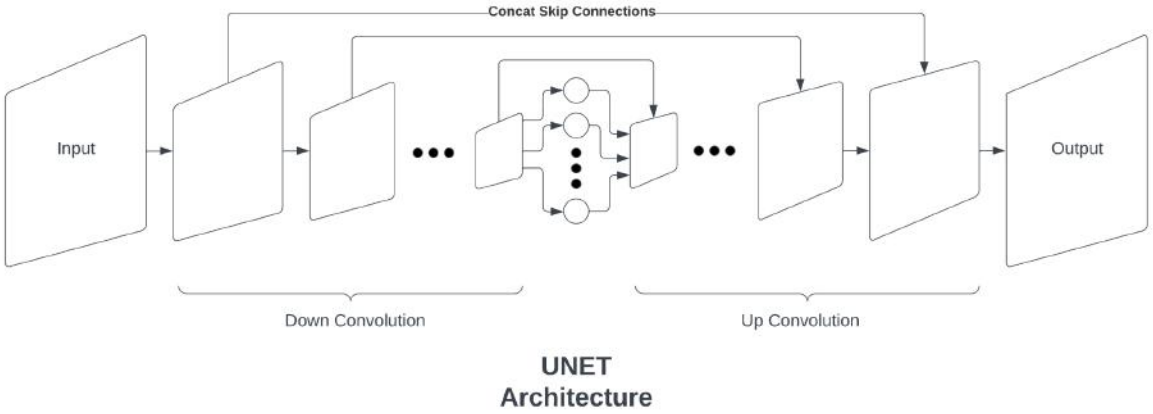


Fig 4.5: UNet Architecture

We started with the UNet architecture, consisting of shrinking convolutions using Down-Sample Block followed by growing convolutions using Up-Sample Block, with skip connections connecting corresponding sized layers. Skip connections ensure the feature reusability. A fully connected layer sits in the middle.

We chose this architecture since it is a standard default one, and we had to start somewhere. It did not produce very good results, and caused repetitive blocks.We conducted tests where we varied the hyper-parameters and number of parameters in the network. We found that

increasing the number of parameters improved the accuracy of the model, but it also resulted in longer processing times for the output.

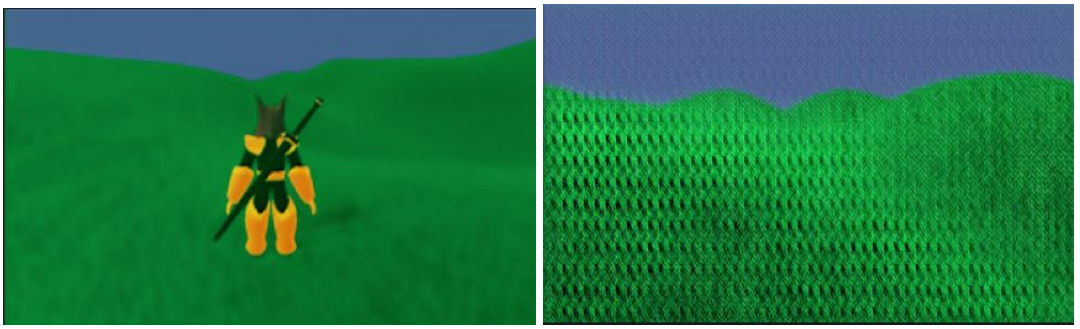


Fig 4.6: Images generated with U-NET architecture

4.5.1.3. The Style Transfer Network

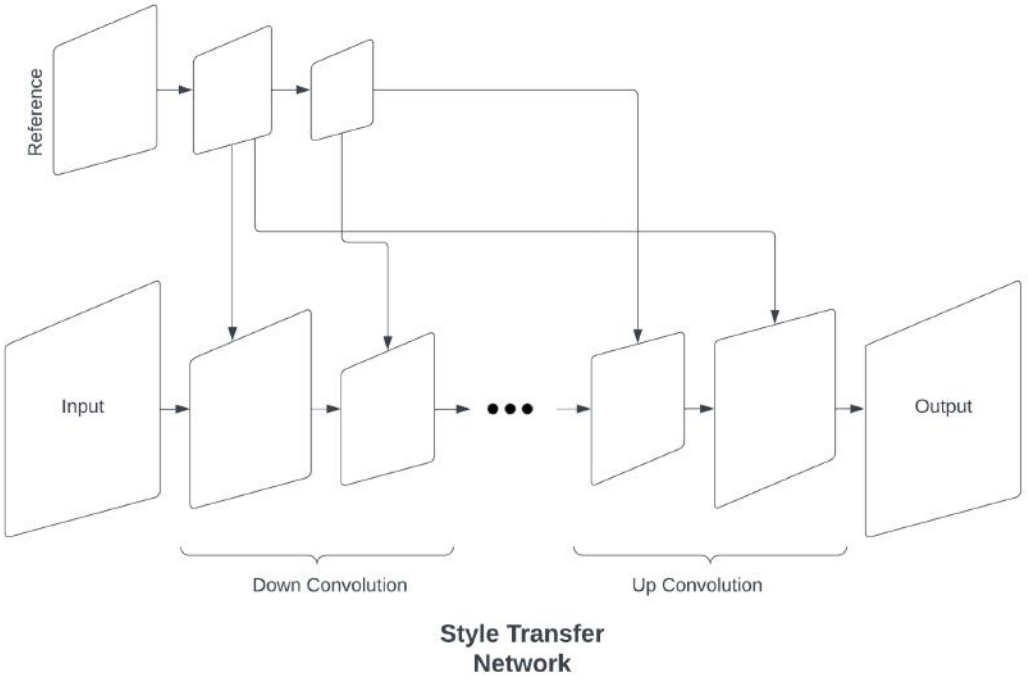


Fig 4.7: Style Transfer Network

Style transfer network applies the style of the reference image to the content of the input image. We figured providing 16 reference images of the procedural grass to the UNet using style transfer techniques would help, and we experimented with a lot of ways to add that data to the UNet layers. It produced a static, overlaid type of effect that wouldn't move with the camera and would appear in the sky, which is not acceptable. Here is an example of such a result:

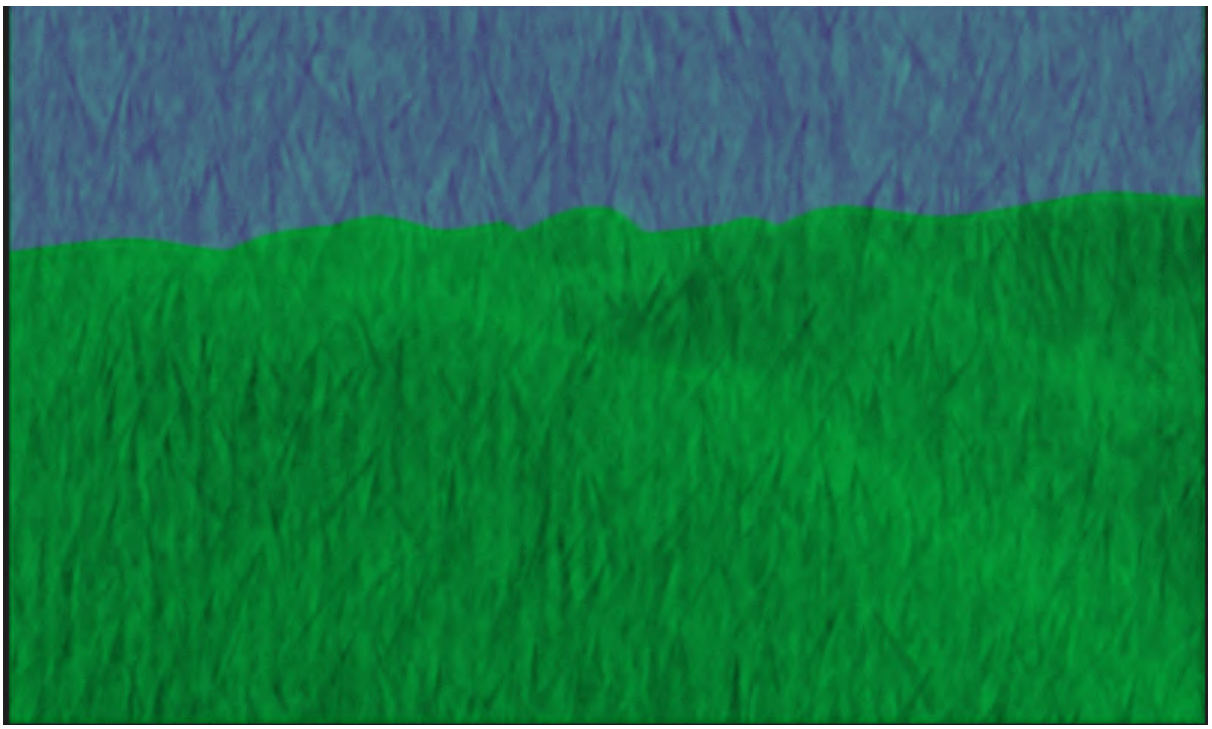


Fig 4.8: Image produced by U-NET Style Transfer Architecture

We then split into two teams, one to explore ResNets and other to explore SPADE.

4.5.1.4. The ResNet Array

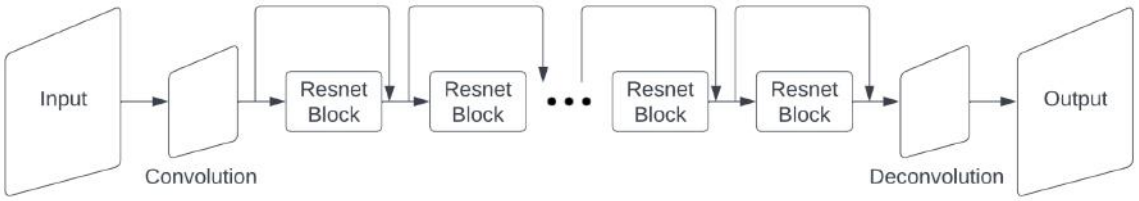


Fig. The ResNet array network

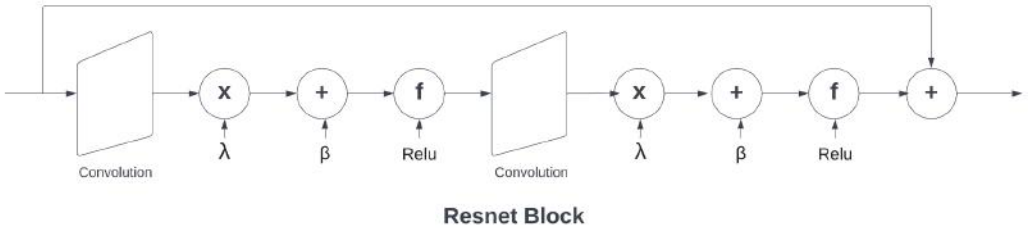


Fig 4.9: ResNet array architecture

This architecture introduced a much more elegant way to introduce skip connections than a U-NET, and it worked much better than previous models, producing wonderful grass images such as this:

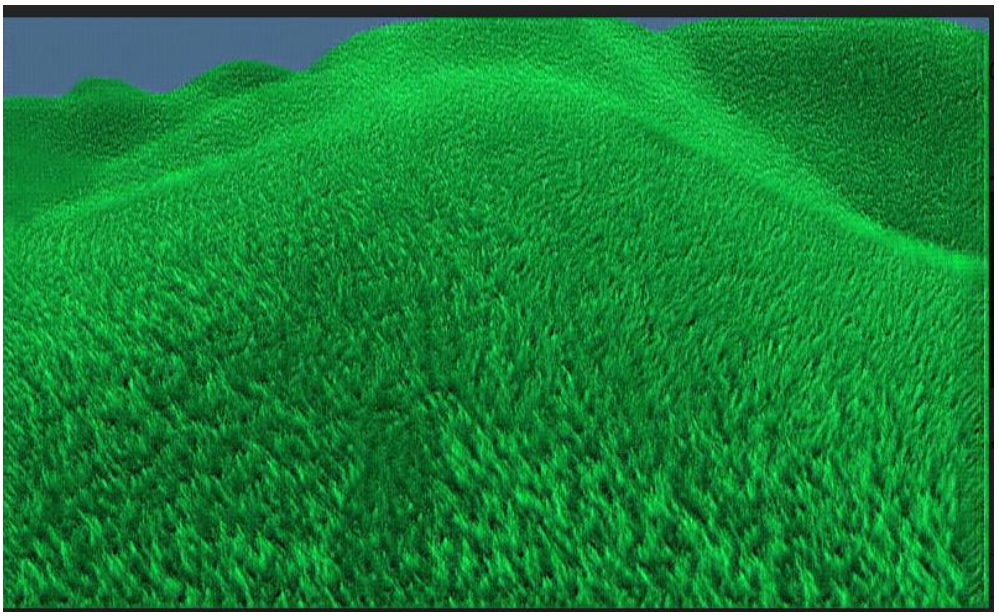


Fig 4.10: Grass generated by using ResNet blocks

We also explored various combinations of this model, and this is as good as it gets. It required over a 100 epochs of training on a 2000 item dataset, followed by 10 epochs on the full dataset.

4.5.1.5. The Hybrid Model

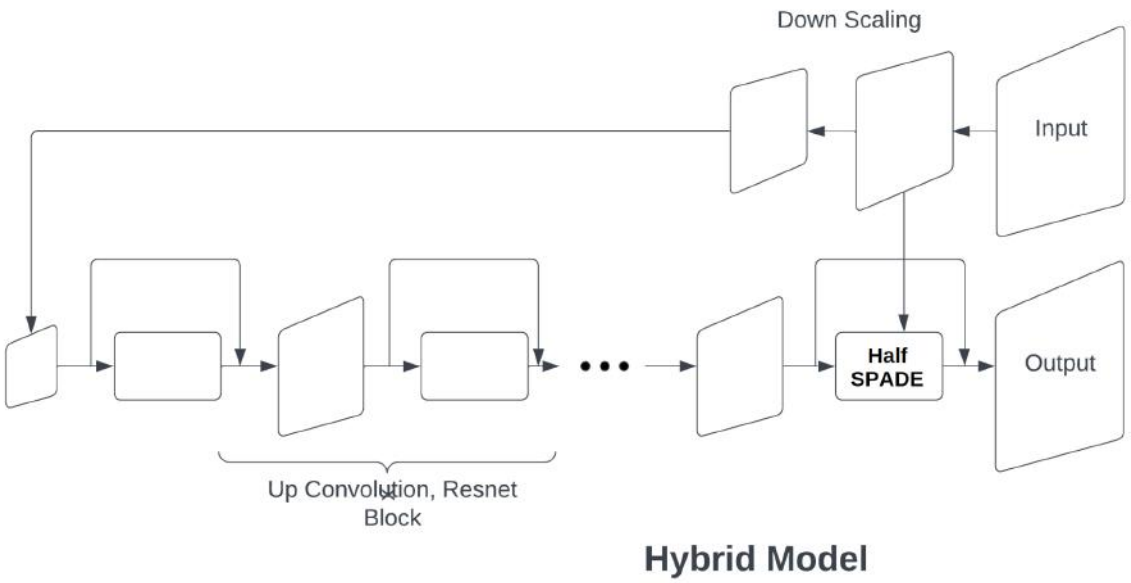


Fig 4.11: Hybrid Model architecture

This model designed entirely by us produced the fastest learning of grass, producing results like these within 24 epochs (which would've required the ResNet array over 80), using only about 700K parameters, in contrast to a few million.To process the input image, we first reduced its size and then fed it into the network. The network consisted of several up-convolution layers and ResNet blocks.

We used Half-SPADE before the final upsampling of the image. Half-SPADE is simply a ResNet block where one of the normalisation layers is replaced with the with SPADE normalisation. The code for this is written simply and within the ResNet block class, unlike the elaborate code for SPADE used in the SPADE model *(See section 4.5.1.6 - SPADE)*. We experimented with different positions for the SPADE normalisation and found that SPADE normalisation in the first convolution learned faster and picked up the nature of the grass faster.

The results of this model were really encouraging, and we trained using both a dataset of 2000 images and a complete dataset.

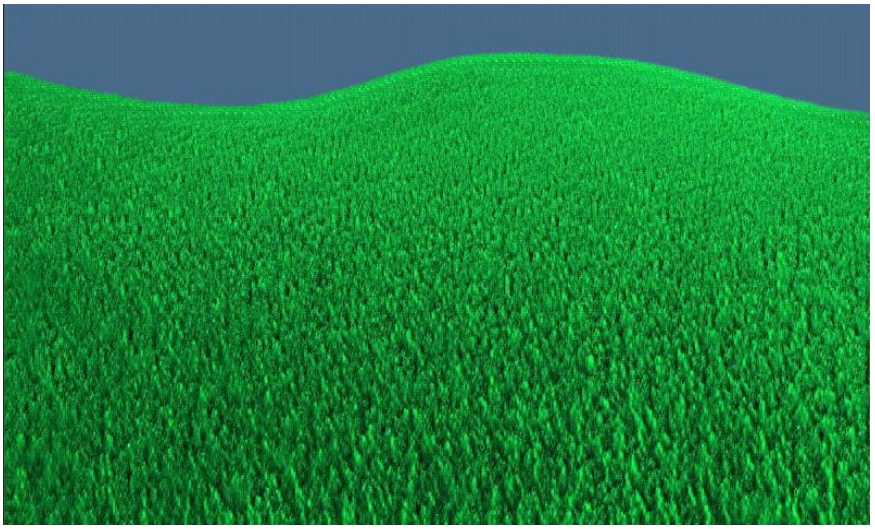


Fig 4.12: Image generated by the Hybrid Model

4.5.1.6. SPADE (Spatially Adaptive Normalisation) model

Spatially Adaptive Normalisation(SPADE) is a conditional normalisation which was originally developed for converting the semantic segmentation mask into a photorealistic image. Unlike other popular normalisation methods like Instance Normalisation(IN) and Weight Normalisation (WI) which do not depend on external data, normalisation in SPADE depends on the conditioned input. It consists of the learning parameters and γ which depend on the input image and vary with respect to the location. In the original paper by Park et. al [9], SPADE normalisation was performed in the SPADE RESBLK as shown in the image below. The activation in the SPADE is performed similar to the batch normalisation in channel-wise manner.

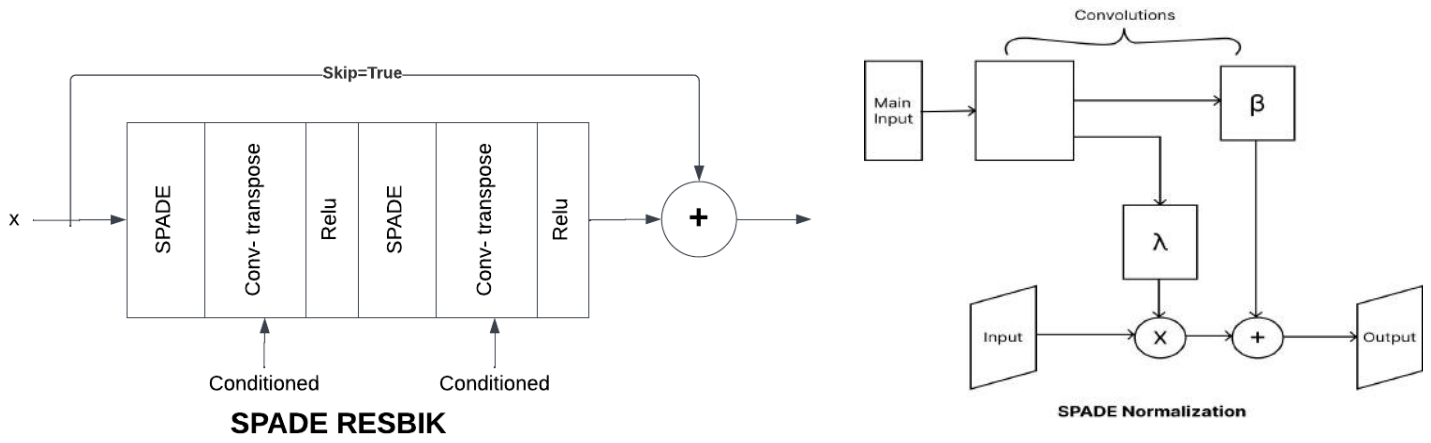


Fig 4.13: SPADE RESBlock and normalisation

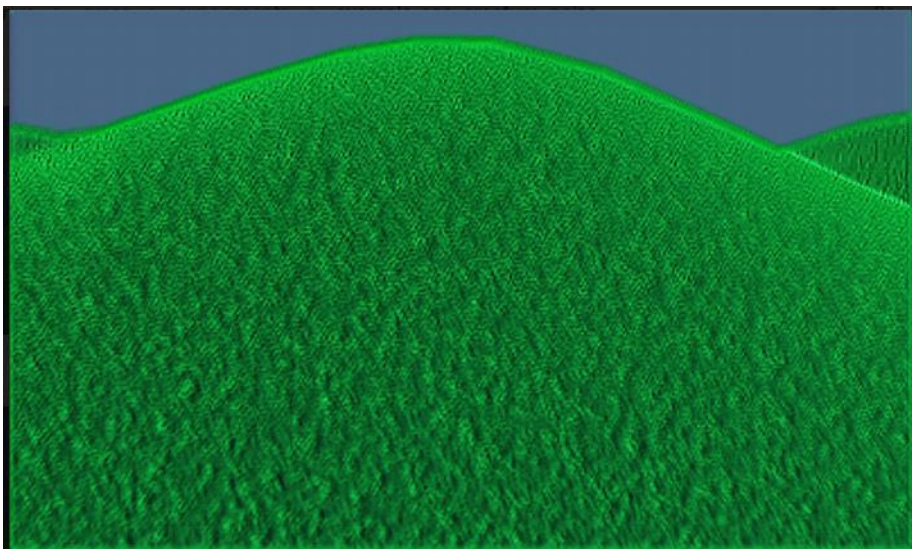


Fig 4.14: Output of the SPADE Network

4.5.2. Discriminators

4.5.2.1. The simple PatchGAN

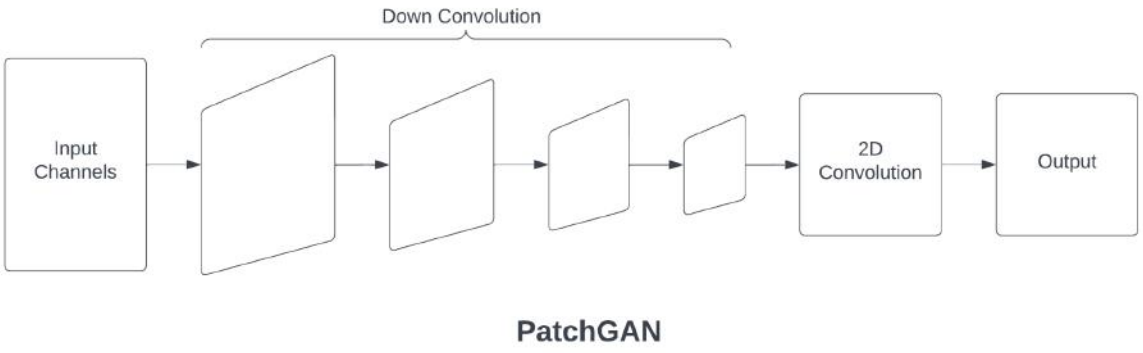


Fig 4.15: PatchGAN architecture

PatchGAN is a type of discriminator in GAN where each value in the output matrix represents the probability of that image patch being fake. The discriminator runs convolutionally across an image, averaging all the results to produce an output matrix.
We used different iterations of this for a large number of our experiments, and turned out good enough. However, it was a little unstable, and trained slowly if it was to be kept from overfitting and providing useless feedback.

