ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

Terrain Generation with Generative Adversarial Networks

Graduation Project Final Report

Ekin Zuhat Yaşar - 150160142
Matthew Oğuzhan Dirlik - 150160123

Department: Computer Engineering
Division: Computer Engineering

Advisor: Assoc. Prof. Dr. Uluğ BAYAZIT

January 2022

Statement of Authenticity

I/we hereby declare that in this study

1. all the content influenced from external references are cited clearly and in detail,
2. and all the remaining sections, especially the theoretical studies and implemented software/hardware that constitute the fundamental essence of this study is originated by my/our individual authenticity.

January 2022

Ekin Zuhat Yaşar, Matthew Oğuzhan Dirlik

Terrain Generation with Generative Adversarial Networks

(SUMMARY)

Terrain generation is a vital task in the process of world building or level design in the digital environment. It is the foundation for finer details and stories to be built upon, thus carrying vital importance. Rendering a world from scratch is a costly task and takes excessive development hours to generate adequate terrain. Our aim is to be able to generate terrain through the use of generative adversarial networks (GAN) as defined by Goodfellow [1].

Generative adversarial networks compete with each other in order to generate fitting terrain from the given data sets, saving developer hours in any project it is utilised in [2]. The GAN will generate superficial terrain from the data supplied that will be serviceable as terrain, while also remaining similar to the input data. Using these implicit data models will also allow us to form more realistic terrain than any hand modeled version, as through the use of real world data the formation will resemble nature and avoid repetition issues that can accompany hand drawn terrain.

Provided a GAN is modified to the correct extent, it can be a useful tool in aiding the process of world generation, removing the need for manual input. A data set of images, in the case of this report height maps of various locations, can be utilised in order to generate 3D maps with ease. Formatting the resulting output into a rendered object model remains a menial task, while yielding comparable results to the traditional method.

Terrain Generation with Generative Adversarial Networks

(ÖZET)

Yüzey üretimi dijital bir ortamda sanal dünya inşaa ederken yapılan kilit bir işlemdir. Üzerine detayları işlemek ve bir hikaye anlatmak için alta atılması gereken temeli çözmeyi hedefler. Sıfırdan sanal bir dünya üretmek otomasyon kullanmadan yapılırsa kullanılması istenebilecek bir yüzey üretimi çok fazla çalışma saatine tekabül eder. Bu projede hedefimiz Goodfellow[1] tarafından tanımlanan bir Generative Adversarial Network(GAN) dizayn etmek, onu elde ettiğimiz yükseklik haritası fotoğraflarıyla eğitmek ve sonuçlarını incelemek.

Generative Adversarial Network'ler elimizdeki veri setini daha iyi üretilmiş bir sanal dünya ile temsil edebilmek için birbirleriyle devamlı bir yarış halindedir. Bu sayede birçok çalışan mesaisinin ayrılması gereken bir işi önceden eğitilmiş bir modelin sonuçlarından çok daha hızlı elde edebiliriz[2]. GAN sağladığımız ürettiği sanal yüzeyi verdiğimiz veri setinden uzaklaştırmadan üretebilir. Dünya üzerinden alınan tarafsız ve gerçek verilerden yola çıkarak üretilen yüzeyler bir insanın taraflı bir çiziminden daha gerçekçi sonuçlara ulaşmamızı sağlar. Gerçek veriler gezegenimizin doğasını daha iyi temsil eder ve kısıtlı insan bilgisininin yol açtığı tekrarlamaları minimuma indirir.

GAN üzerinde yapılan modifikasyonlar yeterli ise yüzey üretiminde kullanışlı bir araç olabilir, hazır bir veri setinden beslenerek otomasyon ile ilerleyebilir. GAN çıktısı ise gerekli çeviri programları ile 3 boyutlu yüzeylere kolayca çevrilebilir.

Contents

1 Introduction and Project Summary

Terrain generation is one of the bases of any ambitious project, be it in the gaming or film industry, that features in a large or open world. The importance of having variable terrain to enhance the user experience can not be understated, with the main feature of the world's top selling video game of all time Minecraft [3], being based around the player exploring the world that is procedurally generated for them. Recent advancements in technology have further expanded on this idea, with games like No Man's Sky having a near-infinite universe containing worlds that are procedurally generated, player unique and fully explorable.

Figure 1.1: A screenshot of a procedurally generated world in 'No Man's Sky'

Source: Hello Games website [4]

The usage of procedural terrain generation tools has created a large impact on the gaming industry as evidenced by the aforementioned examples, and is growing in influence in the media industry following the success of terrain generation in films such as The Golden Compass and Mission Impossible through the use of tools such as Terragen [5]. Prior to this approach worlds would be crafted by hand through a painstaking process, involving teams of both artists and software engineers, that rendered larger scale projects impossible due to time or budget constraints.

One of the main components used when creating any 3D terrain or object is a heightmap. A heightmap is a 2D array of pixels, with each pixel having a corresponding value, that being it's height from the surface. Upon viewing a heightmap the value for each pixel is transformed into it's corresponding value on gray scale, with black being the lowest possible depth, i.e. surface level, and white being the highest possible ceiling. To generate a terrain object from a heightmap, a and transformation into three dimensions is required, which can then be textured. An arbitrary scale, the larger the scale the bigger the height difference between black pixels and white pixels, is selected for the heightmap and a terraforming algorithm is applied to generate the corresponding 3D object. Once a 3D object is obtained it can then be textured manually, have a texture

file overlaid using graphics software, or could even be textured based on input images using a GAN, such as the concept explored by Beckham and Pal [6] in their 2017 study.

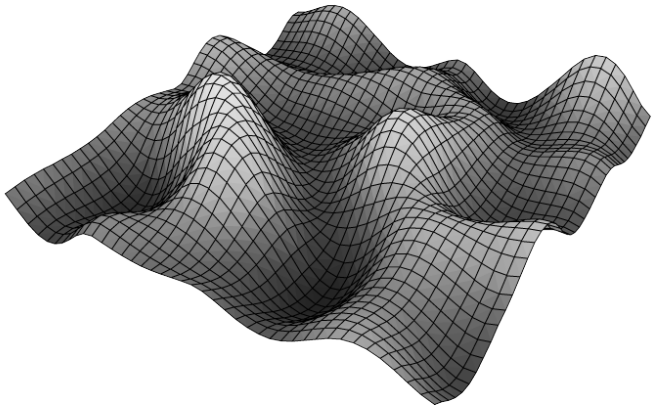

Figure 1.2: An example 3D terrain mesh

Given that terrain generation, adaptation and texturing can be such a complex and time consuming process, the potential benefits to automating the process using deep learning methods are evident. This study will explore the use of Generative Adversarial Networks defined by Goodfellow [1] to enhance this process, and focus on some of the benefits as well as shortcomings of alternative methods and implementations.

Generative Adversarial Networks (GANs) are a relatively new approach to deep learning and function through the use of two main actors, a generator network and a discriminator network. The GAN is frequently described in lay-man's terms as a cop and robber scenario, being that the generator is constantly generating false data samples with the objective of fooling the discriminator into thinking they are real samples. The discriminator performs the task of distinguishing real data from the fakes, with both training as time elapses. With the two networks working against each other while improving simultaneously, a GAN is able to generate realistic and detailed data that conforms to the original data set used. The main difficulty of using a GAN to solve problems such as this is that the configuration needs to be fine tuned as evidenced in the 2020 study published by Mo [7], as if either the discriminator or the generator begins to overpower the other or finds a flaw to exploit, the results will be unsatisfactory.

2 Comparative Literature Survey

2.1 Terrain Generation Prior to GANs

Human development and manual terrain creation will be able to achieve a level of detail and customisation unlike any neural network available as of publishing this article. Due to this fact it is widely agreed that a tandem use of both generation, to handle the project on a wider scope, and human development to add the finer details will yield the optimal results in the field of terrain generation. The topics discussed in this section will focus on the terrain generation through automated or procedural software.

Prior to the development of GANs and other approaches to terrain generation using neural networks, a common method still utilised today was to generate terrain through the use of random noise functions. The most common of these functions was developed by Perlin in 1985 [8] and is known as Perlin noise. Perlin noise would be generated in one dimension with values in an arbitrary range. The values generated through the use of this algorithm would then be translated into a heightmap, which in turn could easily be shifted into a 3D format to be used as terrain.

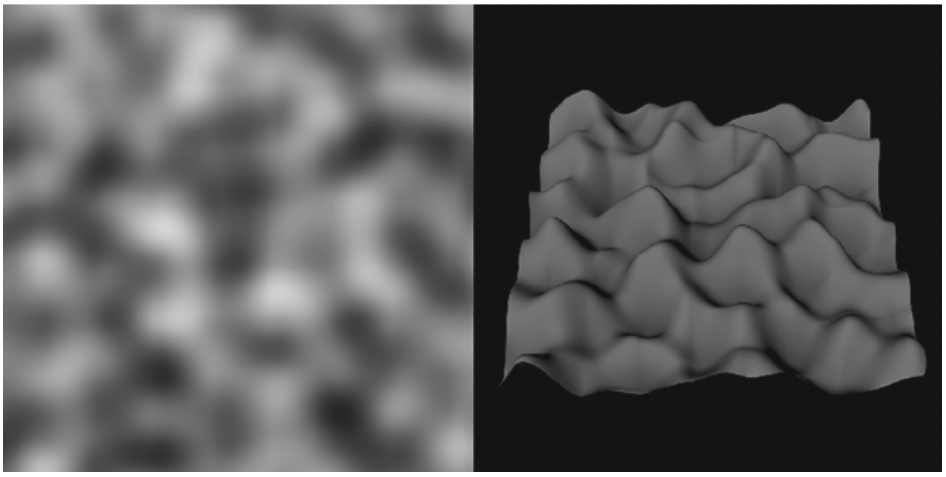

Figure 2.1: A heightmap and it's corresponding terrain generated with Perlin noise

Source: Fig. 1 of [9]

2.2 Generative Adversarial Networks

2.2.1 Basic GAN

The concept of GANs was first introduced through the work of Goodfellow et al [1] in 2014. Goodfellow brought GANs into the discussion as a generative model that could be useful for situations requiring unsupervised learning, however developments have been made that has allowed GANs to see use in systems incorporating semi-supervised, or fully supervised learning [10]. Since their inception GANs have seen use

in a multitude of fields, rising to mainstream fame through the implementation of a StyleGAN2 in Deepfake technology. The malicious potential of technologies such as Deepfakes has even elicited legal response from governments in countries such as the USA [11], with officials aware of the potential of the technology.