4.5.2.2. The Multilevel PatchGAN

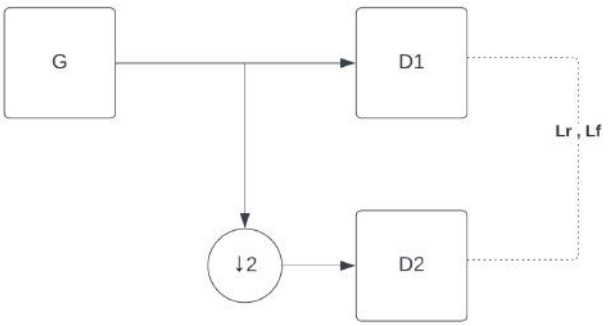


Fig 4.16: Multi-level PatchGAN architecture

We simply used multiple patchGAN on different scaled versions of the inputs/outputs, and this produced overall more stable results with faster training. The purpose for our research with multi-level PatchGANs was to test our hypothesis whether sending the normal and half-downscaled image to two different identical discriminators would pick up on details of the grass at varying levels, providing more depth to the quality of our grass generations. The final adversarial losses obtained from these patchGANs were weighted and added, and the

final real/fake losses were averaged before returning the final loss function of the discriminator. The resulting graph functions showed a strong improvement in our generations initially, but later resorted to some stagnation.

4.5.3. Loss Functions

Loss functions are a method of evaluating how well a neural network has learned to map the given input data to the actual output.

1. **Loss Types:**

   BCEWithLogitsLoss
   It is a combination of BCE(Binary Cross Entropy) loss and Sigmoid layer in a single class. Combination of both functions in a single class produces a more numerically stable result.
   The loss function is given by:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right]$$

   L1 Loss
   It measures the absolute mean error(MSE) between each element in the input x and target y .
   The loss function is given by:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = |x_n - y_n|$$

Where x and y are of arbitrary length with n elements and N batch size.

2. **Adversarial Loss**
   Adversarial loss is calculated based on the probabilities given by the output of the discriminator. We used BCEWithLogitsLoss as the adversarial criterion. It consists of two parts: Generator loss and Adversarial loss. Generator loss is used to measure how well the generator is able to fool the discriminator and Discriminator loss measures how well the discriminator can classify the fake images generated by the generator.

3. **Reconstruction loss**
   Reconstruction loss measures how well the generator is able to generate the output data given the generated data. It performs the comparison between the generated image and the original image. We used L1 Loss as the reconstruction loss.
   By incorporating the reconstruction loss, the generator is encouraged to not only generate images that look like the actual image but also to capture the underlying structure and patterns.

The reconstruction loss is added to the adversarial loss to generate the total loss and is given by:

*Total Loss (Generator) = Adversarial loss + (lambda * reconstruction loss)*

Here, lambda is a hyper-parameter which defines the impact that reconstruction loss has on the network relative to the adversarial loss. We experimented with different values of lambda and found that decreasing the value of lambda results in the mode collapse and higher value of lambda results in slow learning of the discriminator.

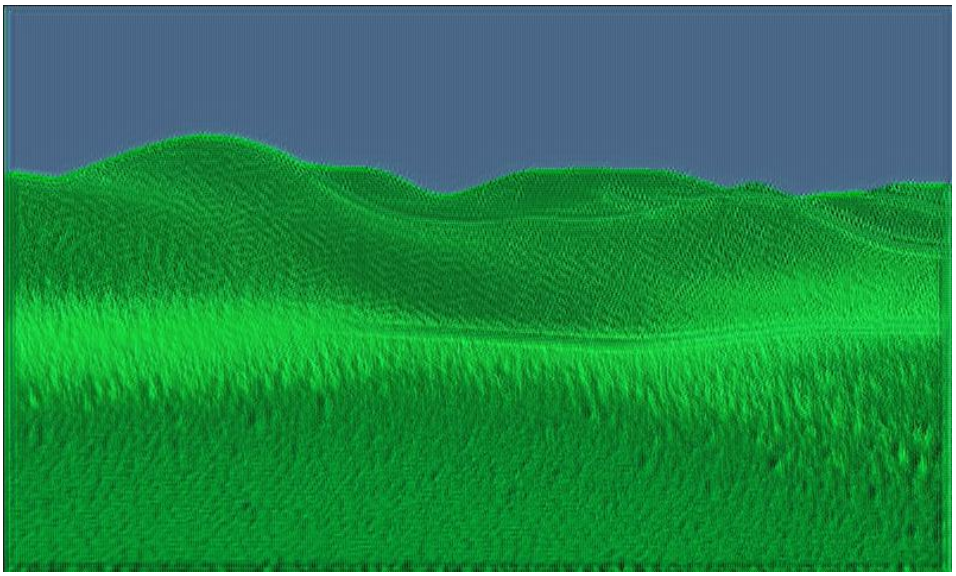


Fig 4.17: Image generated with low lambda value

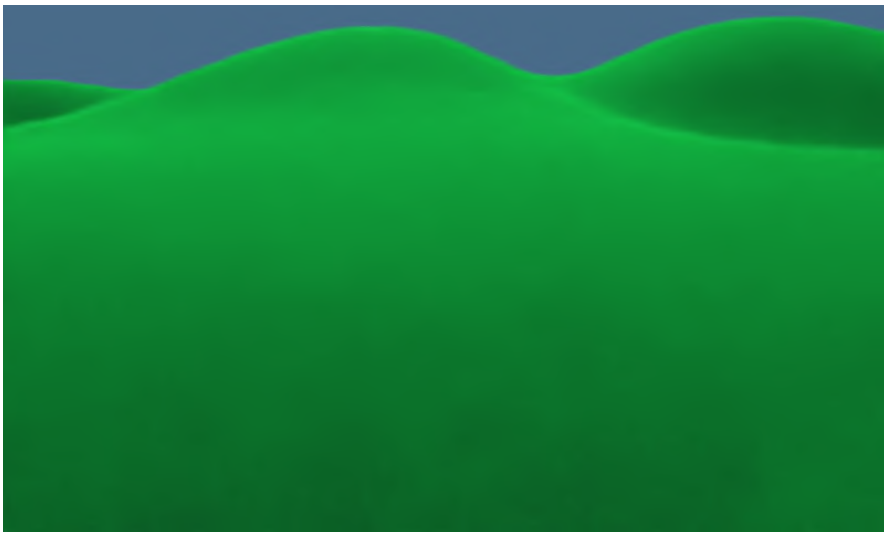


Fig 4.18: Image generated with high lambda value

4.5.4. Optimizers

Optimization is a process of changing the attributes of a neural network which are often termed as learning parameters like weights and biases to reduce the loss. We experimented with the following optimizers:

1. **Adagrad**
   It is an extension of stochastic gradient descent which has a variable learning rate on a per-parameter basis which gets updated based on how frequently the parameters get updated. Higher the update for the parameters, lower is the update for the learning rate.
   It is given by:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

2. **Adam:**
   Adam, a variation of stochastic gradient descent, incorporates the benefits of two other extensions of stochastic gradient descent - Adagrad and RMSProp. It adjusts the learning rate using the first moment (the mean) and second moment (the variance). Our findings indicate that Adam is the most suitable optimizer for our particular task.

3. **Nadam:**
   Nesterov-accelerated Adaptive Moment Estimation (Nadam) is an extension of Adam optimizer that utilises Nesterov momentum to perform better optimisation.

4.6. Chapter Summary

The core objective of this chapter was to design and train multiple neural network models and experiment with their architectures and inputs so that they may serve as feedback to the environment development. We used PyTorch and PyTorch Lightning on Google Colab and Kaggle to conduct these experiments. Our preprocessed dataset is given to the training pipeline, which then feeds this to our GAN model.

The generators and discriminator networks are optimised based on loss value. Our static initial generator design consists of a UNet architecture and some other major models that we explored were Style Transfer networks and SPADE. This conclusively led to our final model being a hybrid architecture. Similarly, our initial design for discriminators was a patchGAN; we later found the performance of multi-level patchGANs taking different scaled versions of input comparatively better. One major problem that we often encountered across a number of models that we trained was mode collapse. We also experimented with different loss types

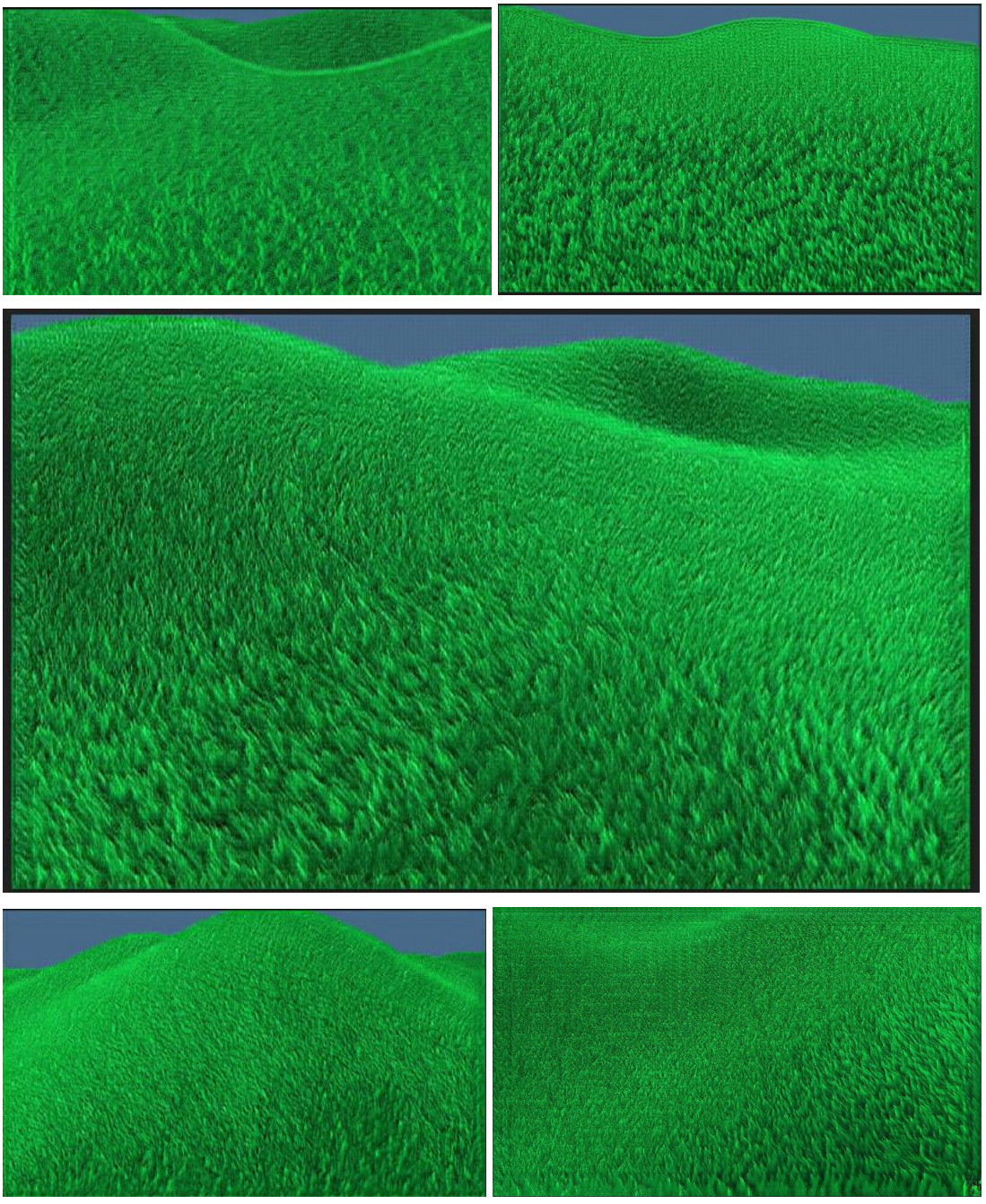
and optimizers but the models weren’t trained enough yet for these to provide strong, conclusive results.

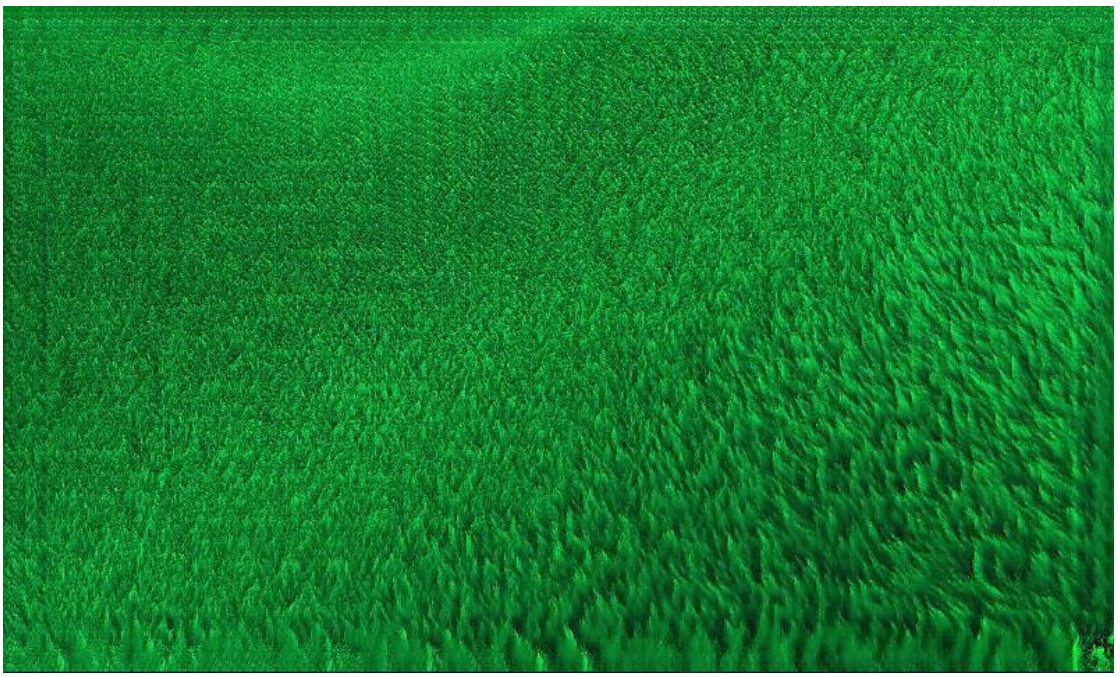
CHAPTER 5

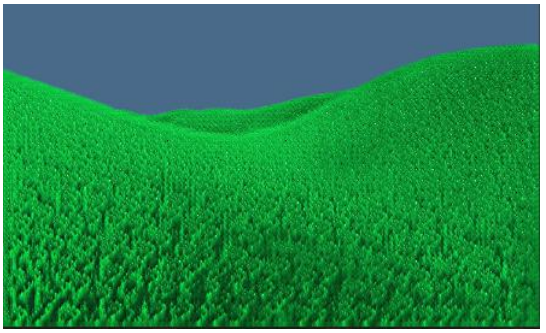
RESULTS AND CONCLUSION

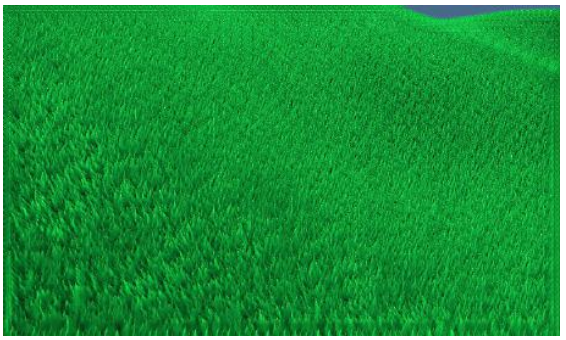
5.1. Gallery of generated images

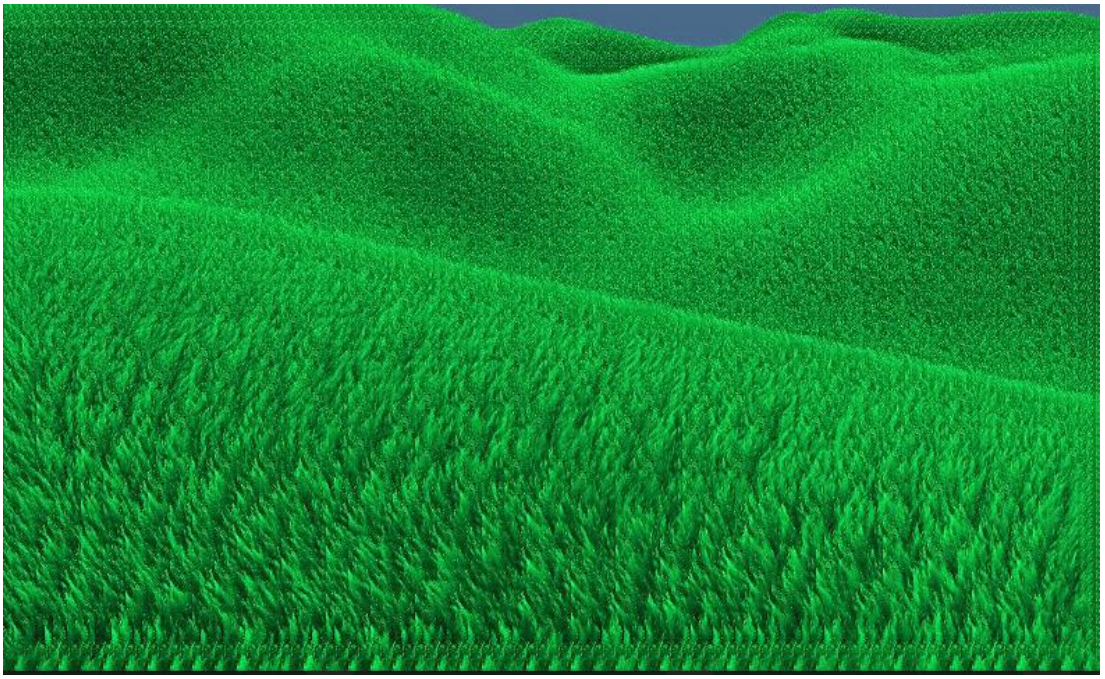
Here is a gallery of some of the best grass generated by our various models and their fine-tuning over time.











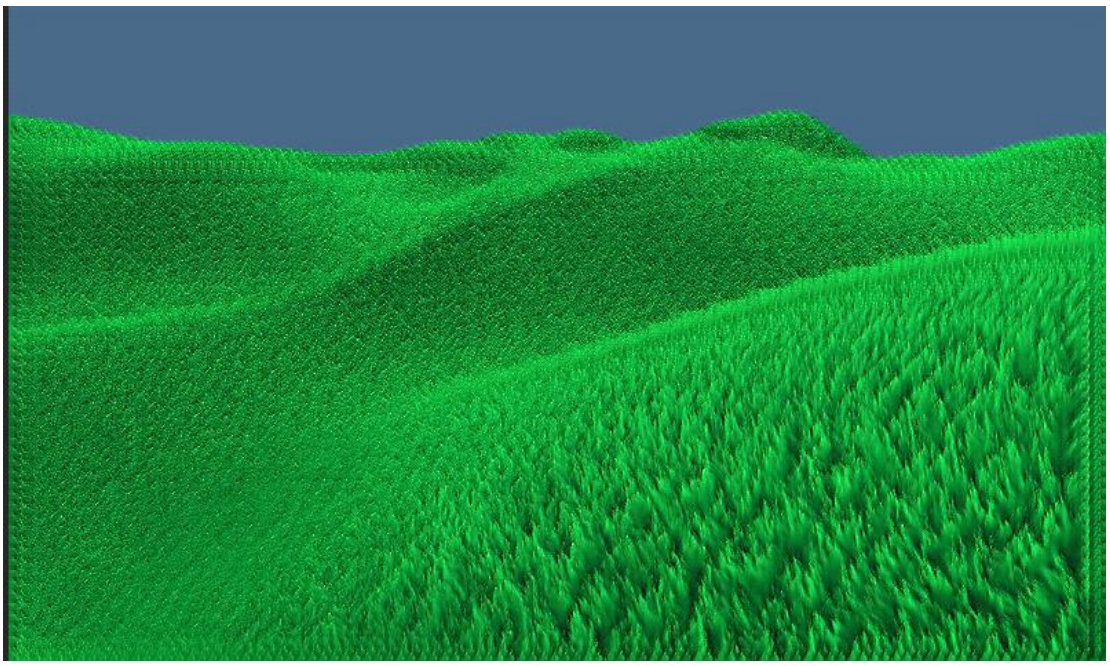


Fig 5.1: Glimpses of best grass generated by our models

Some of the models we used along with their design process and results are described in the Appendix section of this report. Further details about all models used and the experiment process will be described in the final documentation/thesis of the project.

5.2. Performance

We gauge the performance of our models on the basis of the quality of generated grass and the rate of these frames generated per second. Our final model generated one of the sharpest images, but experiment set 20 (see section *7.1.20 - Experiment set 20 - codename: Kraken*) had one of the best overall performances in terms of both of the aforementioned aspects at an average speed of 10 fps. Some of our experiment generation speeds peaked up to 15+ fps, for example, experiment set 19 (see section *7.1.19- Experiment set 19 - codename: Dolphin*), but despite having promising initial generations, these later were affected by a lot of stagnation and fluctuation, due to which good quality grass couldn't be attained.

There was a clear trade-off seen between the quality of grass generated and the speed it could run at, as expected. The best grass we generated was from experiment set 21 (see section *7.1.21 - Experiment set 21 - codename: Galleon*), but this was far too slow to run in real time (about 3 fps). We aim to specifically focus on elevating these performance metrics of our neural rendering models in our future research works.

The performance of the CPU, GPU and memory resources provided by *Google Colab* was adequate for our purposes, and we saw good training speeds that allowed us to train an

average of ten epochs for batch size 8 in around an hour of training time. The performance of Pytorch Lightning implementation of training code was seen to be about half that of implementation in base Pytorch. It was, however, considered an acceptable compromise.

5.3. Challenges Faced

We encountered a lot of challenges throughout the development cycle of our project, and one remedy adopted to effectively tackle them was to carry out both environment development and research in parallel in a systematic feedback loop.

One such challenge we experienced early in our progression was with implementing ZMQ, since the idea was to live transfer the input dataset stream into the Python-based model training. However, this approach was surprisingly slower, significantly limiting our operation speeds due to which we later adapted to saving the datasets beforehand. Similarly, we also struggled with Unity's Barracuda library since it is relatively new, and not enough information is yet available on its documentation. This prevented us from doing the most basic and frequent tasks like tensor concatenations, and we later worked around it by implementing it ourselves in some other section.

Another major issue we faced was while integrating our Neural Renderer Playground into our working state and we simply couldn't make it work in the evaluation mode. This was later found out to be caused by drop-out regularisations not being supported in the mode, and the only option was to avoid it in future experiments. There were other contributing factors too that slowed down our research experiments, such as only having a single computer that can handle real-time testing and being unable to run our experiments on Google Colab in the background, which potentially limited us from training our models for long durations. For such challenges as discussed above, it was significantly helpful to us to have followed our customised project development cycle and implement our received feedback from previous models as improvements in future iterations.

5.4. Conclusions and Future Work

The results obtained so far have been satisfactory considering the training time and performance metrics. We have been producing grass in real-time at around 10 fps and we aim to achieve frame-by-frame consistency in our upcoming generations. The grass generations are fairly realistic, which form a solid starting testimonial to our approach for adding other objects in our input dataset along with grass or experiment with other types and densities of grass based on input colours.

We have been successful in making a dataset generator that generates unlimited amounts of training data for our networks by navigating the infinite terrains in our emulated playable environment, which is significantly less resource-intensive compared to conventional

methods. We also aim to implement real-time testing where our inputs are not saved, but directly fed from our neural rendering model into the view space with a target of around 60 fps.