The basis of any type of GAN exists in the two neural networks that are in constant competition with each other in a zero sum game. A GAN consists of two main players in the form of a generator network and a discriminator network. The generator network takes a vector of noise of an arbitrary dimension, for example Perlin noise [12], and creates images resembling the original data set. The discriminator takes the generated images, in combination with the original data set and checks the validity of each image, determining it to be real or fake. The idealistic goal of a GAN is to achieve equilibrium, where the discriminator is able to predict an images validity with 50% accuracy. A rudimentary diagram of data flowing through a GAN is given below in figure 2.2. The two networks train each other simultaneously through their relative output and feedback, however one of the modifications we have made to our project is to have each network train at a different rate. We found that this improves the accuracy of results and helps combat one of the networks from overpowering the other for a sensible number of epochs.

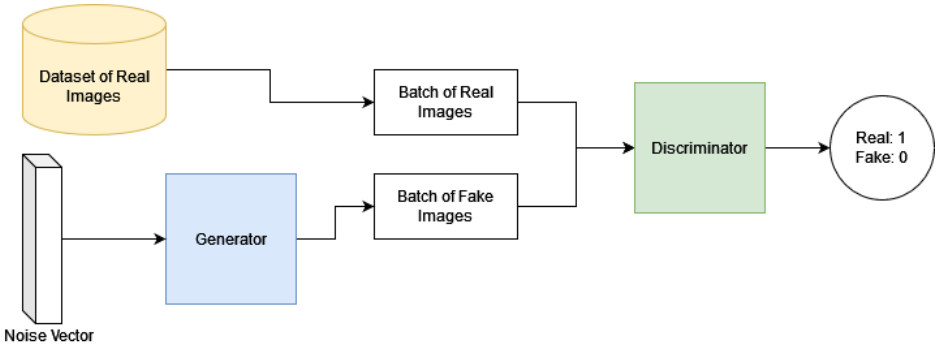

Figure 2.2: Concept diagram of a GAN

2.2.2 DCGAN

Due to the dual network nature of a GAN it is a system that lends itself to modification, thus there are a multitude of configurations. The most common implementation due to it's noticeable improvement when used with an image dataset is a Deep Convolutional GAN (DCGAN), introduced by Radford et. al in their 2016 paper[13]. Implementations of multiple GANs in consecution has also been utilised to great success in projects where both form and style are important, as evidenced in the 2017 study by Zhang et al. [14] where text can be transformed to photo-realistic images using two networks. A DCGAN differs from a simple GAN in form through the use of a Convolutional Neural Network (CNN) to replace the maxpooling process with connected layers, and the application of an activation, such as a sigmoid layer at the last layer of the discriminator [13].

Both the discriminator and the generator network share the same goal, which is to

minimize their respective loss against the other. The discriminator achieves this by trying to maximize the average log probability of real images, while minimizing the average log probability for those that are fake. This is translated into the mathematical formula below:

$$\max_{\theta} E_{x \sim p_{data}}[log D_\theta(x)] + E_{z \sim p(z)}[log(1 - D_\theta(G_\theta(z)))]$$

Equation 2.1: DCGAN Discriminator Goal

The generator decreases it's loss through the generation of more realistic and undetectable images. Thus, it will aim to minimize the log of the inverse probability that the discriminator predicted for a given fake image. This is translated into the mathematical formula below:

$$\min_{\theta} E_{z \sim p(z)}[log(1 - D_\theta(G_\theta(z)))]$$

Equation 2.2: DCGAN Generator Goal

This project aims to help both networks meet these goals through our configuration of the DCGAN, as well as the implementation of different learning rates for the discriminator and generator network.

Our implementation is based on the original DCGAN implementation, through the use of Leaky Rectified Linear Unit (Leaky ReLU) as an activation in the generator network, and convolution for the image in the discriminator network. We based our approach on this through gradual upscaling of the convolution layer by factors of 2, until we reached the desired channel of the output heightmap which for our implementation was 256x256x1.

Image to image translations are the most common use case of GAN models, with the leader in this area being the Convolutional GAN model known as 'pix2pix' formulated by Isola et al. in their 2017 study [15]. Beckham and Pal implemented a modified version of pix2pix in combination with their own DCGAN in order to add texture to procedurally generated maps in their 2017 paper [6], with the pix2pix GAN using data as pairs, heightmaps as well as textures rather than checking for a single variable. This study proved that for projects of a larger scale it is possible to stack GANs in tuples, however further research is required to discover the point that this method reaches diminishing returns in terms of training time. Our implementation is comparable to the first stage DCGAN used in this project, with the main difference coming in the batch model and convolution methods utilised.

3 Developed Approach and System Model

3.1 Data Model

Creating and developing our own data set had its own challenges. We utilized Selenium WebDriver to scrape a raw data set, which we further developed by taking each picture's individual pixel value and calculate the standard deviation. In the following subsections we will give relevant information about the topic and explain our process through our development of our data set.

3.1.1 Introduction to Heightmaps

Heightmaps are the pictures in which each pixel holds the data of elevation as its color value. Most common heightmap is a grayscale, upside down image of a surface usually selected randomly or knowingly from any part of the Earth. The scale going from black to white represents the elevation data of that exact coordinate represented with the pixel. A point represented by a pixel is always elevated more than the points represented by darker pixels, and of course elevated less compared to the points represented by brighter pixels. It would be worthwhile to note that for a more accurate representation, one may utilize colored heightmaps, which are not grayscale. This would mean that instead of values between 0 and 255, we could utilize 3 bytes representing RGB. This would greatly improve the accuracy of a heightmap but in turn would make the data set a lot heavier and affect performance of any GAN.

For our project we have utilized grayscale heightmaps with PNG image format that we have scraped from a web project, which we further specify in the next subsection. This project we scraped from aimed to create heightmaps for the game Cities: Skylines, in which players have the freedom to import their own heightmaps which are rendered into game levels for a more immersive and unique player experience. The game is about building and managing a city including but not limited to topics like infrastructure, housing, public services(schools and hospitals alike), means of production and many more. Cities: Skylines gives players the option to import and use any place on earth, which is possible with importing heightmaps to the game.

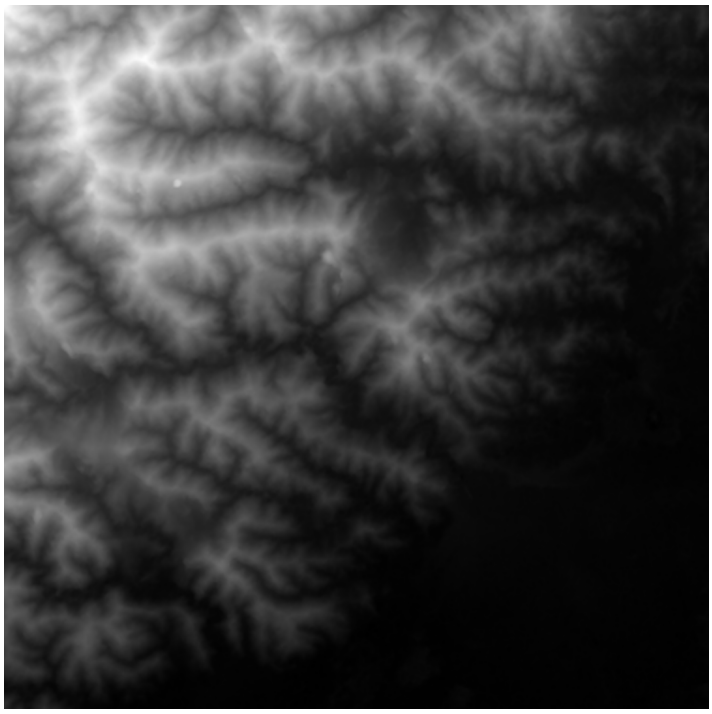

Figure 3.1: A Heightmap example from our web scraper [16]

3.1.2 Developing a Web Scraper Tool for Heightmap Data Set

Normal process to select and download a heightmap would be manual labor. It would be an excessive task to obtain a hundred or a thousand let alone tens of thousands of valid heightmap images from all around the Earth. To obtain this, we developed a web scraper we call our Map Scraper. We utilize an open source browser automation tool called Selenium. Selenium includes a web framework called Selenium WebDriver, which has a range of tools that helps us automate browser related tasks. Since manual laboring a data set creation would not be sensible.

We instead developed a web scraper, which for this project is hand built to work with our selected heightmap tool[17] that also has a deployed website[18]. It would likely be an easy task to modify the tool to utilize any other heightmap generator source. The values of website elements we pass to our Selenium WebDriver would need to change.

Our scraper has multiple parts and we want to explain it in a little bit detail. Below function utilizes MatPlotLib library's Basemap[19] to filter the coordinates to pass our scraper only longitude and latitude combinations belonging to land. This way we can scrape heightmaps only over land. Considering the percentage of water compared to the land on Earth, this is a necessary filter to apply to our longitude and latitude values.

```
1 def draw_map():
2     bottomlat = -89.0
3     toplat = 89.0
4     bottomlong = -170.0
5     toplong = 170.0
```

```
6     world_map = Basemap(projection="merc", resolution='c',
    area_thresh=0.1, llcrnrlon=bottomlong, llcrnrlat=
    bottomlat, urcrnrlon=toplong, urcrnrlat=toplat)
7     world_map.drawcoastlines(color='black')
8     return world_map
```

Our main function starts with setting up Selenium WebDriver values. We utilized Mozilla's Firefox Browser. These values can also be manually changed by going to URL "about:config" in the browser. Values that set Firefox profile preferences are for utilizing Firefox whereas values given to our WebDriver with variations of driver.find calls are correlated with the website we are scraping with.

```
1     profile = webdriver.FirefoxProfile()
2     profile.set_preference('browser.download.manager.
    showWhenStarting', False)
3     profile.set_preference('browser.download.folderList', 2)
4     profile.set_preference('browser.download.dir', '/tmp')
5     profile.set_preference("browser.download.dir", "C:\\
    heightmaps\\new_heightmaps\\")# intended directory
6     profile.set_preference("browser.download.useDownloadDir"
    , True)
7     profile.set_preference('browser.download.
    improvements_to_download_panel', True)
8     profile.set_preference('browser.helperApps.neverAsk.
    saveToDisk', "image/png")
9     driver = webdriver.Firefox(profile)
10    driver.get("https://heightmap.skydark.pl")
11    search_button = driver.find_element_by_xpath("//a[@title
    ='Set lng/lat']")
12    save_button = driver.find_element_by_xpath("//a[@title='
    Download PNG height map']")
13    lng_field = driver.find_element(By.ID, "lngInput")
14    lat_field = driver.find_element(By.ID, "latInput")
15    apply_search_button = driver.find_element_by_xpath("//
    button[@onclick='setLngLat(2)']")
16    world_map = draw_map()
```