We also speculated about a set of milestones that a sequence of research work fully exploring the idea could take:

Fig 5.2: Research milestones for future work

Other than refining current GAN models, another possible research would also be to explore other existing deep learning models and provide a comprehensive comparison with current work. These are some possible ideas that we wish to study in our future research works.

Possible application areas of this research developed further includes procedural generations to enhance the level of detail (possibly make it infinite) in visual renders so that virtual environments can be made more vivid and immersive in real-time, requiring the same computing resources regardless of the complexity of the details visualised. For example, dense grass consisting of a large number of grass blades would take the same amount of time to be rendered as light grass consisting of a small number of them.

The flexibility seen here with being able to drop redundant information compared to traditional techniques is also quite interesting, and can be applied in a wide variety of applications, such as image compression.

6. Bibliography

[1] Perbet, F., & Cani, M. (2001). Animating Prairies in Real-Time. In ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D'01) (pp. 103-110). Chapel Hill, United States: ACM. doi: 10.1145/364338.364375.

[2] Knowles, B., & Fryazinov, O. (2015). Increasing realism of animated grass in real-time game environments. ACM SIGGRAPH 2015 Posters. doi: 10.1145/2787626.2787660.

[3] Tiwari, A., & Fried, O. (n.d.). State of the art on Neural Rendering. Retrieved December 26, 2022, from https://onlinelibrary.wiley.com/doi/am-pdf/10.1111/cgf.14022

[4] Wulff-Jensen, A., Rant, N.N., Møller, T.N., Billeskov, J.A. (2018). Deep Convolutional Generative Adversarial Network for Procedural 3D Landscape Generation Based on DEM. In A. Brooks, E. Brooks, & N. Vidakis (Eds.), Interactivity, Game Creation, Design, Learning, and Innovation. ArtsIT DLI 2017 2017. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 229 (pp. 95-105). Springer. https://doi.org/10.1007/978-3-319-76908-0_9

[5] Stable Diffusion 2-1 - a Hugging Face Space by stabilityai. (n.d.). Retrieved December 26, 2022, from https://huggingface.co/spaces/stabilityai/stable-diffusion

[6] Wang, T.-C., Liu, M.-Y., Zhu, J.-Y., Tao, A., Kautz, J., & Catanzaro, B. (2018). High-resolution image synthesis and semantic manipulation with conditional gans. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 5589-5597). IEEE. https://doi.org/10.1109/cvpr.2018.00917

[7] Cignoni, P., Scopigno, R., & Tarini, M. (2005). A simple normal enhancement technique for interactive non-photorealistic renderings. Computers & Graphics, 29(1), 125–133. https://doi.org/10.1016/j.cag.2004.11.012

[8] Mittermueller, M., Ye, Z., & Hlavacs, H. (2022). Est-Gan: Enhancing style transfer gans with intermediate game render passes. In 2022 IEEE Conference on Games (CoG) (pp. 1-8). IEEE. https://doi.org/10.1109/cog51982.2022.9893673

Links:
[L1] roystan.net/articles/grass-shader/
[L2] catlikecoding.com/unity/tutorials/scriptable-render-pipeline/custom-shaders/

7. Appendices

7.1. Appendix A: Experiments Conducted and Architectures Implemented

The following figure represents our progression with the research development. Dead-end branches have not been included, and many experiments have been grouped together for ease of understanding. At a glance, the steady progress in the quality of grass generated can be seen. The arrows represent modification of a particular experiment set to arrive at another. Converging arrows represent combining the elements of two experiment sets, which for example gives rise to Hybrid models.

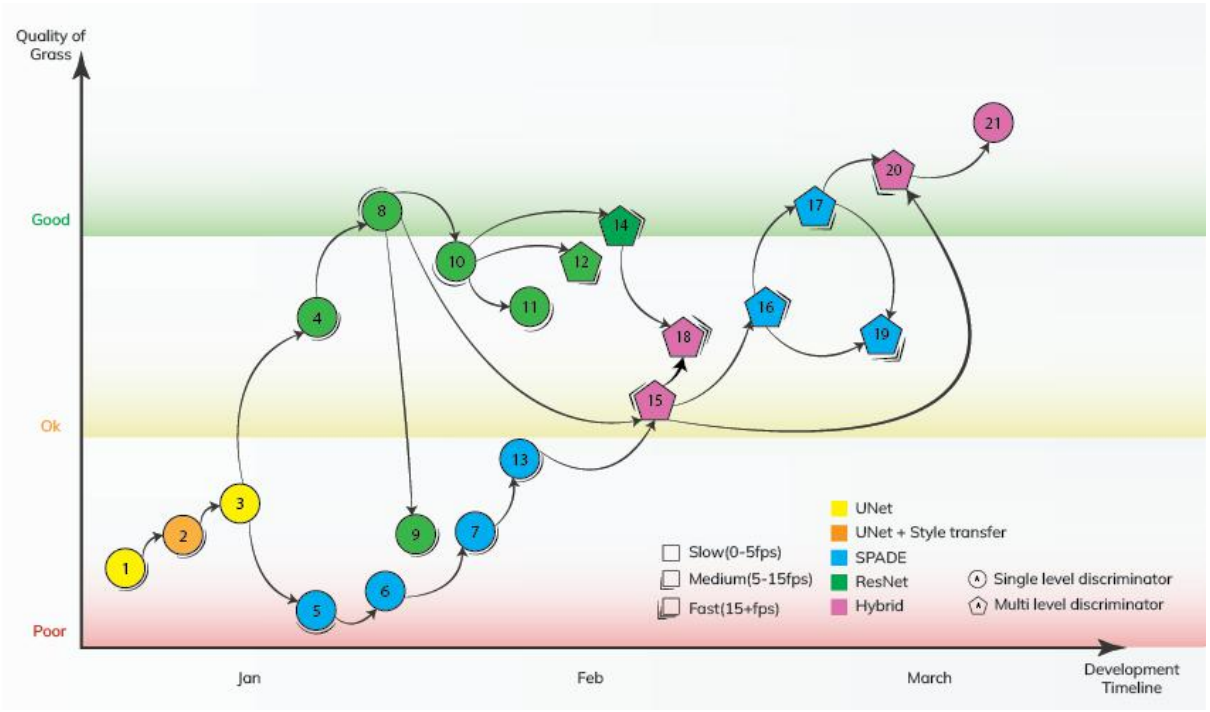


Fig: Map of the research development progression

Here is the summary of each of these experiment sets, along with the major results and findings.

7.1.1. Experiment set 1 (Codename: Sailor)

Dataset: Version 1
Discriminator: Single Level, heavy sized
Architecture: UNet

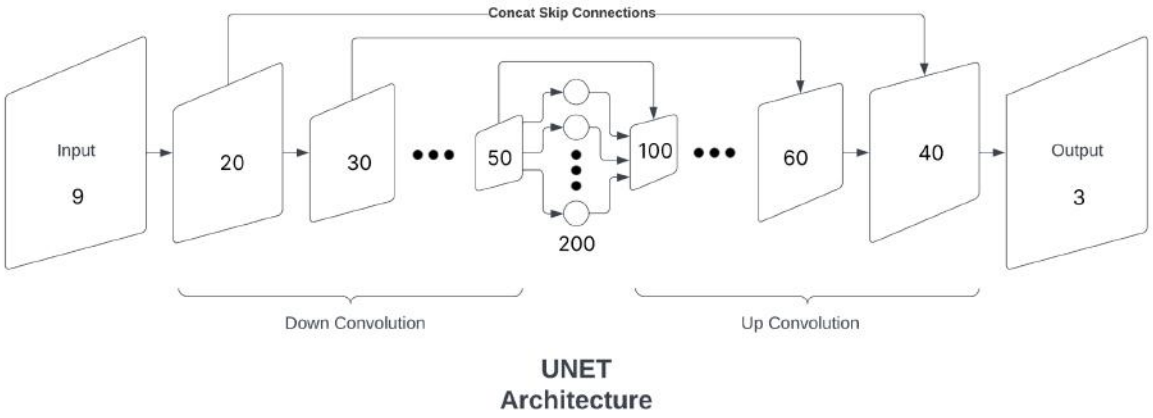


Fig: Architecture used in this experiment set

This was the first model we created, based on the famous Pix2Pix model. It has a total of 15 million parameters, but only an estimated 500,000 of them exist within the Down Convolution and Up Convolution layers. Due to a coding error, the rest of the parameters was wasted on flattening the last Down Convolution layer into the fully connected layer.

Results
This model never learned about the grass. It also seemed confused by the inclusion of the character model in the input set. Whenever it learned to render the grass better, the character model would be rendered worse, for which it would be punished and the grass would become a worse quality in the next iteration

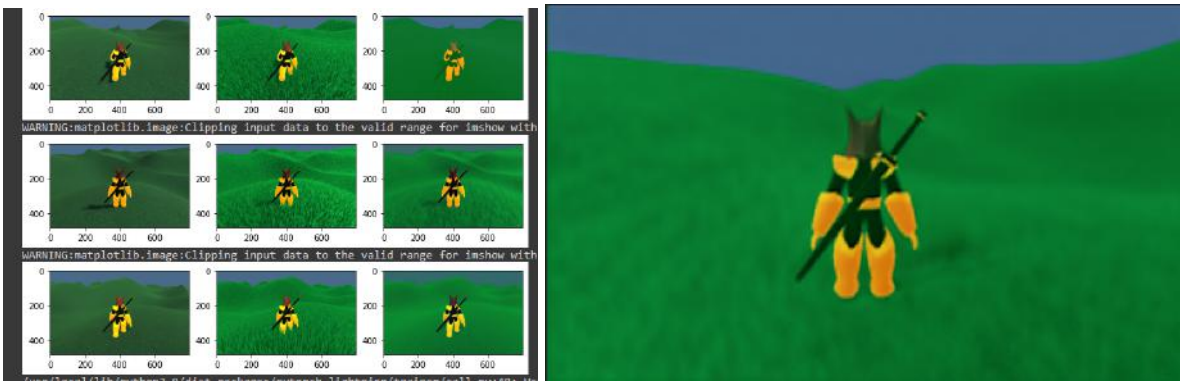


Fig: Experiment set Sailor's results

Insights

Overfitting: All of those redundant parameters managed to overfit some of the more prevalent images in the input-output set. This fact could be verified by the fact that the same model could not produce visible grass for other input-output sets - just those particular ones, and the grass it produced seemed to be a blurry version of the outputs, with the light and dark sections being in around the same places

Character model removal: After this experiment, the decision to remove the character model from the dataset was made, which gave rise to Version 2 of the dataset.

Removal of Object ID input: After the character model was removed, the only type of object in the scene was the grassy landscape, and thus this input was removed as well.

7.1.2. Experiment set 2 (codename: Sea Mantis)

Dataset: Version 2

Discriminator: Single Level, medium sized

Architecture: UNet with Style Transfer

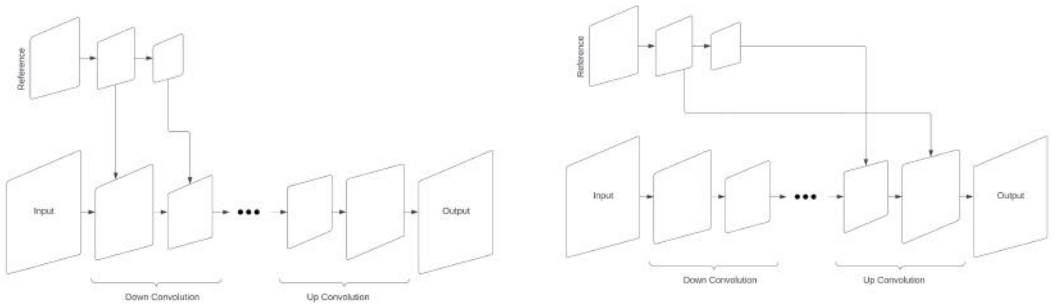


Fig: Architecture variations of this experiment set. Left is decoder concatenation, and right is encoder concatenation

Various configurations of the style transfer network were tested, including encoder or decoder concatenation, middle layer concatenation, tensor addition instead of concatenation, and also variations such as separate trainable parameters for the reference encoder, and using the same trainable parameters for the reference encoder as the main encoder.

It has a large number of parameters, around 5 million, spread across all the layers in a balanced way. The number of channels doubles as the width and height of every subsequent layer halves.

The motivation for this design was the hope that the model learns the nature of grass from the style transfer connections.

Results
Most of these configurations did not produce good results. The best results were produced by encoder concatenated configuration with separate encoders for the inputs and the references. The nature of the result was that it looked like grass was overlaid on top of the image, as if we are looking through a glass window with the grass drawn on top of it. It did not move with the image, and stayed static. The model hit a plateau after this point, and the learning became stagnant. Grass also appeared in the sky.

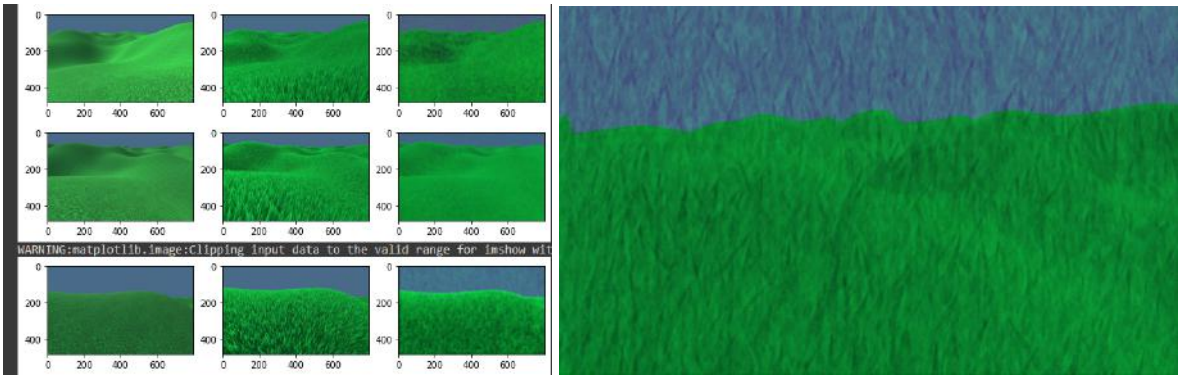


Fig: Results produced by this experiment set

Insights
The main takeaway from these experiments was that style transfer was not a good way to go about doing this. In no future model was style transfer used or revisited, for this reason. It is however not completely ruled out. This decision also simplified the input mechanism in the Neural Renderer Playground.

7.1.3. Experiment set 3 (codename: Starfish)

Dataset: Version 2
Discriminator: Single Level, medium sized
Architecture: UNet

The architecture of this model was a much more balanced version of the UNet, and with a much larger number of layers and parameters. It also did not have a fully connected layer, instead just growing from a convolved layer. The reasoning behind this design was primarily the large size of the original Pix2Pix network, and the inability of the generator in previous networks to learn about the nature of the grass, which was hypothesised to be because the model was not complex enough.

The reference grass used in Sea Mantis was also used here, but it was provided concatenated with the inputs instead of using a style transfer network. Optical Flow was removed from the set of inputs, which at this point only contained the input view and the depth map.

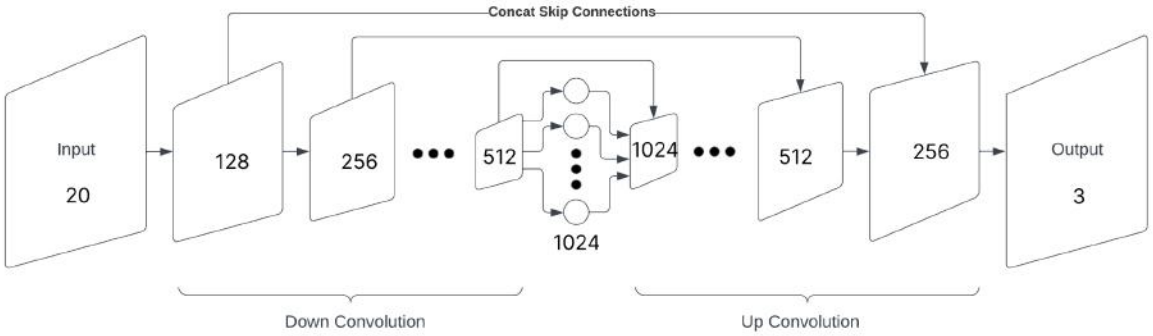


Fig: One configuration of the architectures used in this experiment

These models were trained each with the full dataset, containing about 16,000 items, for about 5 epochs.

Results

This model learned very slowly, but was the first one to successfully pick up the nature of the grass, at least at first glance. As it learned further though, it lost that trait and instead started producing a lot of repeating blocks. Since the model was heavy, it iterated slowly, and was estimated to run at about 3 fps in the Neural Rendering Playground (which had not been fully developed yet to verify this).

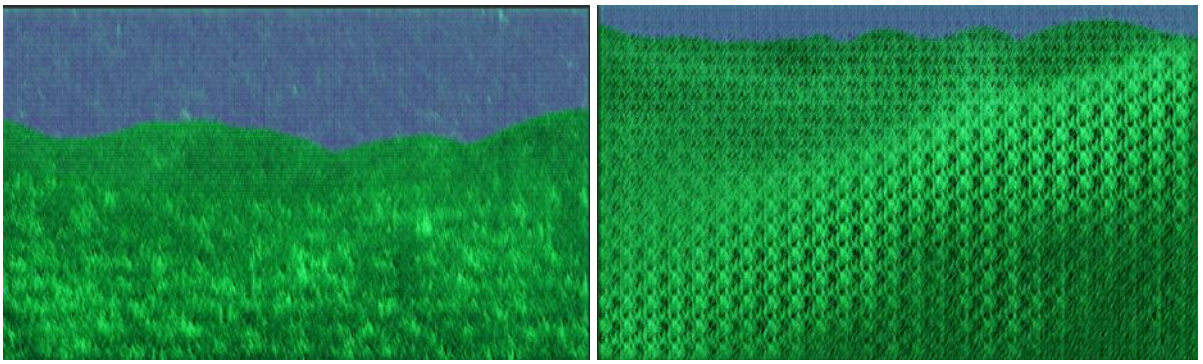


Fig: Results of the experiment. Finally picked up grass (left), devolved into repeating blocks (right)

Insights and Hypotheses

Since grass still appeared in the sky, the use of the reference grass was considered unsuccessful, and it was decided to not be used in future models. From insights gained from the Pix2PixHD paper, it was speculated that the repeating blocks were caused due to the generation size being much larger than what the Pix2Pix UNet was designed for (800 pixels width in contrast to 256 pixels). The biggest insight however, was the failure of the UNet in terms of complexity balanced with performance. To produce good results, UNet simply seemed to need to be too complex to run at a reasonable speed. Hence, the decision was made to use ResNet blocks in future models instead, which is what seemed to also remove the

repeating blocks in the comparisons presented in the Pix2PixHD paper. It was also observed that in most of these.

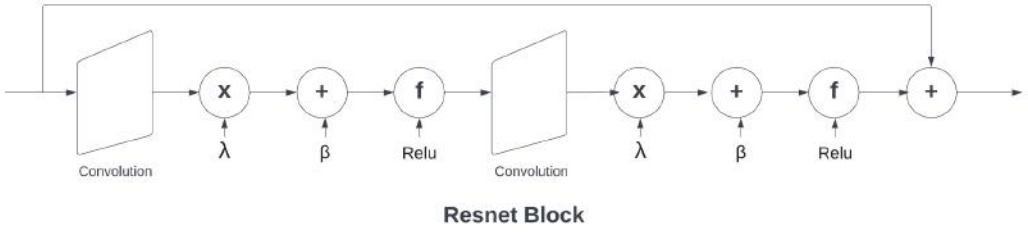


Fig: ResNet block to be used in future experiments

Upon exploring possible options, two trajectories were identified. An encoding and decoding network could be used with ResNet blocks, or a simply decoding network could be used with SPADE normalisation layers. It was decided that both of these trajectories were to be followed in parallel development lines.

It was also discovered that this model ran slower even when it was not supposed to. A lengthy debugging session was held to figure out the problem, but no particular problem could be pointed out - although it was suggested that the problem was with the code itself, not with the model alone. It was decided that for the next experiment set, the code was to be written from scratch. This turned out to be a success, speeding up the iterations in the next experiment to the expected amount.

Another observation was made suggesting that a larger number of sweeps through the same dataset should improve performance. Hence, we decided to train the next model for a large number of epochs on a small subset of the dataset. We also decided to use smaller and smaller learning rates as epochs trained.

A stagnation problem was also noticed, a point after which the discriminator was seen to not be providing useful feedback to the generator. We supposed this was because the discriminator was being overfit, and thus decided to regularise it in future experiments.

7.1.4. Experiment set 4 (codename: SeaHorse)

Dataset: Version 2
Discriminator: Single Level, lightweight
Architecture: ResNet Block Array

This model consisted of two down-convolution layers, followed by six ResNet blocks, and then two up-convolution layers. The number of channels in the ResNet blocks was made to be

128, producing a total number of parameters to be about two million. The input only consisted of the view and the depth map.
The design of this model was based on the insights gained from the Starfish experiment.

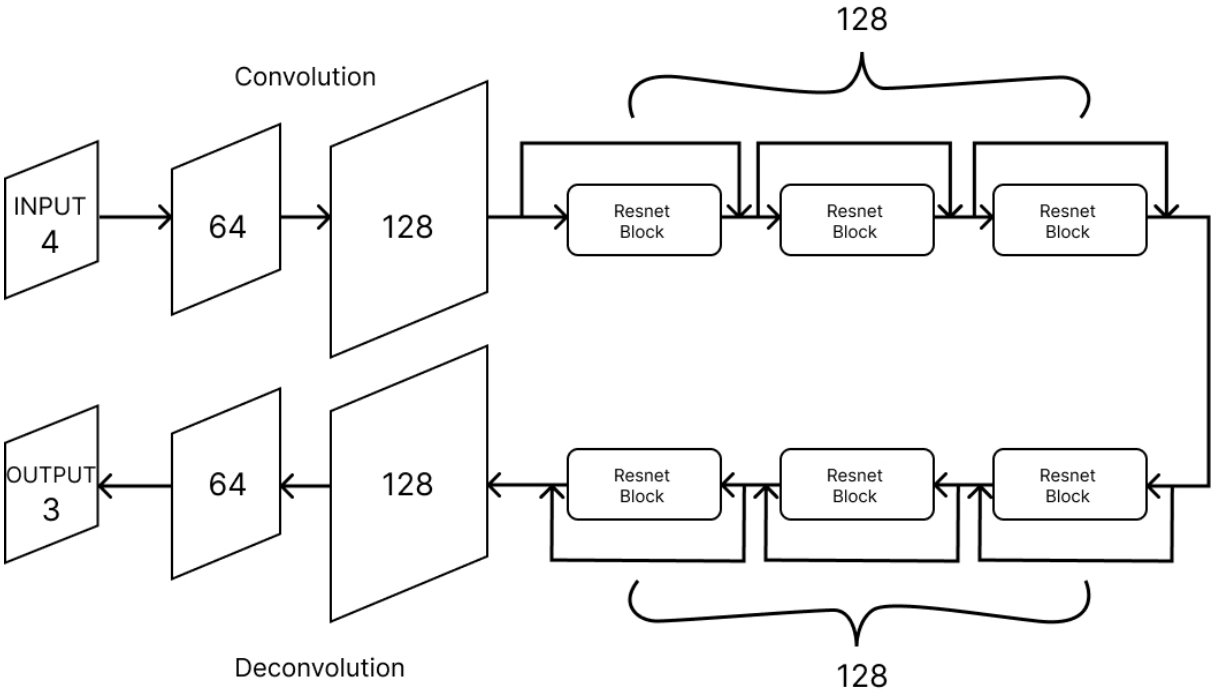


Fig: Architecture used for the SeaHorse experiment

It was trained for about 100 epochs, with three different learning rates. A large value for the first 30 epochs (0.0005), and a smaller value for the next 40 (0.0001), and the smallest value for the next 30 (0.00005)

The number of parameters in the discriminator was heavily reduced as a regularisation method to prevent overfitting. A good balance was sought out to be achieved between the adversarial loss and the reconstruction loss *(see section 4.5.3 - Loss Functions)*. A weight was given to the reconstruction loss that was about 200 times greater than the Adversarial loss, which was halved about every 30 epochs.