Our loop inside the main function that does all the work is as follows. At the start of every loop, we have a small infinite loop until we can be sure we are sending longitude and latitude values belonging to land and not sea. After that, we clear the value fields, put in new values, search and save the result. After countless testing, we have come to the realisation that we needed some artificial time delay to cope with site loading and hardware processing.

```
1     # i = number of heightmap.png files to search and
    download.
2     for i in range(1000):
3         while True:
4             # generate a random longitude and latitude value
```

```
5             lon, lat = random.uniform(-179, 179), random.
    uniform(-89, 89)
6             # convert to projection map for checking
    purposes in the 'if' below
7             xpt, ypt = world_map(lon, lat)
8             # Check if that point is on the land, we need
    land heightmaps to have height deviation
9             if world_map.is_land(xpt, ypt):
10                # if it is on the land break our 'while True
    ' and go to our 'try:'
11                break
12        try:
13            # clear lng and lat fields to input fresh values
     later
14            lng_field.clear()
15            lat_field.clear()
16            # input fresh lng and lat values to the search
    tool
17            lng_field.send_keys(lon)
18            lat_field.send_keys(lat)
19            # search the intended land with the inputted lng
     and lat values
20            apply_search_button.click()
21            # wait for the searched area to load, otherwise
    we cannot get consistent returns
22            time.sleep(2)
23            # save the heightmap.png to our directory
    selected at the beginning
24            save_button.click()
25            # time to sleep, which helps us to have no
    problem with our downloads
26            time.sleep(1)
27        except:
28            input("loop is thrown out with exception")
```

3.1.3 Optimizing the Data Set

After scraping thousands of heightmap images at a time, we came up with a solution to filter our pool. Getting all images in a list and calculating their standard deviation in relation to the value each pixel hosts leads us to a solution which we found meaningful enough. The value 5000 in the example below was changed throughout because we came to the conclusion that most images below 1500 were unusable at best.Between 1500 and 4500 we have found that it could be a hit or miss, leading us to use 5000 as a standard. We should also clarify that we held on to some of the images belonging to a lesser standard deviation value where we thought they were sufficient enough.

```
raw_images = glob.glob('C:\\heightmaps\\new_heightmaps\\*.
    png')
for raw_image in raw_images:
    pixel_data = Image.open(raw_image).getdata()
    standard_deviation = np.sqrt(np.var(pixel_data))
    if standard_deviation < 5000:
        os.remove(raw_image)
```

After our filtering, we went on to make our data set finally usable for our GAN design. The images we scraped from our source were sized 1081x1081 pixel squared, which proved to be a challenge after our researched indicated that most people developing a GAN utilized data set containing somewhere between 64 or 512 squared images. We first tried to crop 16 256x256 or 4 512x512 images out of them, but we found it lead to a faulty data set even if we further filtered them with standard deviation calculation.

After we decided that 256x256 pixel sized png images were the optimal way, we wrote the simple code below to resize our images that we have filtered with the standard deviation calculation. Simply takes the source images, resizes them and save them into a new directory.

After our processing, we ended up with multiple data sets containing PNG heightmap image files. These data sets are different from each other in standard deviation, image count, image pixel size and so on. We tested our generator and discriminator designs on each one with epoch counts varying between a hundred and a thousand, observing the outcome and analysing the difference and similarities of different generator and discriminator layers.

3.2 Structural Model

3.2.1 Dataset Construction

We fed our image dataset to the generator as a Keras dataset. Our implementation is customisable to any dataset the user may desire to use, provided that all images are of 256x256 pixel resolution. The program extracts the files from a ZIP archive and saves them on the devices local storage for use in the GAN.

The code shown below can be modified to the location of the files on the users computer. To turn this image file into a dataset compatible with Tensorflow, first preprocessing was performed using *ImageDataGenerator*, followed by an import to a dataset using *flow_from_directory*. The preprocessing performed on the file is done in order to normalize the images to a range of [-1, 1]. Normalization is calculated by subtracting and dividing the mean of the pixel values, which range between 0 and 255, equalling 127.5 in the case of images.

```
real_image_paths = list(glob.glob("C:\\heightmaps\\use_these
    \\resized\\newfile\\*.png"))

```

```
3 def change_range(image):
4     return [(i / 128.0) - 1 for i in image[:, :, :]]
5
6 image_generator = ImageDataGenerator(
7     preprocessing_function=change_range,
8     horizontal_flip=True,
9     vertical_flip=True)
10
11 real_generator = image_generator.flow_from_directory(
      real_dir,
12
      batch_size=BATCH_SIZE,
13                                                    shuffle
      =True,
14
      target_size=TARGET_SIZE,
15
      color_mode="grayscale",
16
      class_mode='input')
```

3.2.2 Generator Model Design

The main function of a generator in a GAN is to generate fake images, that look progressively more like the real images as the model is trained. It takes a vector of arbitrary dimensions as input, in our case we used a single vector of a length of 4096. Deconvolution is then applied to his layer until it reaches the desired output dimensions. We started by convoluting and transposing the input vector into a 4x4x256 layer, followed by 5 layers of convolution, batch normalization, an activation layer and finally an upsampling layer. Once the deconvolution layers are complete, a final convolution of a single filter is applied and we receive our generated grayscale image. We perform upsampling and convolution as an effort to remedy the checkerboard artifacts previously discussed in section 4.