Results
This model turned out to be quite revolutionary. The model seemed to have learned the pointy and numerous nature of grass. It generated what looked like grass, but didn't seem to care where it put the grass blades in the first place, which was absolutely ideal. As seen in the *figure 4.1 in section 4.4 - Development Methodology*, the jump in quality was phenomenal, and we were set on a good track. The model could not be tested, since the Neural Playground Environment was not fully developed, but considering that the grass was not trying to be copied pixel to pixel, it was suggested that overfitting was minimal.

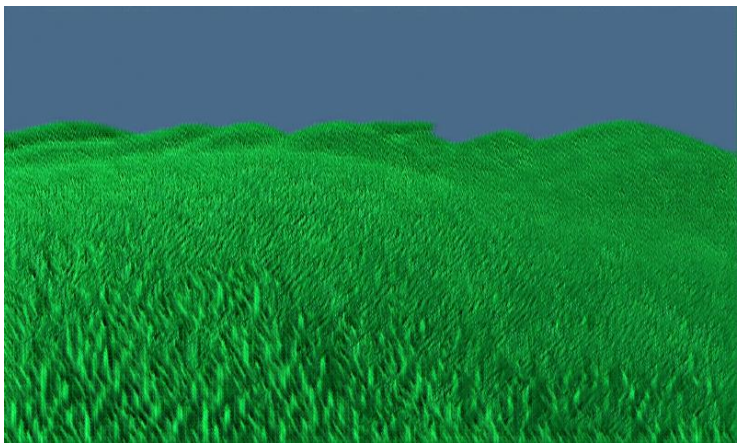


Fig: Image Generated by SeaHorse experiments

Insights
The training process with larger number of epochs with smaller training datasets, and the variable learning rate and reconstruction weight balance was considered a success. For future experiments, the balance between these values would be considered more carefully.

One interesting observation was the directionality of grass. The grass blades seemed to be pointing in random directions, even though they are supposed to be pointing outwards perpendicular to the surface. The model was essentially having to guess these directions. We figured that we could eliminate the need for the model having to do that (and instead focus on generating the grass itself) by providing that data within the input. Normals map, which records in three channels the values of vectors pointing perpendicular to any surface on the scene was ideal, and since it is part of Unity's deferred shading pipeline and produced anyway, the overhead added by the inclusion of this input was very small and thus ideal.

7.1.5. Experiment set 5 (codename: Octopus)

Dataset: Version 2
Discriminator: Single Level, heavy sized
Architecture: SPADE

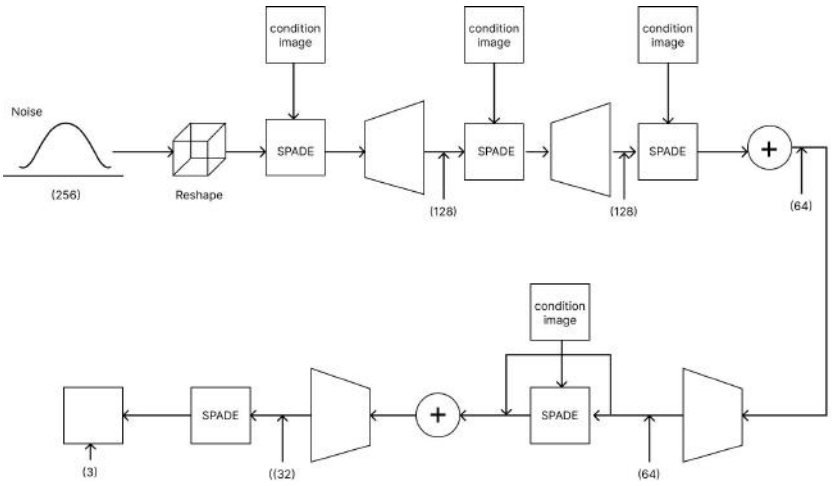


Fig: SPADE architecture for Octopus

This model was inspired from the original SPADE paper. It consisted of four SPADE Resblk, four interpolation layers and one convolution layer. Each SPADE Resblk contains two SPADE classes, two ReLU activation layers, two convolution layers, and one skip connection. The input to the network is Gaussian noise that is reshaped using the PyTorch view function, and the starting input to the SPADE block is a 128-channel input followed by an interpolation layer. Skip connections were only used in the last two SPADE Resblk, and the experiment only used adversarial loss. The learning rates for the discriminator and generator were 0.004 and 0.001, respectively, and the batch size was 8.

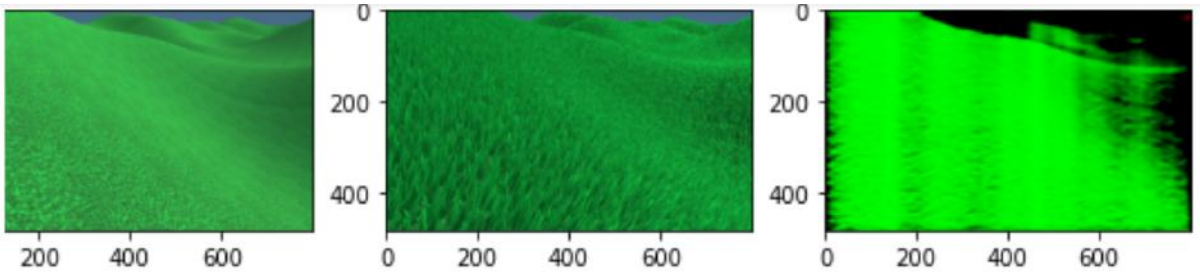


Fig: Output of SPADE model at iteration 1101/2064

Results:
The model faced difficulties in accurately reproducing colours, which was attributed to the lack of a reconstruction loss. Additionally, there were vertical lines present in the output image. Since the model was not making significant progress and continued to produce repetitive patterns, it was decided to halt the training of the model.

Insights: Without the reconstruction loss the model cannot learn fast and this encouraged us to experiment more with the lambda. We hypothesised that the presence of vertical lines was due to the use of the nearest neighbour method for interpolation. Moreover, the discriminator

was learning significantly faster than the generator, so one possible solution could be to train the discriminator only half as often as the generator.

7.1.6. Experiment set 6 (codename: Jolly Roger)

Dataset: Version 2
Discriminator: Single Level, medium sized
Architecture: SPADE

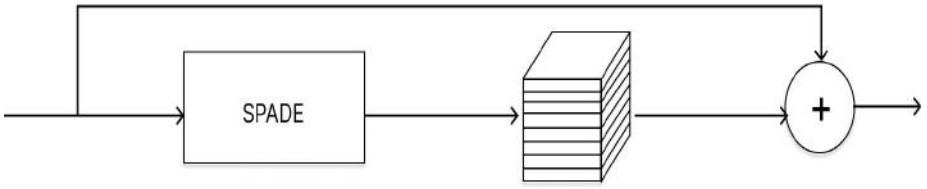


Fig: SPADE Resblk architecture for Jolly Roger

We performed this experiment by modifying the original SPADE architecture as it was too big. Since the Unity game engine generates mip-maps of images using deferred shading, we believed it would be more appropriate to directly provide mip-map images to the SPADE. During training, we generated mip-maps using bicubic interpolation, and the upsampling of the image was performed using transpose convolution. The SPADE Resblk in this case only contained one SPADE block, one convolution layer, and one skip-connection. To address the issue of the discriminator learning faster than the generator, we trained the discriminator only half as often as the generator.

Result: This model performed better in terms of speed and quality than Experiment set 5. It was able to pick the colours fast but it kept on generating a repetitive pattern. The results seemed to stagnate after a certain number of epochs. We also started with really small learning rates for both generator and discriminator which was 0.0002.

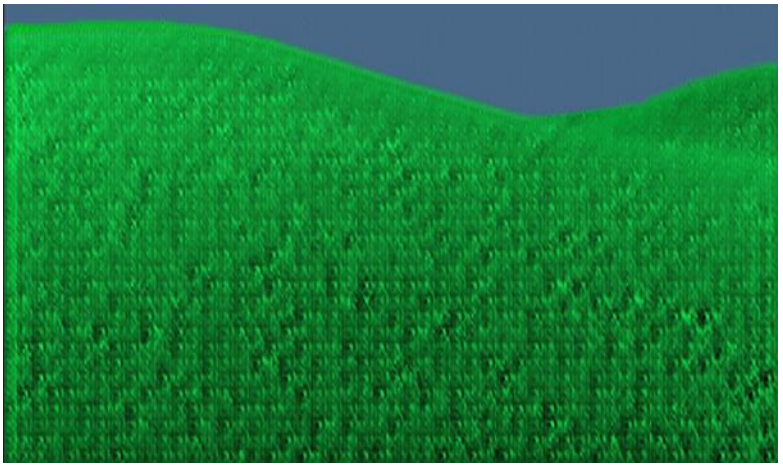


Fig: Grass generated by Jolly Roger at 2nd epoch

7.1.7. Experiment set 7 (codename: Sea Snake)

Dataset: Version 2
Discriminator: Single Level, heavy sized
Architecture: SPADE

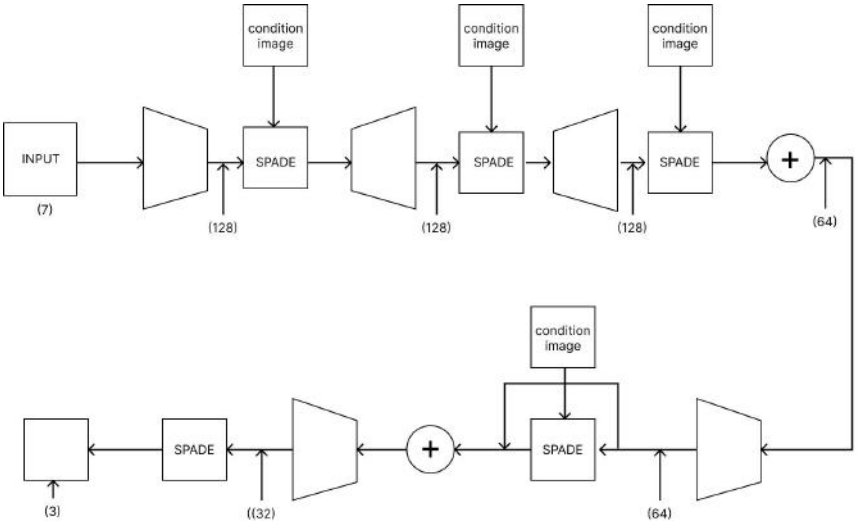


Fig: SPADE architecture for Sea Snake

As the results from the previous experiment didn't look that promising we decided to experiment with the original SPADE architecture. We provided the mip-maps as the input to the SPADE block, and instead of giving noise as an input, we provided the downscaled version of our input image as the input to the generator. We used bicubic interpolation for downscaling the image and transpose convolution for upscaling the image. We also performed the weighted average of the fake image loss and real image loss for the discriminator. The generator, with nearly 14 million parameters, had a slow training time of 1.5 seconds per iteration, and was trained twice as much as the discriminator.

Fig: Grass generated by Sea Snake at 8th epoch

Result:
This model was able to quickly learn the patterns of the grass. The large value of lambda seems to have made the learning fast.

Insights:
Bicubic interpolation worked better than the nearest neighbour method and using transpose convolution helped to solve the problem of vertical lines completely.

7.1.8. Experiment set 8 (codename: Leviathan)

Dataset: Version 3
Discriminator: Single Level, lightweight
Architecture: ResNet Block Array

The architecture of this model is identical to the one used in the SeaHorse experiments (experiment set 5)

The number of input channels however was increased to 7, to accommodate the normals map in the newly generated dataset.

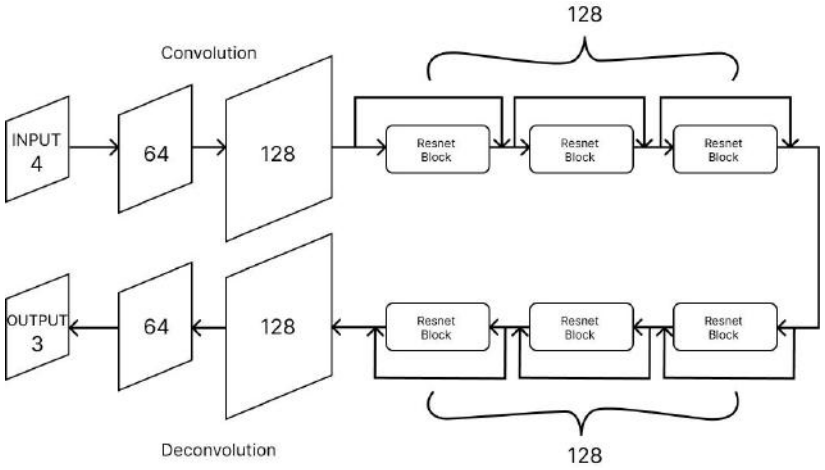


Fig: Architecture used in this experiment

It was also trained for 100 epochs with about 2000 items in the dataset, but with five levels of learning rates were used instead of three. The variance in that set of five learning rates was made smaller for the discriminator compared to the generator. Afterwards, the training was continued for 4 epochs with the full dataset (about 8192 items)

Results
This was another big jump in the quality of the grass generated, right on par with the best results we have gotten so far. The inclusion of the Normals map was considered a success, and the balance of learning rates and loss value weights changing over time was considered responsible for the improvement.

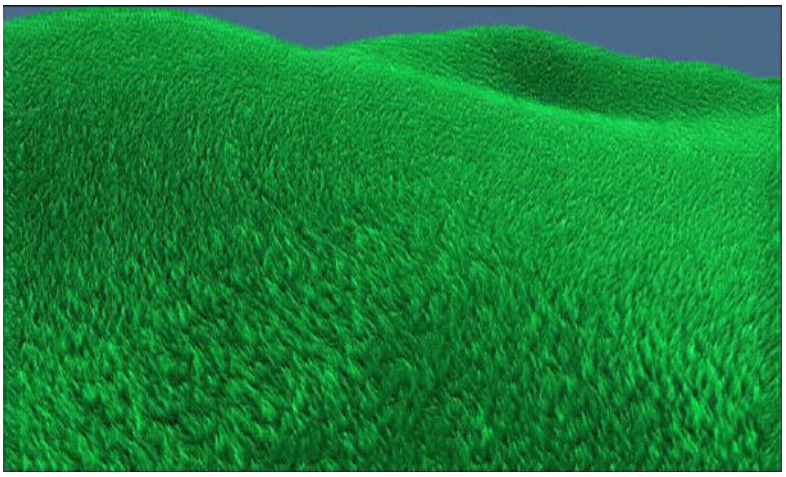


Fig: Grass generated by Octopus after the full training as described

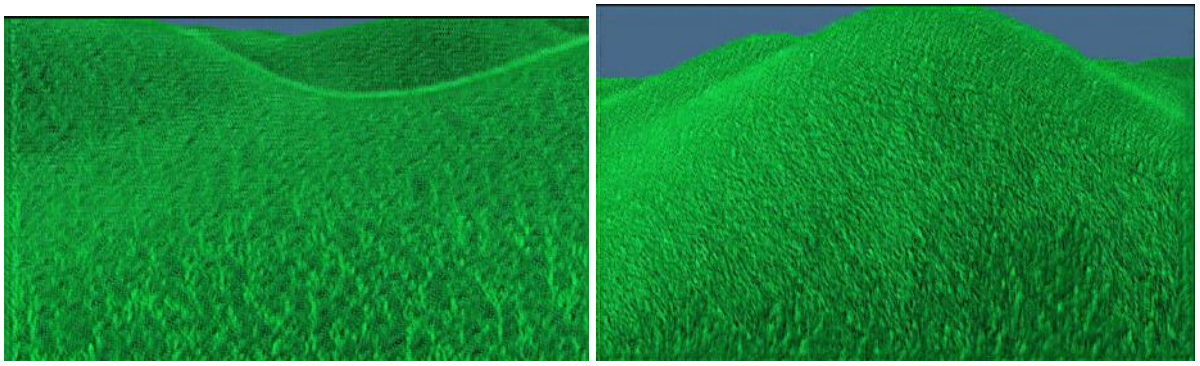


Fig: Grass generated at epoch 50 (left) and epoch 100 (right)

Insights and Problems Faced: A lot was learned during this training process about balancing the hyperparameters and having them change over time. One insight observed was the fact that during training, the model first trained to produce a blank green surface instead of a textured one as the result, and only then it would learn to draw grass on it. This convinced us to generate yet another dataset where instead of having textured grass in the input, we would simply have a blank green surface. The hope was that training would be faster.

However, the biggest problem was discovered when the Neural Renderer Playground was built to the working state. This model simply did not work in evaluation mode, only in training mode. After a long, multi-day testing and debugging session, the batch normalisation layers and the drop-out layers were considered the culprits. In later experiments, it was discovered that batch normalisation was not the problem, and it was just the drop-out layers which seemed to deterministically drop out certain learnable parameters, making the model pick up on that pattern and take advantage of it. In the end, we could not make this model work without drop-out, and thus could not run it in the Neural Renderer Playground. The decision was also made to not use drop-out regularisation in future experiments. Since it is such a useful method of regularisation, the consequences of its removal weren't known.

7.1.9. Experiment set 9 (codename: Oyster)

Dataset: Version 3
Discriminator: None
Architecture: ResNet block array

The architecture of this is identical to that of Experiment set 8 (Leviathan). It was trained for about 30 epochs, and stopped as the results did not improve for over 10 epochs. The purpose of this experiment was to see what would happen if we did not use a GAN, and instead simply trained the generator using Reconstruction loss. It was supposed to give us insights into what the purpose of the discriminator actually is.

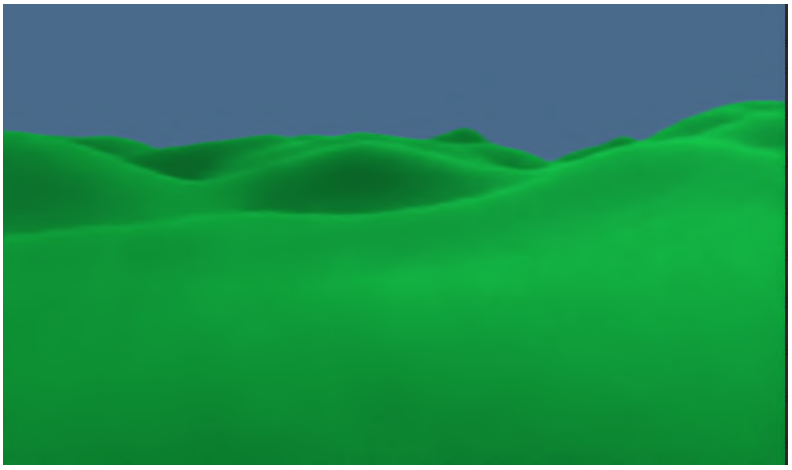


Fig: Image generated by this model

Results: The model learned to get the overall colours and large scale lighting correctly. This lighting showed where the grass would glimmer and how that would be different from if the terrain were flat. However, no grass was generated.

Insights: The purpose of the discriminator was seen in this experiment - to learn about the nature of the grass, which is what was hypothesised during the planning of this project. Without a neural network model learning about the pointy, bendy and numerous nature of grass, the model would either have to be extremely large and learn about it pixel-by-pixel (likely overfitting the dataset), or it would only produce the overall drastic change in the image - the colouring changes. This was quite conclusive that we absolutely needed a GAN to train this network.

7.1.10. Experiment set 10 (codename: Seagull)

Dataset: Version 4
This model was the first to use this dataset, which contained smooth, blank green landscape as the input, as well as the depth map and the normals map. The decision to use untextured smooth inputs was made from the insights collected in Experiment set 8 (Leviathan)
Discriminator: Single Level, lightweight
Architecture: ResNet Block Array

This was the first model using the new ResNet Block architecture that would be used in every subsequent experiment. This architecture did not have the dropout layer.

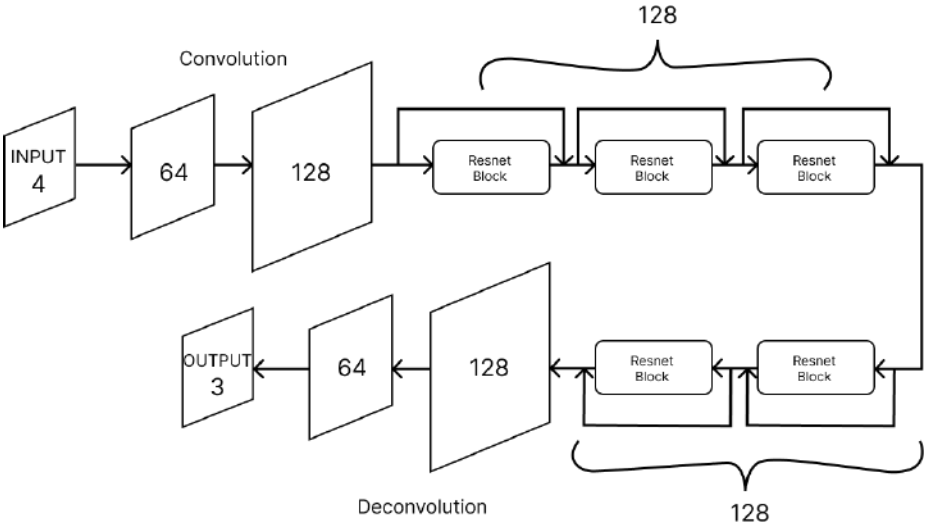


Fig: Architecture used in this experiment set

In order to compensate for the loss of dropout regularisation, one of the ResNet blocks was removed, making it 5 total. The training was done in a very similar manner to the Leviathan experiments, but it was cut short after about 40 epochs of training, after a potential problem was noticed (banding)

Results: Since the training wasn't done for as many epochs as the Leviathan experiments, direct comparison was difficult to make. It seemed to be on track, having a similar quality compared to the images produced by Leviathan after a similar amount of training. An immediate problem was noticed: grass seemed to grow on particular lines, creating rows of lines as patterns (see figure below)

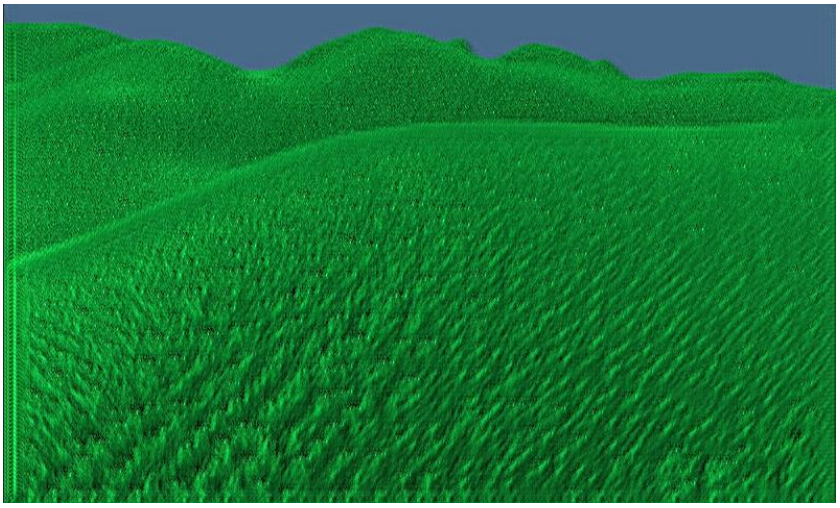


Fig: Image produced by this experiment, showing obvious banding-related patterns

Insights: This was the first model to be loaded and run successfully in the Neural Renderer Playground in unity. The cause of the patterns in the image was obvious during testing in that

playground: the patterns did not follow the movement of the landscape as the player navigated around, instead it followed the lines where lighting changed from darker to lighter shades. It was determined that the edges of banding patterns in smooth gradient lighting were being used as seed points to grow grass on. The solution to this problem is adding noise to the input images, to create more obvious seed points for the model to cling to. The issue with this, however, is that adding noise to the images themselves would not provide consistency of the noise patterns between frames. It was thus decided that the noise would be added onto the terrain itself using a shader, and a new dataset (version 5) was to be created.

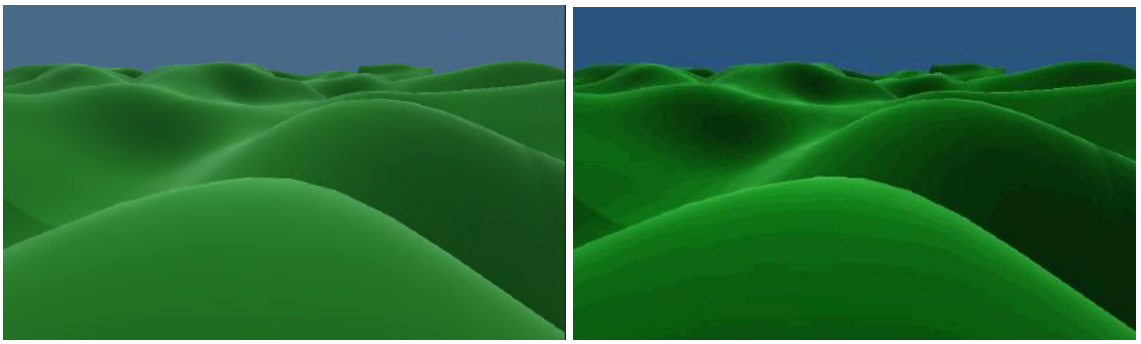


Fig: Input image (left), and exaggerated contrast showing banding lines which were used to calculate the seed points to create grass. This pattern is not obvious visually; the model, however, seems to have picked it up.

7.1.11. Experiment set 11 (codename: Hammerhead)

Dataset: Version 4
Discriminator: Single Level, lightweight
Architecture: ResNet block array

The architecture of the model used here was identical to that of Experiment set 10 (Seagull). However, it used batchnorm layers in the discriminator for one run, and spectral norm layers in it for another identically trained run. Both results could thus be directly compared

Result: the experiment with Spectral Norm started producing grass-like artefacts a few epochs before the experiment with batchnorm. Both experiments produced similar results, but the Spectral Norm seemed to be learning faster, until they both slowed down to the point where it was difficult to tell.

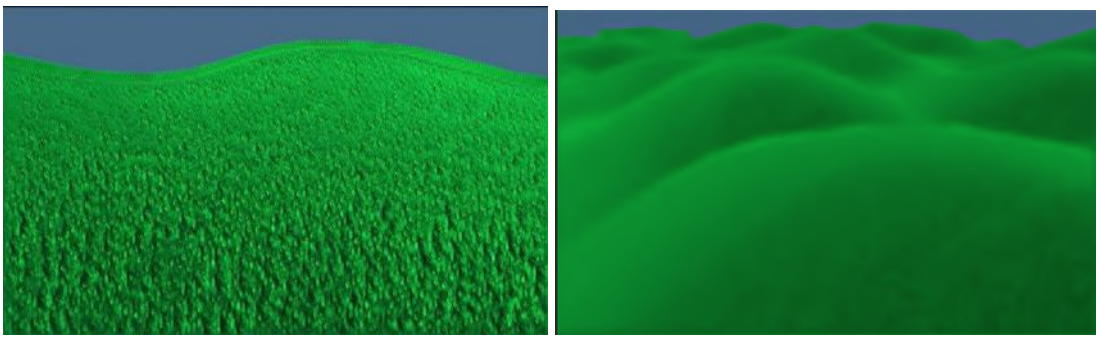


Fig: Results produced by the model trained using spectral norm discriminator (left) and the batch norm discriminator (right) after a similar amount of training

Insight: Spectral Norm was seen to be better than Batch Norm, although not very conclusively. Further experiments would be required to verify it further. However, for now, the decision was made to use Spectral Norm in the discriminators of future experiments.

7.1.12. Experiment set 12 (codename: Coral)

Dataset: Version 4
Discriminator: Multi Level, lightweight
Architecture: ResNet block array

The architecture of the model used here was identical to that of Experiment sets 10 and 11 (Seagull and Hammerhead). This was also an experiment testing variations of the discriminator. It was trained in the same way as Hammerhead, making it possible to directly compare the results. The discriminator was split into two models, both PatchGANs with similar layers, but one works on the full sized output, while the other works on a scaled down version. Two different loss values are produced, which are weighted and added.
Result: The multilevel discriminator quite conclusively managed to train faster than the single level discriminator, showing grass artefacts as many as 5 epochs earlier.

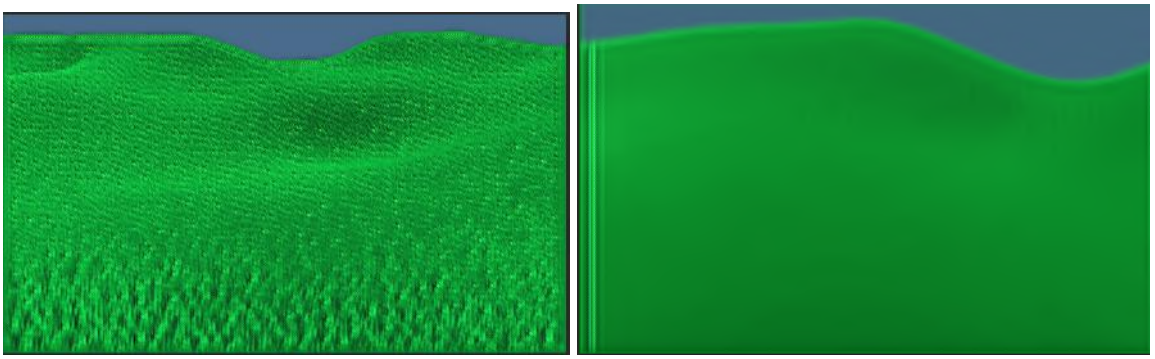


Fig: state of the grass after similar amounts of training using the multilevel discriminator (left), and the single level discriminator (right)

Insights: The decision was made to use the multilevel discriminator for all future experiments.

7.1.13. Experiment set 13 (codename: SwordFish)

Dataset: Version 5
Discriminator: Single Level, heavy sized
Architecture: SPADE

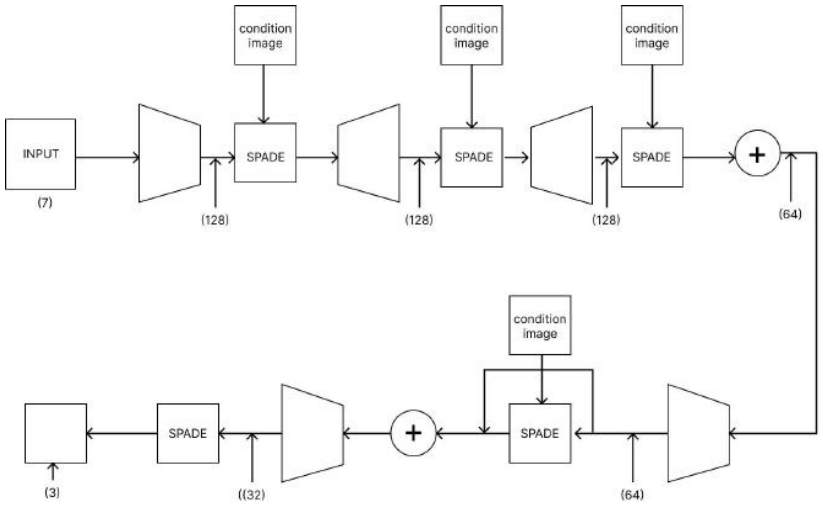


Fig: SPADE architecture for SwordFish

It followed the same architecture as the experiment set 7 with minor changes. From our previous experiments we found that the order in which the generator and discriminator are trained can affect the results. To ensure the best possible outcome, we opted to use PyTorch Lightning for training. Both the real image loss and fake image loss were assigned equal weights, and the learning rates for both the generator and discriminator were decreased to 0.0003 and 0.00001, respectively

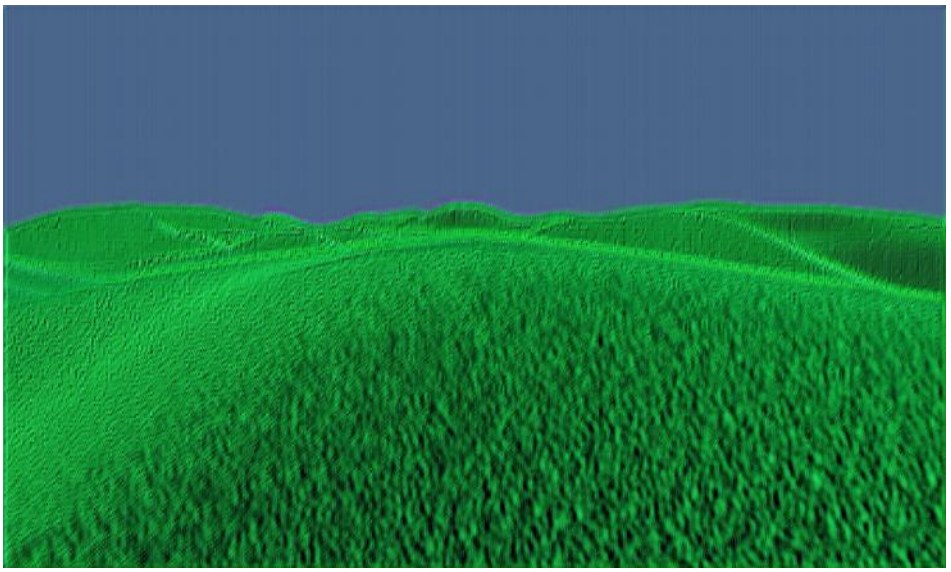


Fig: Grass generated by SwordFish at 7th epoch

Results:
The use of PyTorch Lightning appeared to have improved the results, as the model was able to quickly learn about the grasses. However, after a certain number of epochs, the model's performance did not improve significantly. Nonetheless, of all the SPADE models we experimented with, this one demonstrated the most promising results.

7.1.14. Experiment set 14 (codename: Kelp)

Dataset: Version 5
Discriminator: Multilevel, lightweight
Architecture: ResNet block array

The architecture of this model consisted of five ResNet blocks which operated on tensors twice as large as the ones in previous experiments, since only one downsampling layer and one upsampling layer was included. The ResNet blocks also had only 64 channels each, decreasing the number of parameters a lot. This model, codenamed Kelp, or Large ResNet blocks, was used using different configurations, including different number of layers, channels, and different hyperparameters, to test the limits of what such an architecture can achieve. Experiments were branched as well, using different configurations after training for a certain number of epochs with the same configuration.

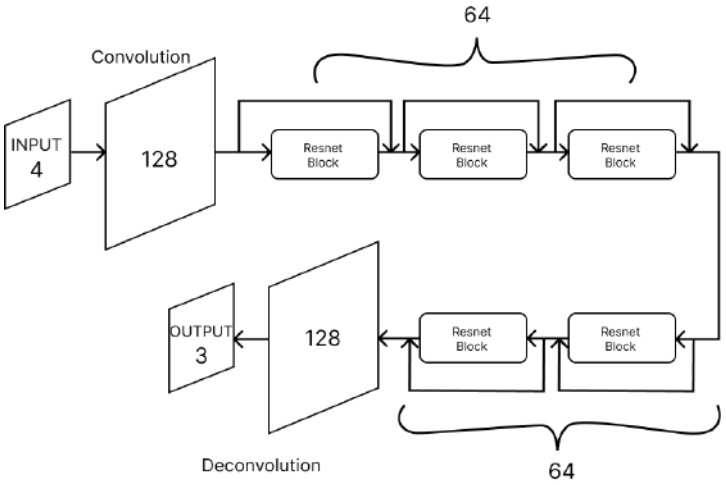


Fig: Model used in this experiment

This model used a new dataset, where noise was added to the input.

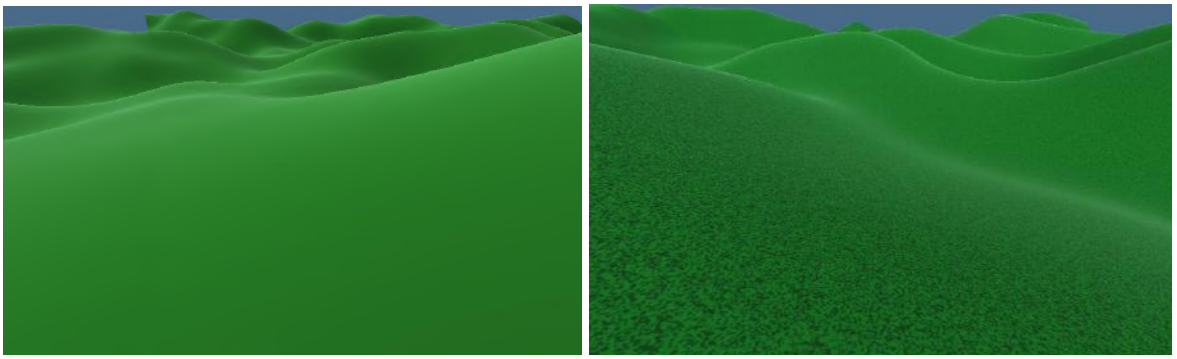


Fig: Smooth terrain (left) and noisy terrain (right)

Results: With the noise being added to the terrain space, the banding pattern problem was solved, with such patterns not being visible anymore. The large ResNet blocks were able to produce quite good looking grass with a very small number of parameters

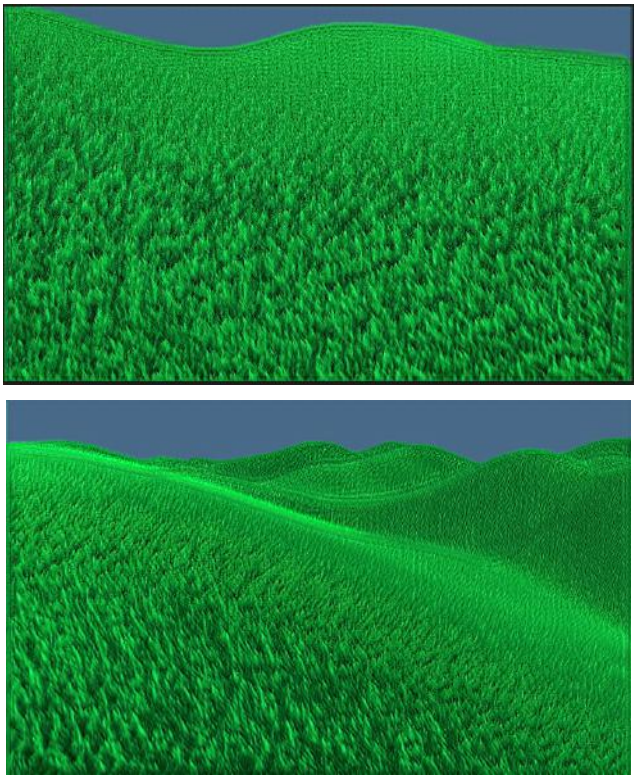


Fig: Large ResNet blocks only required about 350,000 parameters to learn up to this point (top), and 800,000 parameters not doing that much better (bottom)

An 800,000 parameter version was used for branching testing. First, it was trained for 50 epochs using the established training methods and learning rate changing techniques. One continuation of this resulted in mode collapse, which was for the first time visualised using graphs.

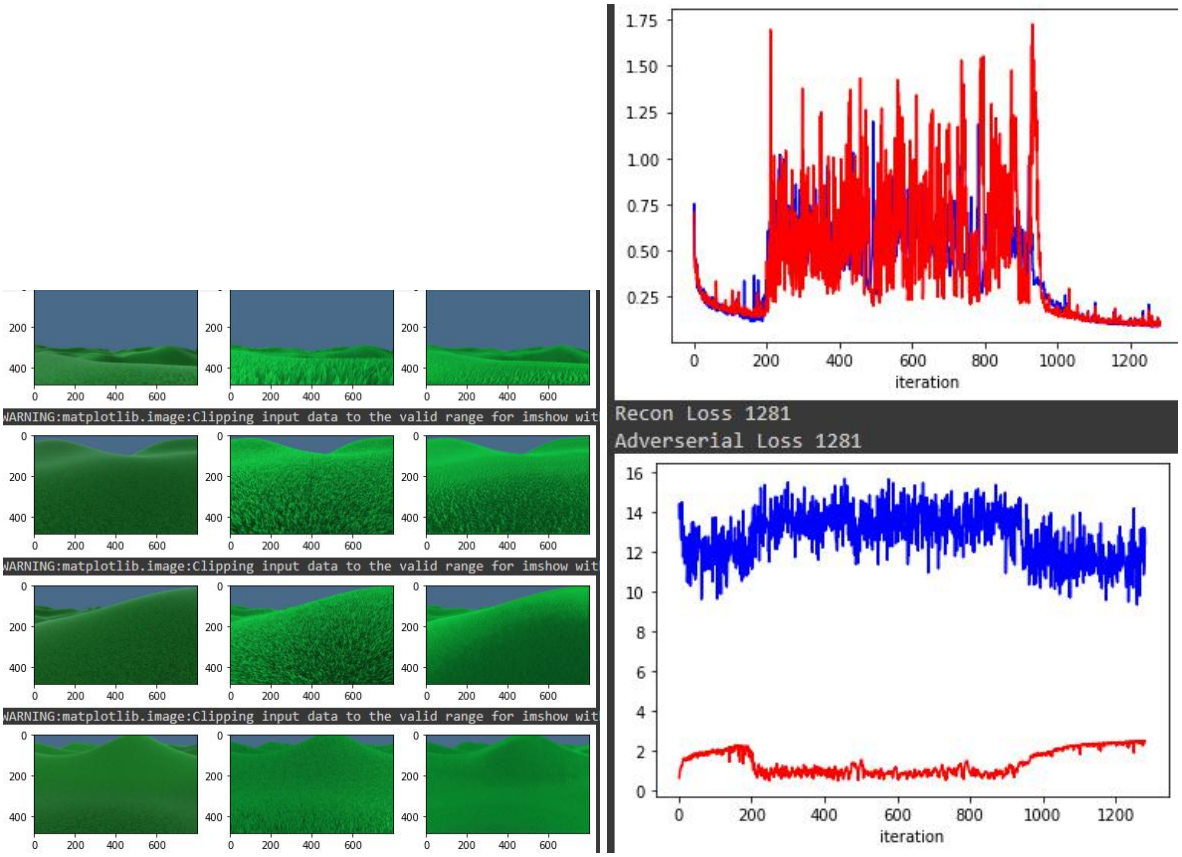


Fig: Model hitting mode collapse

Another continuation was performed with a lower learning rate for the discriminator, and a lower weight for the reconstruction loss. Different values for those were experimented with.

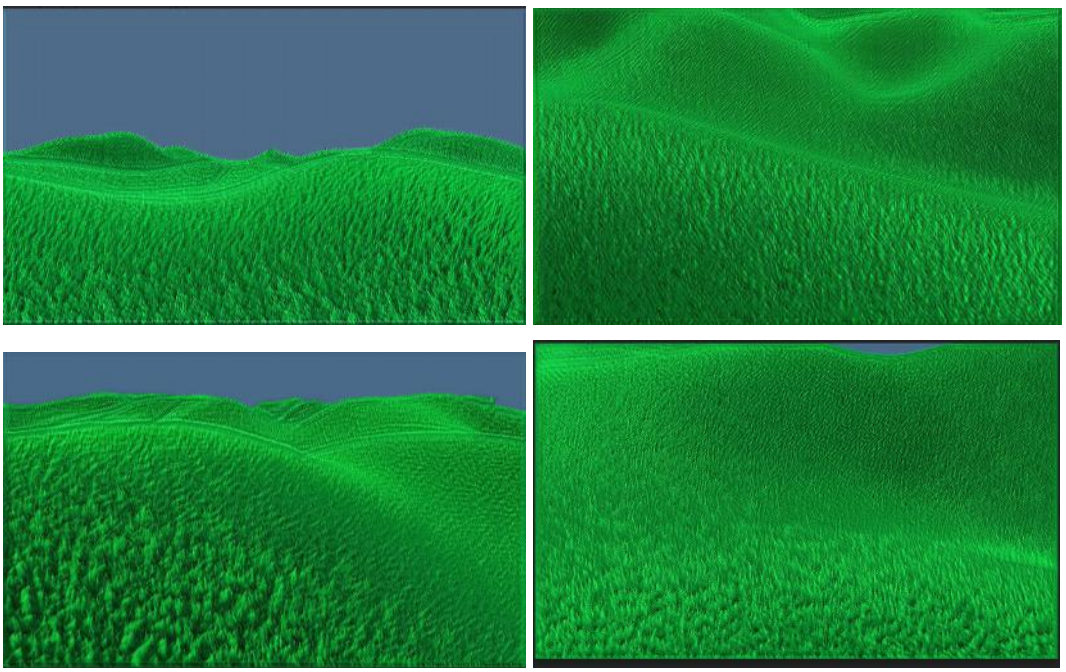


Fig: Different values for reconstruction weights tested, producing varying and inconclusive results

Finally, one continuation was performed without the reconstruction loss at all, and it provided some interesting insights into the working of the discriminator

It seems to have lost the tendency to get colours accurate, and is producing a lot of grass in the same place, causing this bright, dense grass effect. It seems to validate that the discriminator has simply learned to count how much of this grass there is in the image, and produce better scores for bigger counts.

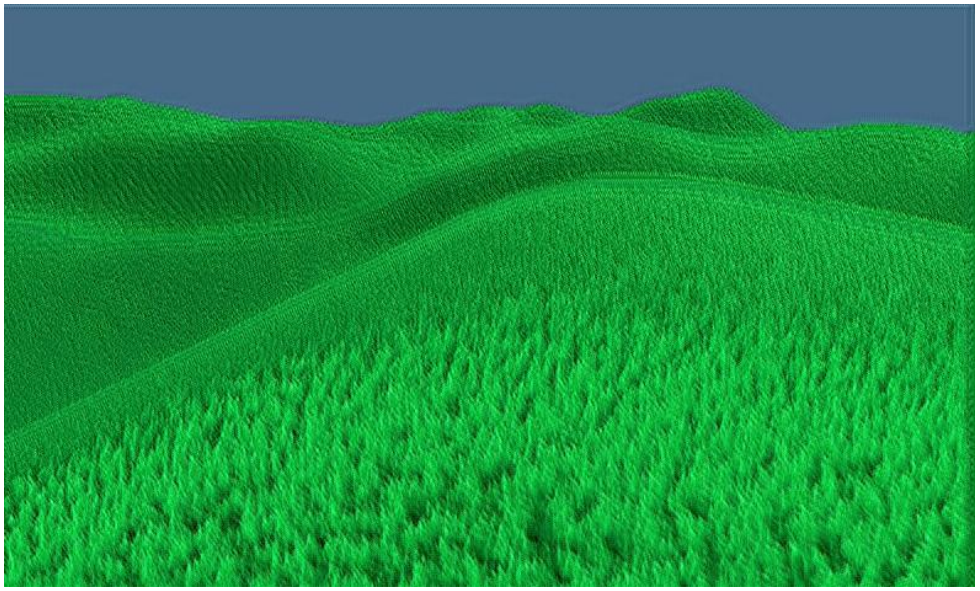


Fig: The bright, dense grass effect, which seems to be favoured by the discriminator after the continuation of the training

Insights:
These experiments provided two main insights. First, the value of the weight of the reconstruction loss is important and must be kept within 30 and 200 times the weight of the adversarial loss to produce good results. The other insight is that large ResNet blocks seem to be incapable of learning to draw large structures such as big close-up grass blades. We also learned that a smaller number of parameters does not always mean higher speed, since it now takes more time to process the large tensors being passed through the large ResNet blocks. Speed, thus, was considered to not just depend on how heavy the network was in terms of number of parameters, but also how large the images being processed are, the two of which relate to each other as trade-offs.