Performing these consecutive deconvolution operations allows us to generate the desired 256x256x1 images from a starting vector of 1x1x4096. The full model of the generator detailing the dimension of each layer and the operation performed is seen below in figure3.5.

3.2.3 Discriminator Model Design

Discriminator networks in GANs are tasked with determining whether the image supplied to them is part of the original dataset or not. The discriminator implemented in our project is a simple convolutional classifier, it is fed an input image and performs downsampling until it reaches a probability. In the discriminator we perform the re-

verse of what is done in the generator. The input image is downsampled by the same number of layers. Initially Gaussian noise is added to the discriminator in an effort to improve result quality, following that each convolution layer consists on convolution, activation via Leaky_ReLU, dropout at a rate of 0,25 to generate a wider variety of output, with a final operation of average pooling. Each layer reduces the size of the input from the previous layer, while increasing the number of learned filters. Average pooling helps smooth the output from each layer to a more cohesive image. The final layer is then flattened, after which a sigmoid activation layer is applied in order to output probabilities.

Our generator and discriminator models can be found detailed in figures 3.5 and 3.6 respectively. A key aspect to the functionality is to ensure that the number of layers performed for the generator and discriminator are the same in order to avoid artifacts on the final output.

3.3 Dynamic Model

In this section, we would like to cover our system models and diagrams starting with our overall design and go on to breakdown individual components with multiple tools inside them.

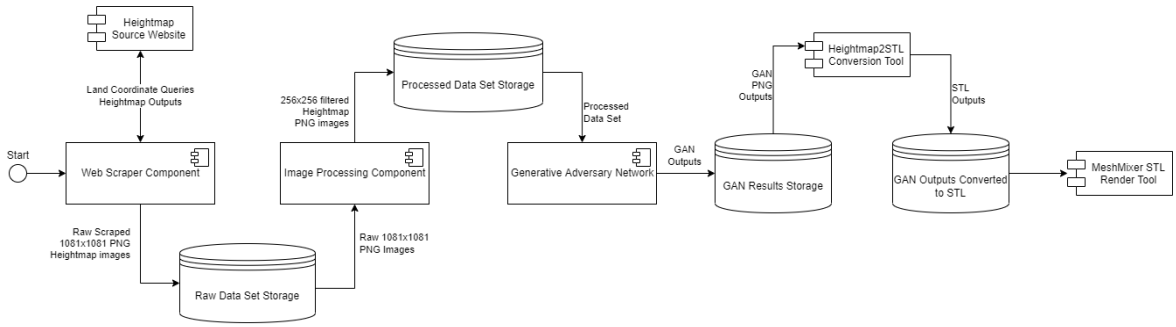

Figure 3.2: Overall System Model

Our overall system design is as in Figure 3.2. We further explained the parts that we shaped as components. Some parts are outsourced, for example Heightmap2stl module[20] is an open sourced image to STL file conversion that achieves that utilizing Java. STL is a format that is ready for 3 dimensional rendering.

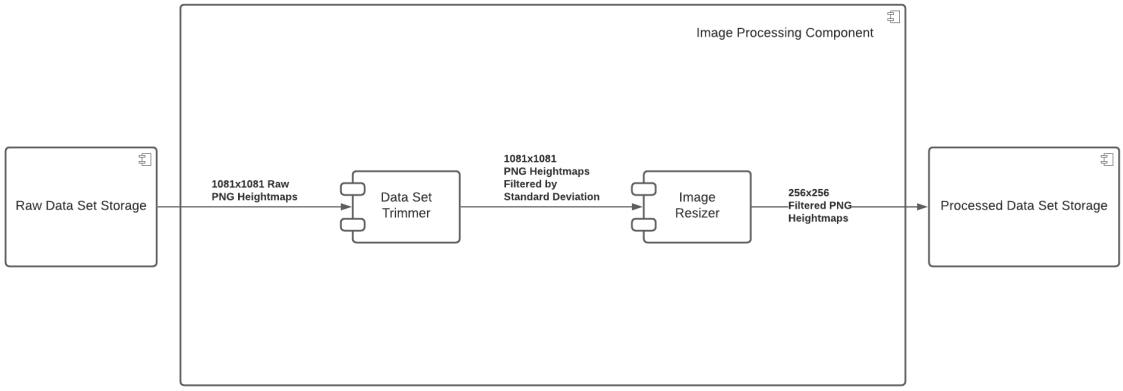

Figure 3.3: Image Processing Component

Image processing component could be broken down as it can be seen in Figure 3.3. This component takes in our raw data set and filters them with our Trimmer module, which in turn calculates and filters the raw pictures with standard deviation calculation. Newly trimmed data set is sent to resizer module to and sent out as 256x256 sized png files.

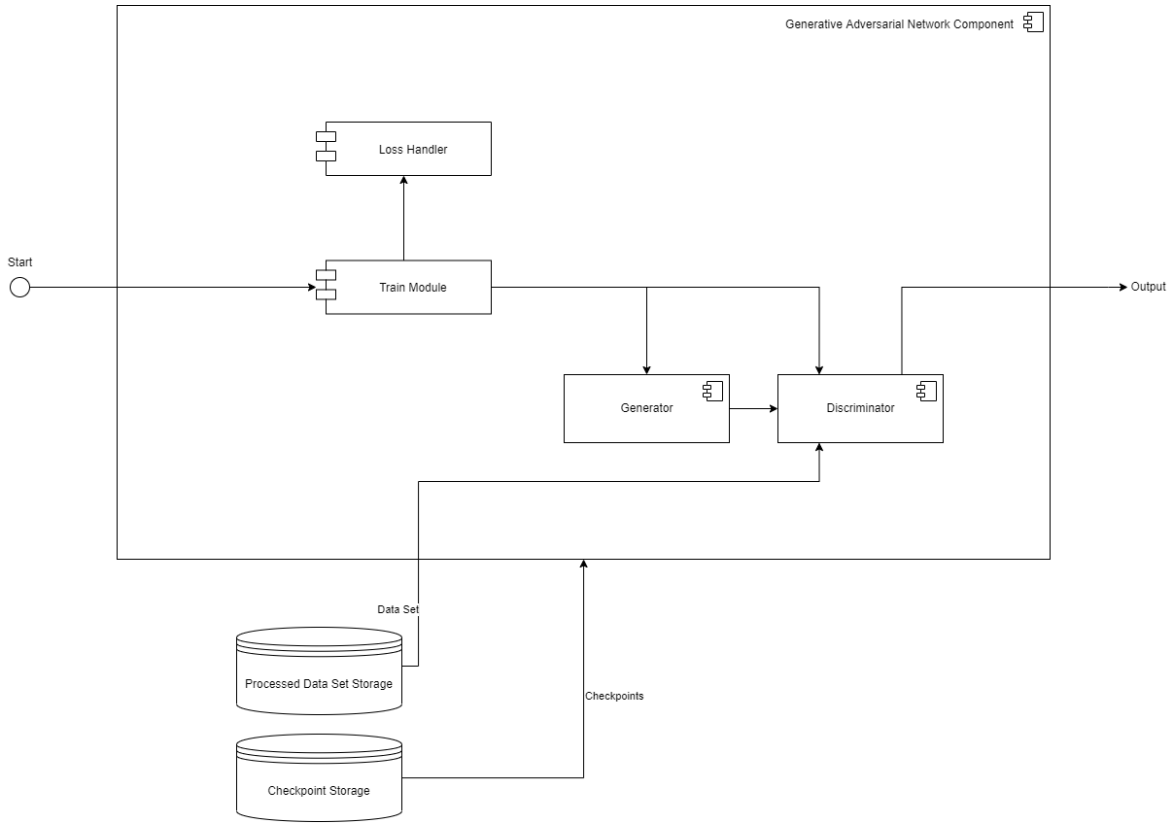

Figure 3.4: GAN model

Our GAN model can be seen simplified in figure 3.4. We save checkpoints to a directory and with a simple change in our code we can select if we want to feed that to our system

so it does not start from epoch zero. We explained our generator and discriminator with models explaining them even more.

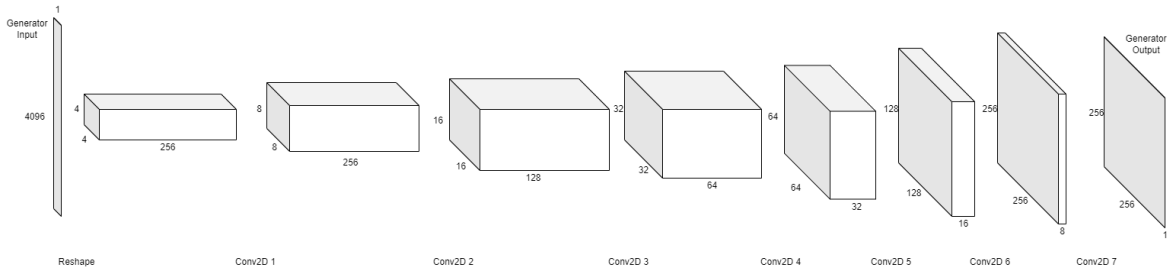

Figure 3.5: GAN's Generator Model

A visualisation of the layers and their dimensions contained in the generator model can be seen in figure 3.5. The operations are performed to allow us to reach our desired 256x256 dimension from an arbitrarily shaped vector of noise, which was 1x1x4096 in our case.

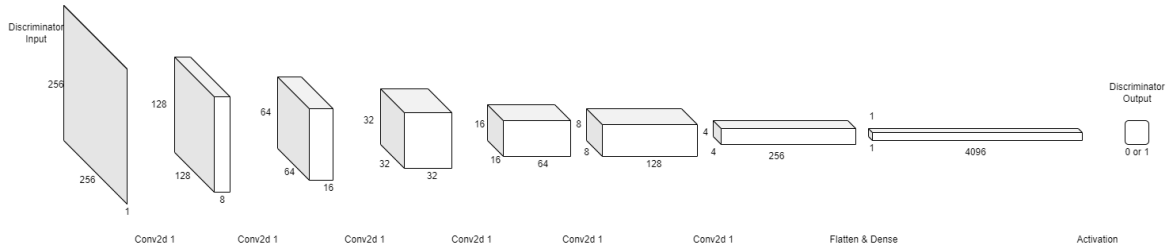

Figure 3.6: GAN's Discriminator Model

Similar to the generator model the layers and convolutions performed in the discriminator model are shown in figure 3.6. Here the resulting output is 1 or 0, which indicates the guess of the discriminator that the image is either real or fake.