7.1.15. Experiment set 15 (codename: Squid)

Dataset: Version 5
Discriminator: Multilevel, lightweight
Architecture: Hybrid

This model represented the first of the architectures that we designed ourselves based on insights gained from previously trained networks. It is a lightweight network with only 760,000 parameters. The motivation for the design came from the line of thought that processing on smaller images is more likely to help build larger structures such as taller close-up grass, which the large ResNet blocks architecture failed to do. This model was inspired by the SPADE models, but since the SPADE models so far hadn't reached generation quality compared to even the worst of the ResNet models, a new approach was considered - replacing all of the SPADE ResNet blocks with the regular ResNet blocks, and replacing just single batchnorm layer into a SPADE layer, written with as little code as possible.

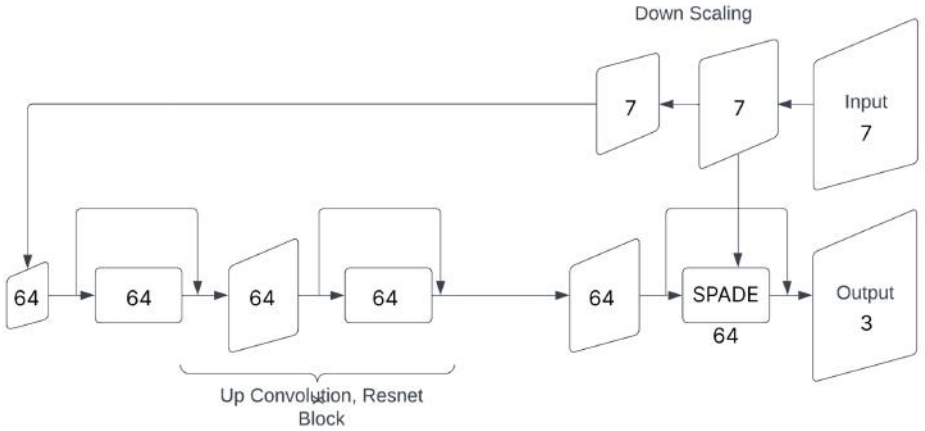


Fig: Architecture of the Hybrid model used in this network

The model was trained with two versions, one containing the SPADE normalisation after the first convolution layer of the final ResNet block, and the other containing the SPADE normalisation layer after the second convolution layer.

Results: The most notable result from this experiment was how quickly the model trained, showing grass-like artefacts within just a few epochs (4-5) and producing results looking as good as Leviathan's epoch 40 images within 15 epochs.

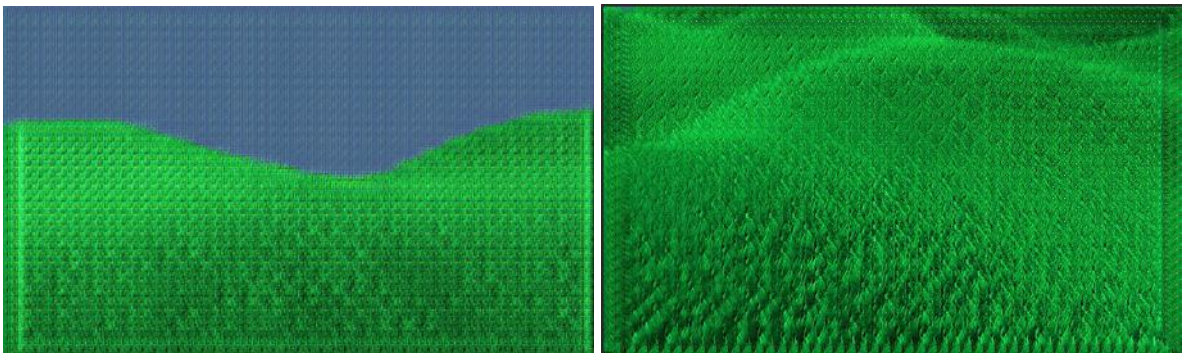


Fig: Results from the Squid model, showing the repeating blocks, and the messy border

Two notable unwanted artefacts in the generated images were heavily repeating blocks, and thick repetitive borders.

Insights and Hypothesis: The architecture of this model was considered a success, being able to produce larger grass-like structures than the Large ResNet blocks, and also being much faster despite having more parameters. It was suggested that the failure of the SPADE architectures in the previous experiments was due to coding errors rather than problems with the architecture, and it was decided that all SPADE normalisation layers in future models will be built using code from this experiment.

It was hypothesised that the repetitive blocks were simply caused due to there being too few parameters. The thick repetitive border problem was attributed to padding issues - all layers used zeroes-padding, which would be changed to reflection-padding in future experiments.

7.1.16. Experiment set 16 (codename: CuttleFish)

Dataset: Version 5
Discriminator: Multi-level, lightweight
Architecture: SPADE
This model has the same architecture as experiment set 15, but with two important differences: reflection padding is used to reduce borders, and every ResNet block has been replaced by the Half SPADE ResNet block. This technically does not make it a SPADE model, but since inputs are provided to every ResNet block, we have categorised it as such.

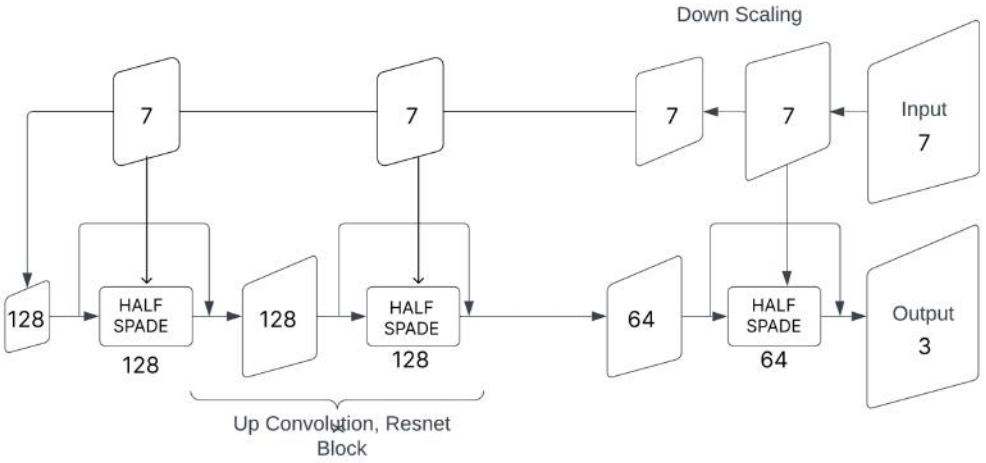


Fig: Architecture of the model

It is a relatively heavy network, containing double the number of channels in earlier layers, and has more than 2.2 million parameters.
This experiment ran faster than originally anticipated, making it applicable to use similarly heavy networks. It was trained for about 35 epochs, but was producing good results by epoch 10, for a training set with 2048 items.

Results: The repetitive patterns and think borders both are almost non-existent in these results, validating our hypotheses on how they occurred. There was however a lot of

fluctuation in the training, with it stepping back a lot in quality before having to basically start over again.

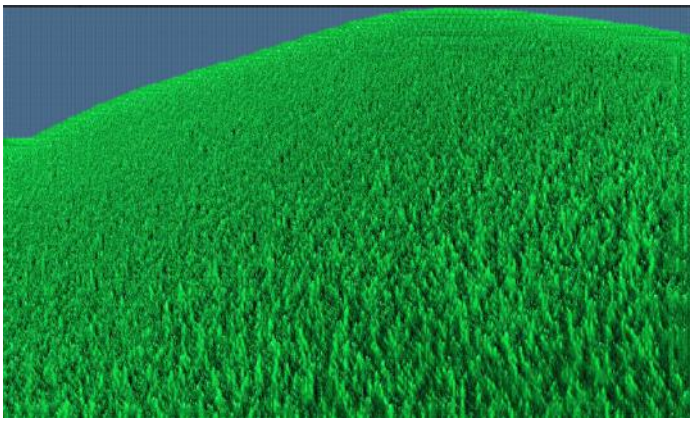


Fig: One of the better images produced by cuttlefish, showing the repetitive blocks and border problem mitigated

Insights: The fluctuations in the training was attributed to the unpredictability of the reconstruction loss dynamics. The model may be producing good grass, but it is punished unfairly by the reconstruction loss. Decreasing the weight of the reconstruction loss, however, was shown to not produce good results in previous experiments. A different idea was decided on: to use the reconstruction loss on a downscaled version of the image, effectively removing the finer details (which may contain grass), for which the model would not be punished, but it would still be pushed to get the colours correctly. Blurring the image was also considered, but has not been explored in any future experiments within the scopes of this project.

7.1.17. Experiment set 17 (codename: Siren)

Dataset: Version 5
Discriminator: Multi-level, lightweight
Architecture: SPADE

The architecture of this model is identical to that of Experiment set 16. The main testing point for this was the idea of downscaling the generated image before calculating the reconstruction loss. The reasoning for this is described in section 15.1.16. Another testing point explored by this set of experiments was automatic decaying of learning rates and reconstruction loss weights. All of these variables were tested separately, making this a particularly large set of experiments. This was another branching experiment, with training being continued after some time using different branching configurations.

Results: The experiments with the decaying factors provided inconclusive results, except for with the decay of the reconstruction loss weight, which consistently produced worse results than simply having it remain static. This decay was done only once per epoch, to not affect the optimiser algorithms, but it seemed to anyway.

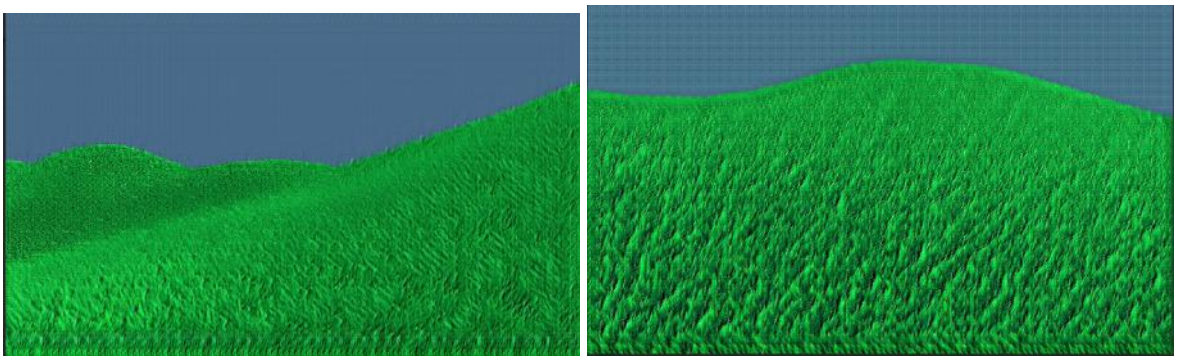


Fig: images produced with decaying reconstruction loss weight (left) and static reconstruction loss weight (right)

Further experiments were conducted to test the downscaling of the generated image before calculating the reconstruction loss. The results are very similar, especially considering they are fluctuating. However, it could be seen overall that a downscaling factor of 10 produced slightly worse results than factors 2 and no downscaling at all.

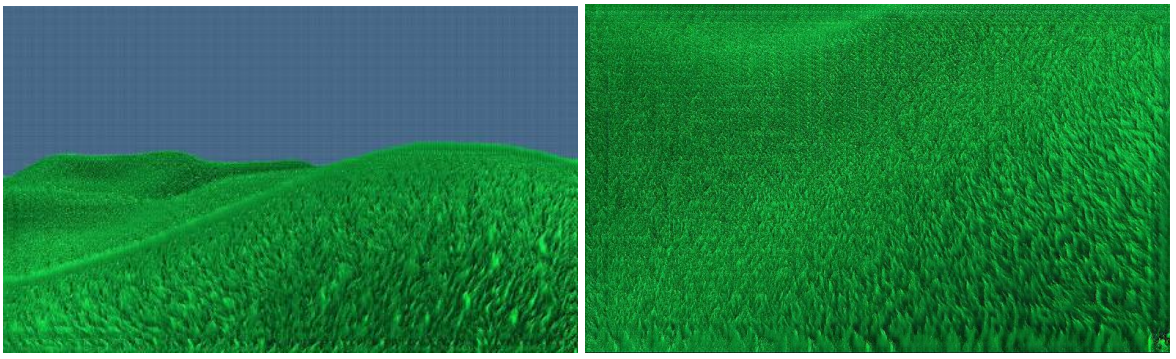


Fig: Images produced by downscaling factor of 10 (left) and 2 (right). The image on the right seems to have captured the nature of the grass better, with the image on the left producing some blobby artefacts

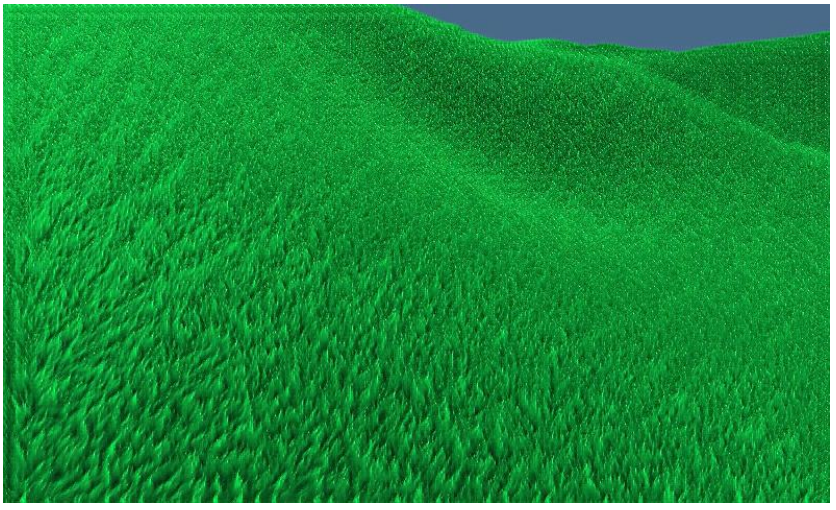


Fig: Image produced without using a downscaling factor, the best grass generated so far

The difference between the quality of grass generated by a downscaling factor of 2, and no downscaling at all could not be considered conclusive seeing the fluctuations in the quality. However, the best results produced by no downscaling were better than the ones with the downscaling factors, being on par with the results of Experiment set 8, and the best results produced so far.

Insights: Downscaling factor for reconstruction loss was considered a failure, and it was decided that this technique would not be used for future experiments.

7.1.18. Experiment set 18 (codename: Hydra)

Dataset: Version 5
Discriminator: Multi-level, lightweight
Architecture: Hybrid

This architecture was another attempt at putting together all we learned into a custom design, and testing out an interesting new idea - fractals. We figured that if a ResNet block can draw small blades of grass on a large image, the same convolutions could draw large blades of grass if simply provided smaller inputs. This could be taken advantage of by passing multiple sized inputs through the same ResNet block, upscaling them to become the same size, adding them up, and then using some further convolutions. There was no SPADE used in this model, but could be considered for future exploration. It is still considered a hybrid though, due to the strange, non-linear architecture.

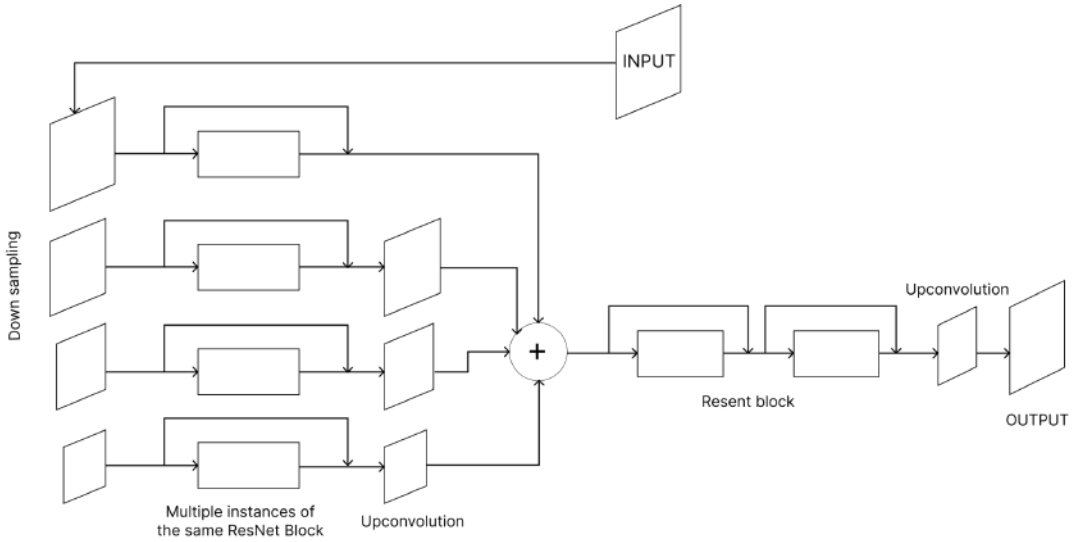


Fig: The strange architecture of this model

Results: This could not produce images on the level of quality as previous experiments. The things it did generate had a different finer structure to it, showing the possibilities of this

concept could be explored further. The generated images also contained a lot of unmoving white dots, which is unacceptable.

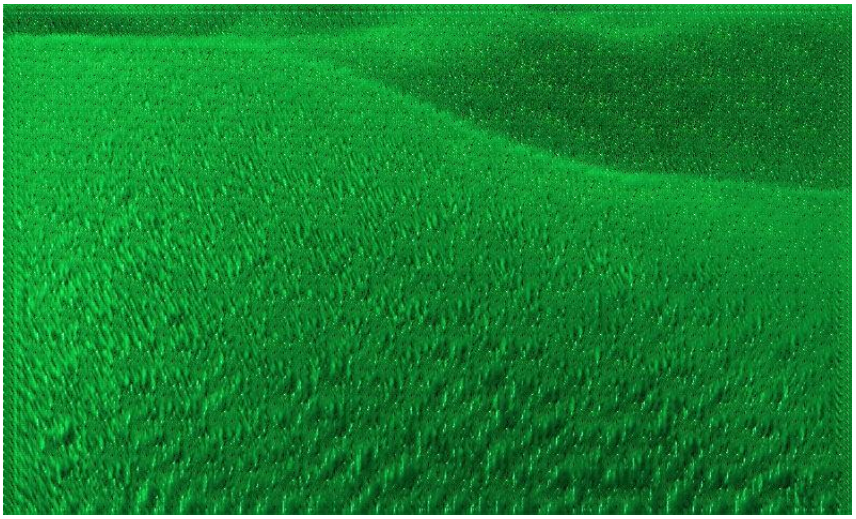


Fig: An interesting result generated by this model

During training, the quality of the image generated peaked, and then started getting worse. The reasons for this behaviour is unknown.

Insights: The exploration of this idea was cut off due to time constraints. It certainly seemed like an interesting exploration path, but a little too new of a territory to rely upon.

7.1.19. Experiment set 19 (codename: Dolphin)

Dataset: Version 5
Discriminator: Multi-level, heavy
Architecture: SPADE

The architecture of this was the same as Experiment set 16, but a heavy discriminator was used, containing more parameters (2.1 million) than the generator itself (1.8 million). The reasoning behind this configuration was to tackle the stagnation of learning in previous models, where after a certain point, the discriminator struggled to learn the nature of the grass any further. Dropout regularisation could have been explored for this, but this set of experiments was cut off before that could be tested, due to time constraints.

Result: It wasn't trained for as many epochs as needed to make conclusions, but at a glance, the stagnation and fluctuation problem did not seem to be fixed

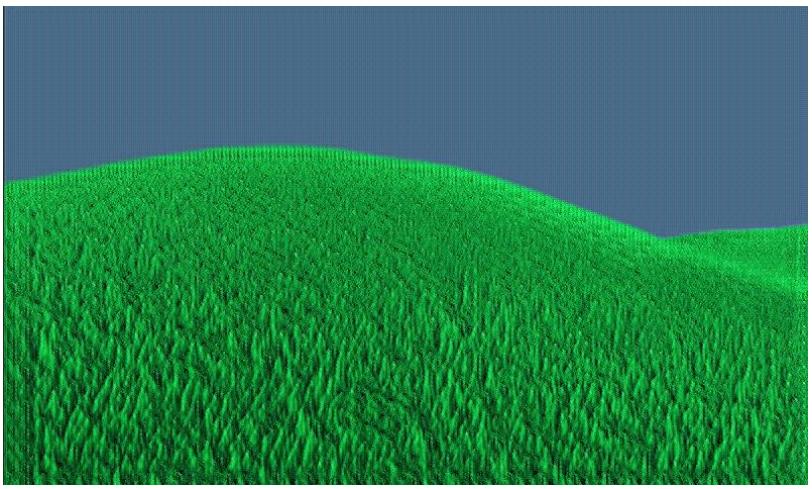


Fig: One of the better results of this model

Insights: Since the heavy discriminator took a lot of time in the iterations, it was decided that future experiments would be conducted with the reliable, lightweight discriminators.

7.1.20. Experiment set 20 (codename: Kraken)

Dataset: Version 5
Discriminator: Multi-level, lightweight
Architecture: Hybrid

This model set out to test the capabilities of the simple hybrid model tested in Experiment set 15, with only a single SPADE normalisation layer, but with all of the improvements learned from the experiments so far. It was also made heavier, having two ResNet blocks in between the upsampling convolution layers instead of just one (with the first among the two last ResNet blocks being the Half-Spade block).

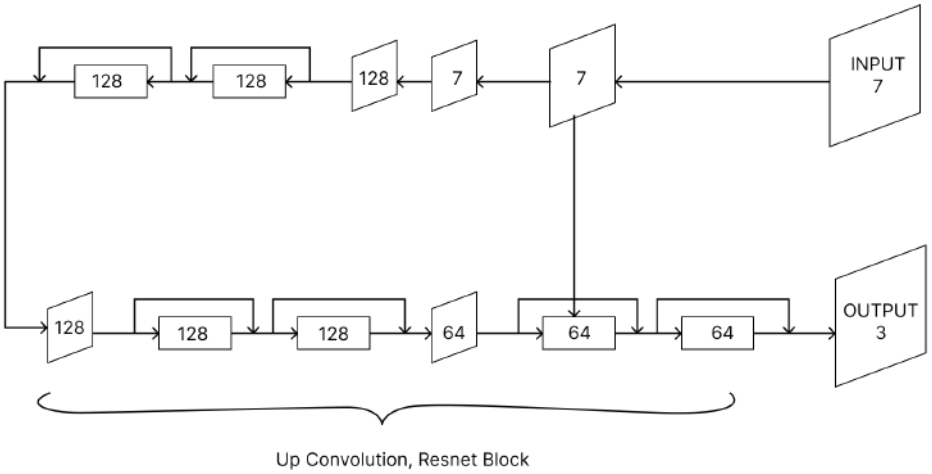


Fig: Architecture of this model

This was another highly branching experiment, since it showed good promise. With 1.8 million parameters, it managed to contain a relatively small number considering how many

layers it has, thanks to all the SPADE layers it dropped, which was the motivation behind this set of experiments. In the main line of the training, it was trained for 50 epochs with a small version of the dataset, and then for 10 more epochs with the full dataset.

Results: It was very fast at training, showing grass within epoch 8. After that point however, it started showing signs of the aforementioned stagnation and fluctuation problems. The training slowed down heavily due to this, with the quality stepping back and effectively restarting many times. At around epoch 50, the quality seemed to be quite good, which improved after training with the full dataset. The best images produced by this model have been the best so far in the project.

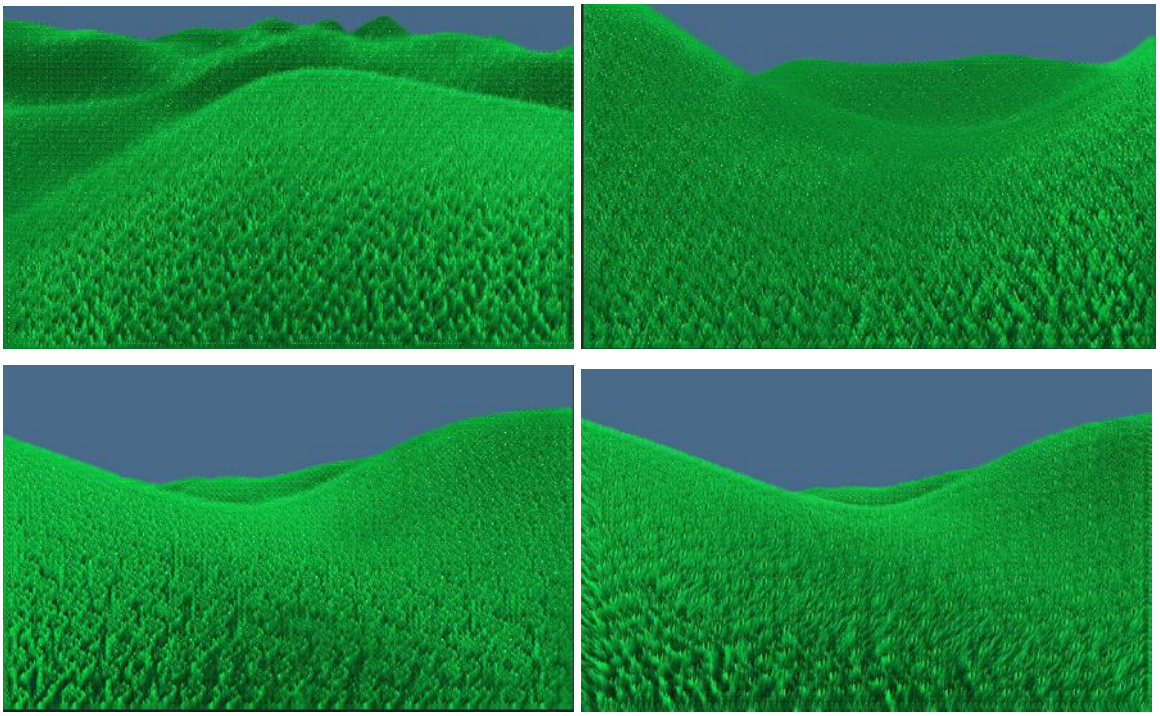


Fig: Results produced by the experiment at epoch 8 (top-left) having learned surprisingly quickly; epoch 18 (top-right) not being that much better; epoch 19 (bottom-left) showing some interesting patterns never seen before and again, and thus displaying the fluctuating nature of the quality; and epoch 35 (bottom-right), showing some improvements

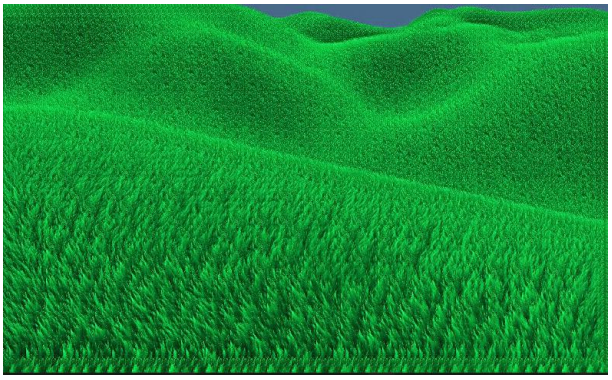


Fig: the result after epoch 55, being the best result seen so far

Insights: This model was considered a success, but the stagnation and fluctuation problem was seen as a huge barrier. Approaches such as CycleGAN are considered to fix it in future experiments. In the testing space (Neural Rendering Playground), it produced worse looking results than seen in the generated images during training, hinting at the fact that a larger (or augmented) dataset was required for future experiments.

Since this experiment was conducted towards the end of our timeline of the project, it was taken up for the demonstration.

7.1.21. Experiment set 21 (codename: Galleon)

Dataset: Version 5
Discriminator: Single-level, heavy
Architecture: Hybrid

The architecture of this model is very similar to the previous experiment (Experiment set 20). The main point to explore here was using a very heavy generator and a very heavy discriminator, to test the limits of how much the model can learn, with no regard to speed. The final model in this turned out to have about 10 million parameters in the generator, and 2 million in the discriminator.

Results: This produced the best results seen in the project, but not by far. It certainly produced the sharpest images, but the quality of the nature of grass itself was not that far away from the best results seen previously. Since Experiment set 20 seemed to slightly overfit to the dataset, these results were certainly expected to as well. However, it held up quite well in the Neural Rendering Playground, except for the fact that it ran very slowly, at about 3 frames per second.

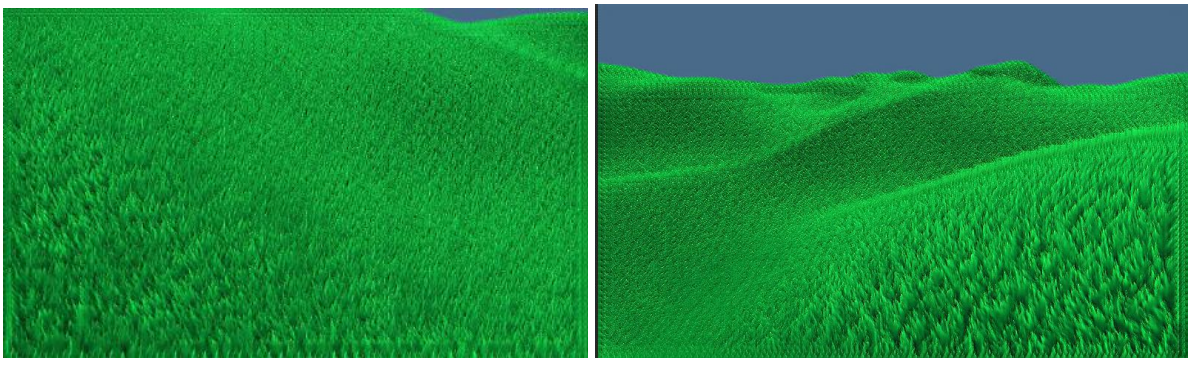


Fig: Images produced by the model

Insights: Since this experiment was not a direct follow-up of others, direct comparisons are difficult to make. Perhaps it shows the usefulness of a heavy discriminator in the right circumstances. Perhaps it shows that even with a large number of parameters, problems like

static white dots and grass-to-sky borders can persist, and thus require a different solution. Analysis of these results are still in progress, and will provide helpful insights for the proper research work that will be done in the future as a continuation of this project.

7.2. Appendix B: Requirement Analysis

7.2.1. Snapshots of Requirement Analysis phases used in this project

Three of the Requirement Analysis phases from our project have been covered in this case study, which shall henceforth be referred to as Initial, Midway and Recent analyses in order of time. The Initial analysis was done just after the feasibility study of the project for which both the functional and non-functional requirements were speculated by the team. For the rest, the functional requirements were defined by the task-holders for the research trials, while the nun-functional requirements were based on those by the task-holders for the environment development. The Midway analysis was done after some period of development and covers the requirements in more detail and with higher accuracy. The Recent analysis was done a week before the preparation of this case study. Each subsequent analysis section below only covers specific changes as well as entries not present in the previous analysis section.

Functional Requirements

Initial Analysis

Playground Environment Development

A. Infinitely navigable environment, using a first-person camera view
B. Randomly generated terrain
C. Procedural grass generation for reference
D. Live creation of training data
E. Ability to load neural rendering models
F. Ability to run neural rendering models in the GPU
G. Display of the output of the neural rendering model on the screen
H. Real-time running of developed neural rendering model

Research Development

A. Loading of a dataset
B. Batching of a dataset for GPU based optimisation
C. Logging of required changeable values
D. Visualisation of logged values in form of graphs or charts
E. Ability to run neural network models for both forward and back propagation

F. Ability to package and save neural network parameters
G. Ability to export neural network models and saved to a format recognisable by the environment

Midway Analysis Changes and Addendum

Playground Environment Development

A. A third-person camera view with an animated character (instead of a first-person camera view)
B. Dataset generation also containing the segmentation map, optical flow map and the normals map (along with the aforementioned view map and depth map) along with certain vector data (position, velocity, rotation, angular velocity and lighting direction)
C. Dataset size of at least 20,000 images
D. Display of the output of the neural rendering model on the screen, as well as a split-screen view with all of the maps mentioned in point 2, as well as live windows to the vector values

Research Development

A. Movement and decompression of dataset files
B. Ability to load dataset items from secondary memory during training and validation, instead of loading all items initially into primary memory
C. Ability to edit all hyperparameters easily in a detached configuration dictionary
D. Saving of trained parameters in an automatic regular way in case of crashing
E. Ability to run neural network modules on command for single provided inputs, for subjective visual validation
F. Ability to easily edit the architecture of the neural network modules
G. Ability to export neural network models and their parameters to onnx files, providing all changes in tensor dimension order and axis directions

Recent Analysis Changes and Addendum

Playground Environment Development

A. A third-person camera view without the character (which will instead be overlaid)
B. Dataset generation only containing the view map, depth map and normals map, along with the aforementioned vector data
C. Dataset size of 8192 items

Research Development

A. Ability to run multiple models in forward propagation for the same inputs for comparative analysis

Non-Functional Requirements

Initial Analysis

Playground Environment Development

A. Navigation axes covering spherical coordinate based Camera rotation, movement in four directions (forward, backward, left and right)
B. Sphere-to-polygonal-mesh collision detection to prevent clipping
C. Random terrain generation based on layered perlin or simplex noise, with at least two layers
D. Visible terrain up to a distance equivalent to 100 meters
E. Changeable scale random subdivision of each terrain polygon to provide vertices for procedural grass generation
F. Shaders providing view map and depth map within a hundredth of a scaled frame-time
G. Procedural grass generation as billboard grass upon each vertex, with shadows casted on one another
H. Live creation of training data and inter-process transmission of it to the python program to enable over 8 items to be processed per second
I. Significant use of GPU
J. Simulation of a neural network model with required operations: convolution, concatenation, element-wise multiplication, addition and subtraction
K. Real-time running of developed neural rendering model with an fps of at least 10 (at most 100ms per frame rendering)

Research Development

A. Loading of the required part of the dataset into primary memory
B. Batching with overhead of less than 10ms
C. Logging taking negligible overhead time, less than 1ms per iteration
D. Encapsulation of changeable tensors and warnings for unauthorised attempts to change them, for catching errors
E. Neural network parameters saved with each value in at least a 16 bit format

Midway Analysis Changes and Addendum

Playground Environment Development

A. Natural movement and rotation of a game player simulated using two components, player and head, with yaw controlled by parent object player and roll controlled by child object head (onto which the camera is attached)
B. Only forward movement required

C. Different distance limits for terrain generation, and procedural grass generation
D. Procedural grass as every individual blade (instead of billboard grass) for realistic simulation of wind and shadows
E. Movement of procedural grass as per a wind-defining image
F. Image loading and saving with overhead of less than a hundredth of a scaled frame-time
G. 10,000 item dataset generation within 90 minutes - requiring a minimum of around two items generated per second
H. CSV file loading, saving and appending with an overhead of less than a hundredth of a scaled frame-time

Research Development

A. Decompression of files into the relevant storage
B. Overhead of creating a batch by reading input images from secondary storage should be less than 10ms
C. Saving of trained parameters every epoch to permanent secondary storage (in our case, google drive)
D. Ability to change kernel sizes, stride lengths, normalisation types and channel numbers of CNN layers through the hyperparameter dictionary
E. Saving of validation set inputs forward propagated through the generator network in full size in permanent secondary storage every given number of iterations (default 50) for subjective visual validation
F. Same as point 5 but for training set outputs for subjective progress visualisation

Recent Analysis Changes and Addendum

Playground Environment Development

A. Ability to overlay two camera pixel grids with an overhead of less than one hundredth of a scaled frame-time
B. Debug screen with pure red, blue and green images (reminiscent of a tricolor flag), leaning backwards at an angle on a plane, providing a consistently increasing value in the depth-map image going from bottom to top
C. Downscaling and displaying of tensor values of inputs provided by the debug screen

Research Development

A. Concatenation of input channels done within the model instead of in the environment

7.3. APPENDIX C: Examples of Environment-Research Feedback

a. Changing inputs

Initially, only view and depth maps were thought to be provided for inputs. Running experiments during the research development presented issues such as the model having to guess which direction the grass must be facing, and which direction the view must be moving. Thus, normals-map and optical-flow-map were also provided along with vectors defining movement. This however made processing too slow, and thus further experiments were done to determine the most important inputs, and thus, most recently, only view, depth and normals were provided as the inputs.

b. Changing training dataset size

Various neural network models were created and tested for both generation and discrimination of generated outputs. As more and more experiments were conducted, the requirement for training dataset size decreased, as the model was performing well even with smaller dataset sizes. As large training datasets require more time to create and load into the research notebook storage space, the requirements for it were relaxed.

c. Type of procedural grass

Initially, the industry standard of billboard grass was considered for the reference grass procedurally generated as the target images for the training dataset. Billboard grass being its own compromise however, resulted in the model learning those compromises and producing artefacts. Thus, the reference grass was changed to be built with every single grass blade being generated separately, reducing those artefacts.

7.4. Appendix D: UML diagrams

The UML diagrams were designed during the first design phase of the project, which was after the first requirement analysis phase. The design of the diagrams went through several alterations, for the most part informally. The diagrams presented in this case study are based on the initial diagrams, with the alterations done after the *midway* requirement analysis included.

UML Use Case diagrams

Environment Development

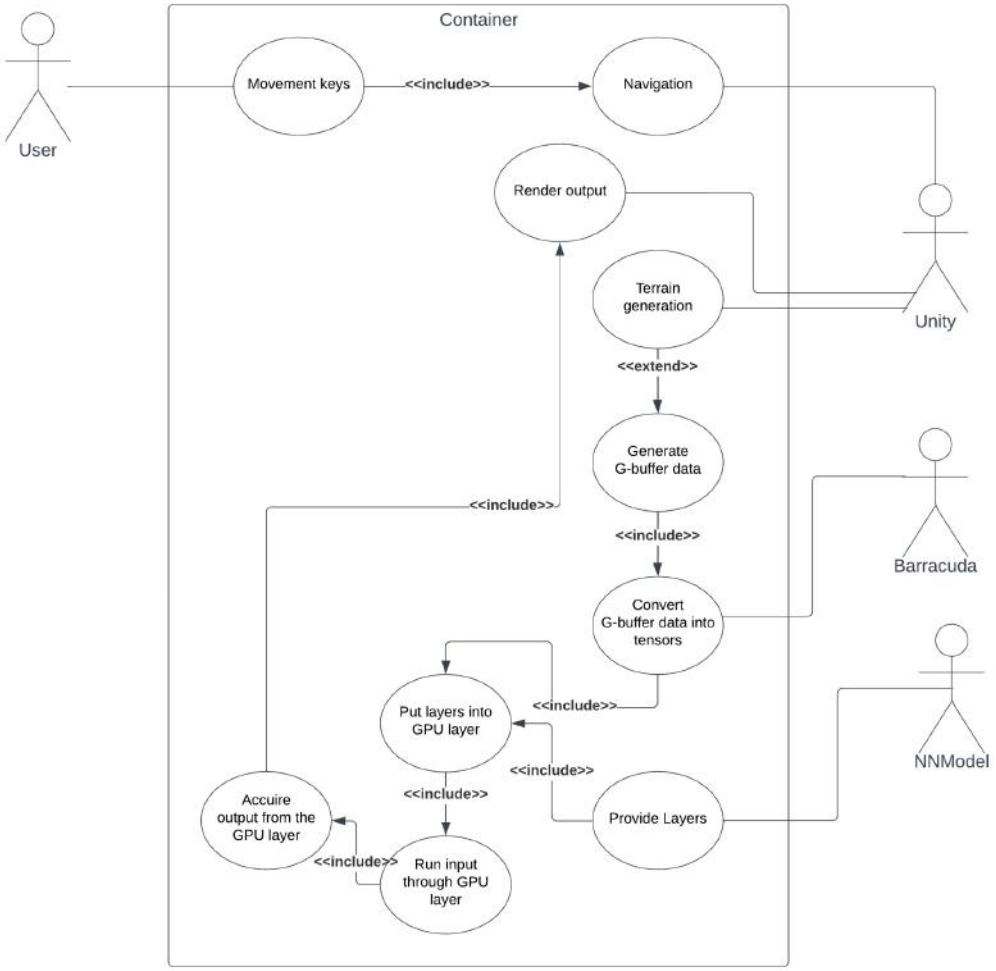


Fig: Use case diagram for environment development

Research Development

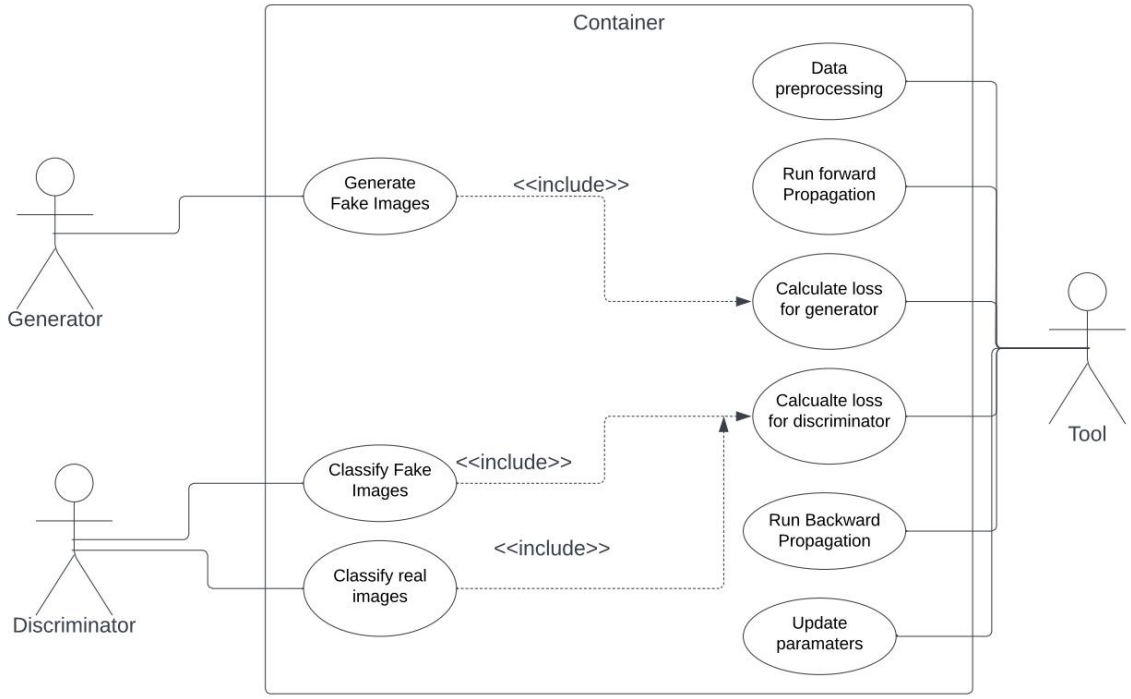


Fig: Use Case diagram covering the roles of Generator and Discriminator, and the Tool

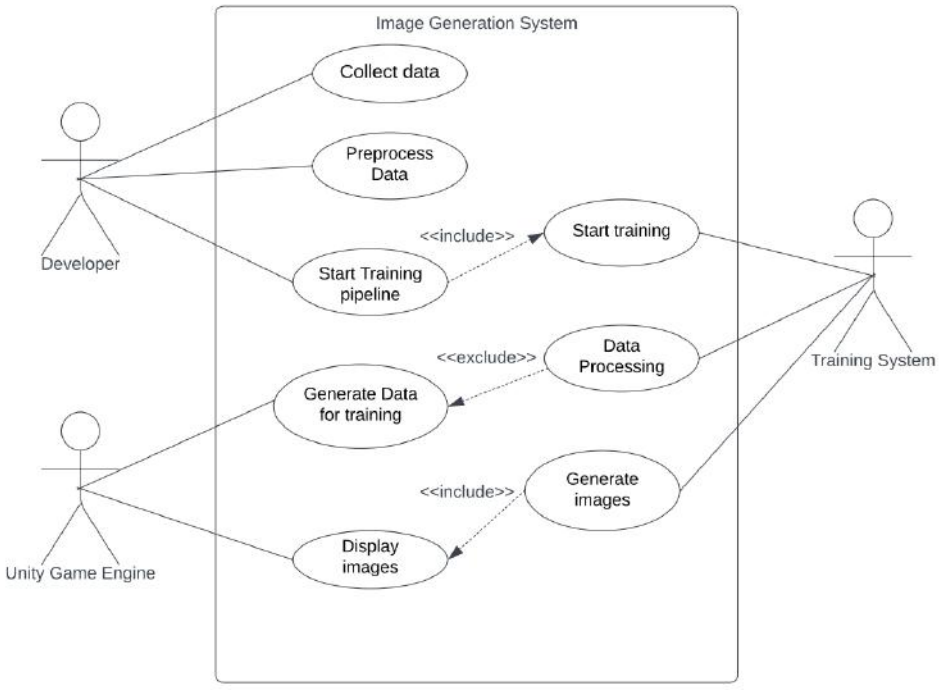


Fig: Use case diagram covering a more general use case

Retrospective Analysis

The ‘tool’ aspect of this was derived from examples found online, since it turned out to be useful

UML Activity diagrams
Environment Development

Fig: activity diagram for a since frame during running a neural rendering model

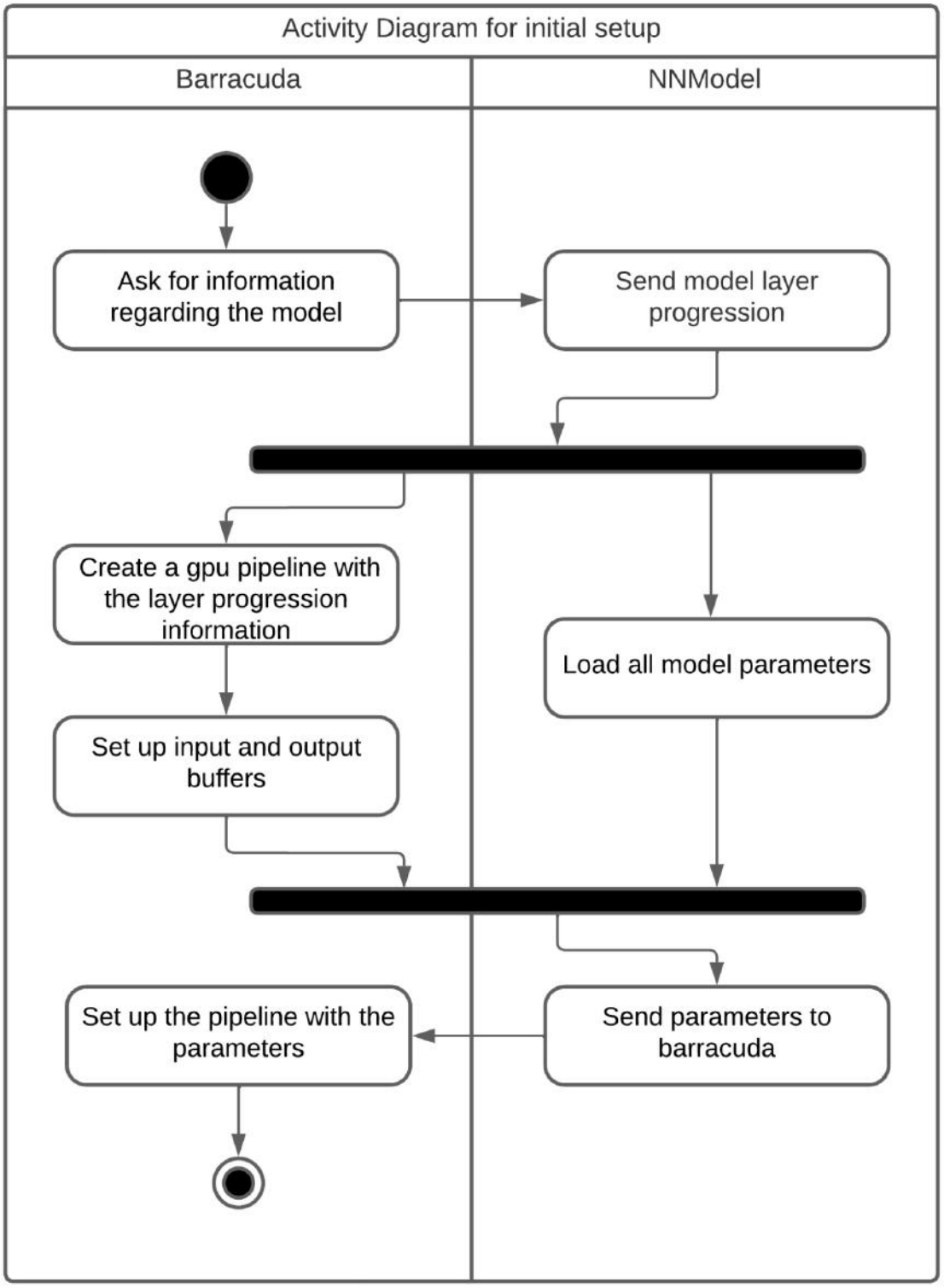


Fig: activity diagram for the initial setup before running a neural rendering model

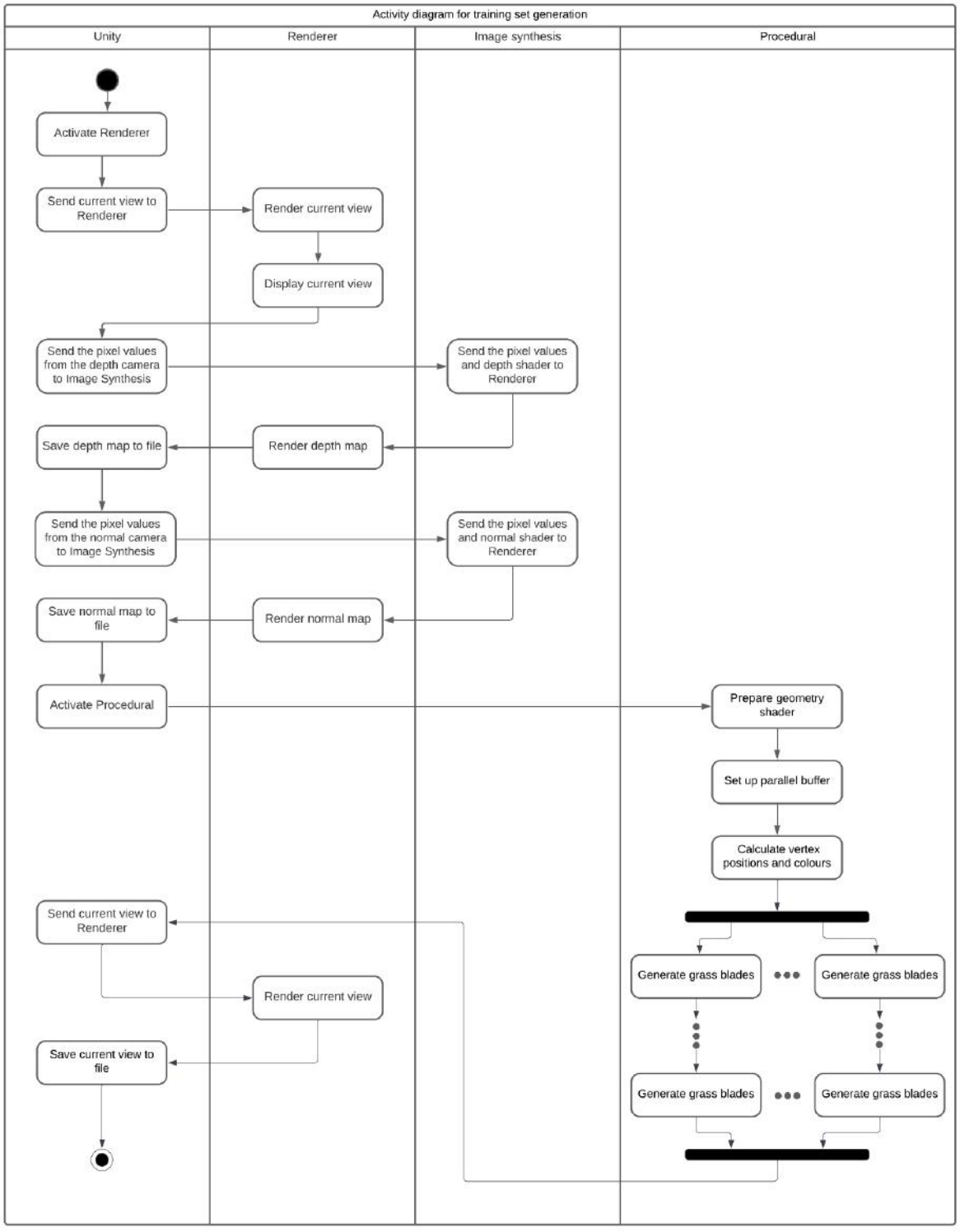


Fig: activity diagram for training dataset generation

Research Development

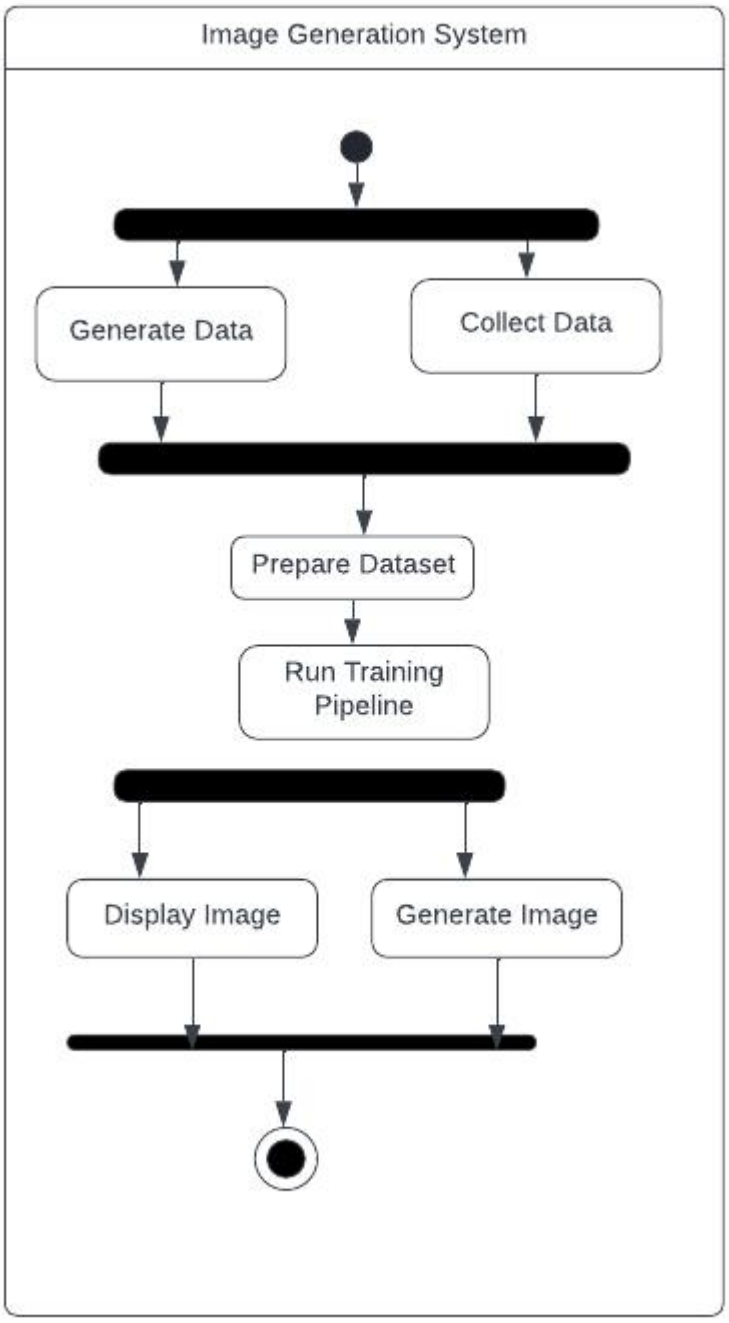


Fig: Activity diagram for Image Generation System

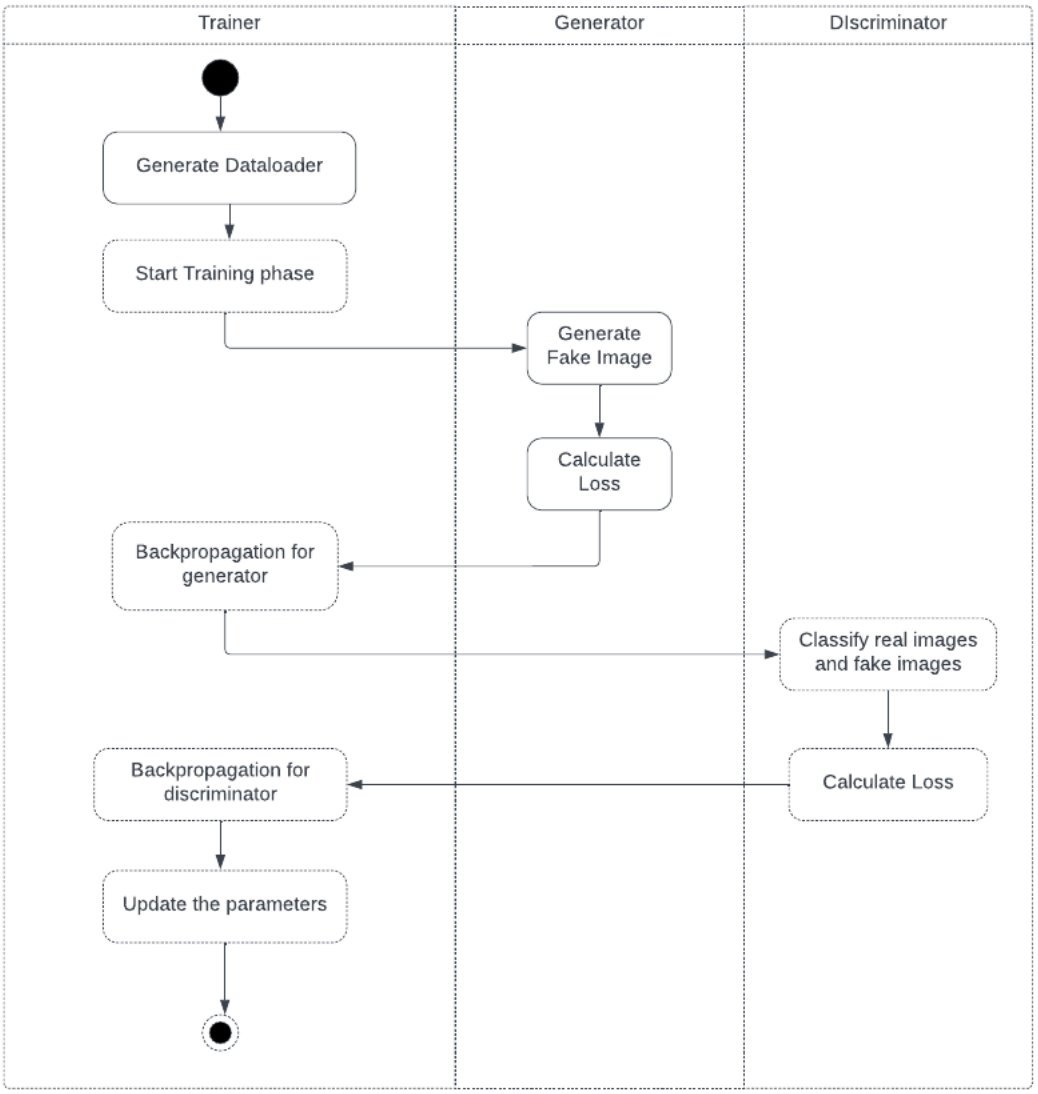


Fig: Activity Diagram

UML Sequence diagrams

Research Development

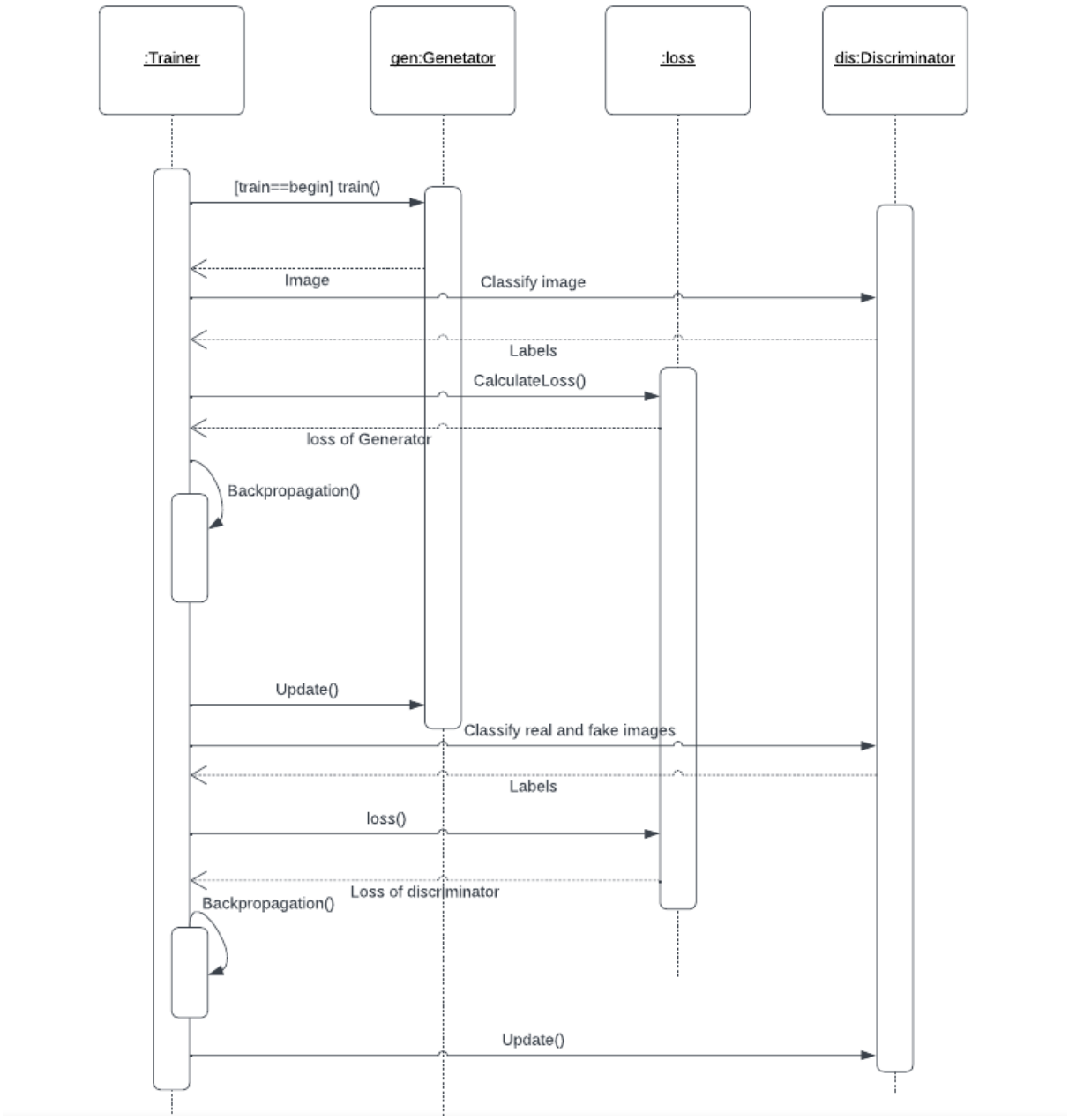


Fig: Sequence diagram for training

UML Class diagrams

Environment Development

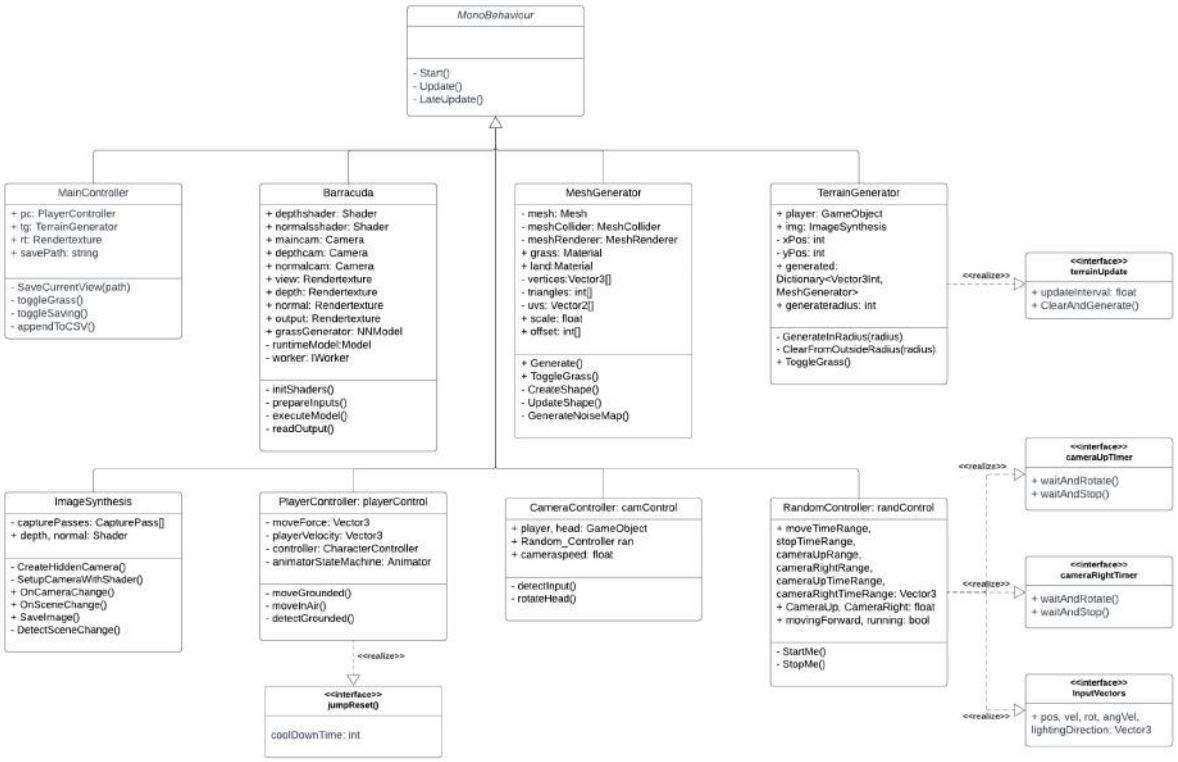


Fig: The classes involved in the Unity environment for the project

Research Development

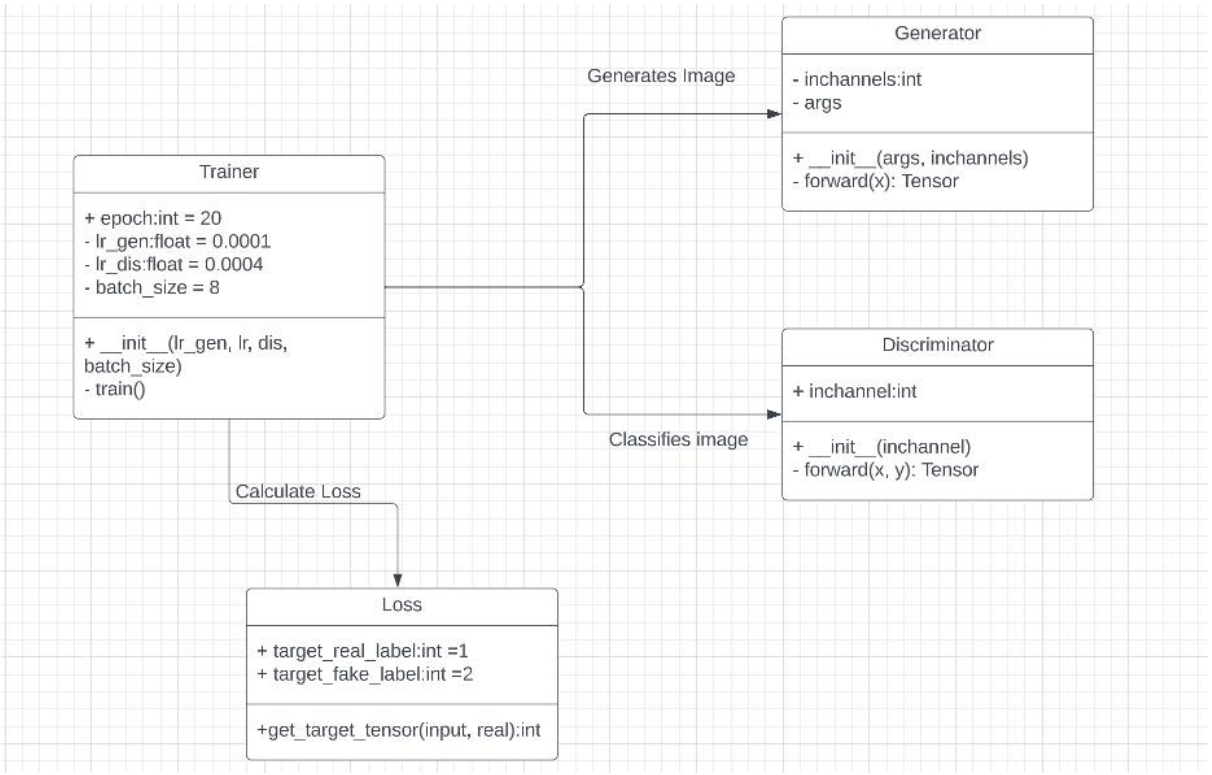


Fig: Class diagram for the machine learning components