The final component in our process from heightmap to terrain is the terrain renderer. We use a Python script to turn our heightmaps into STL files, which can then be rendered in any 3D object viewer of choice, or even 3D printed as a physical model. Example output terrain renders, along with their heightmap outputs from our GAN are displayed below in figures 3.7 through 3.12.

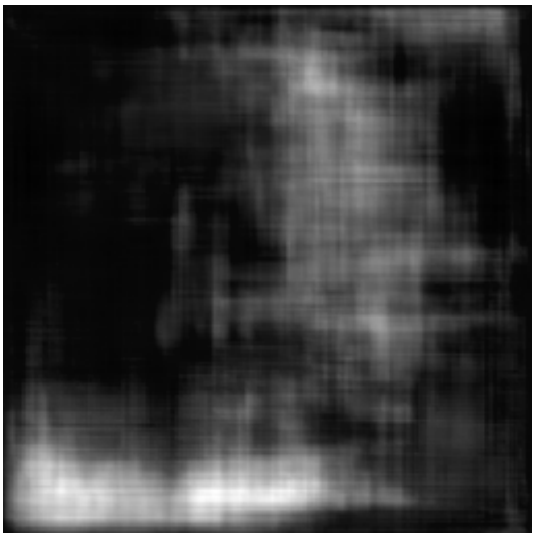

Figure 3.7: Heightmap Example 1

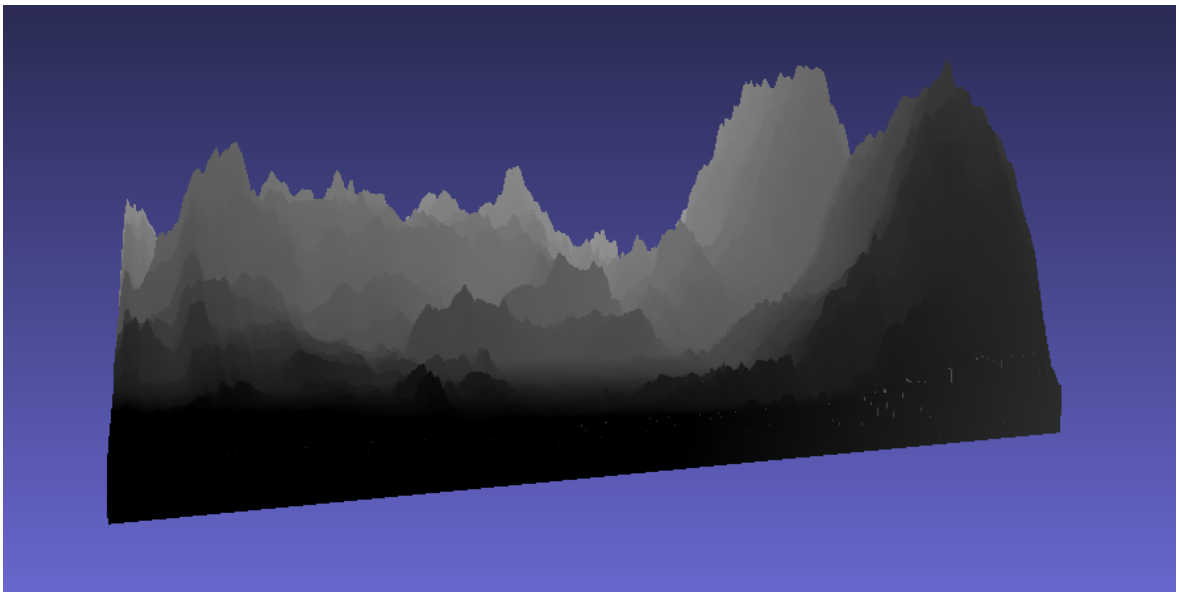

Figure 3.8: 3D Render of Heightmap Example 1

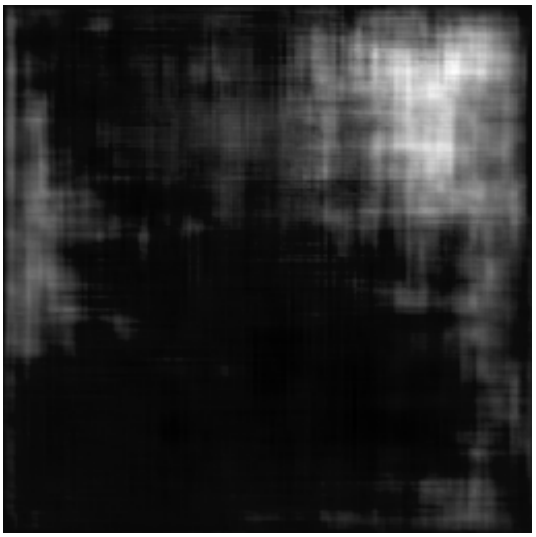

Figure 3.9: Heightmap Example 2

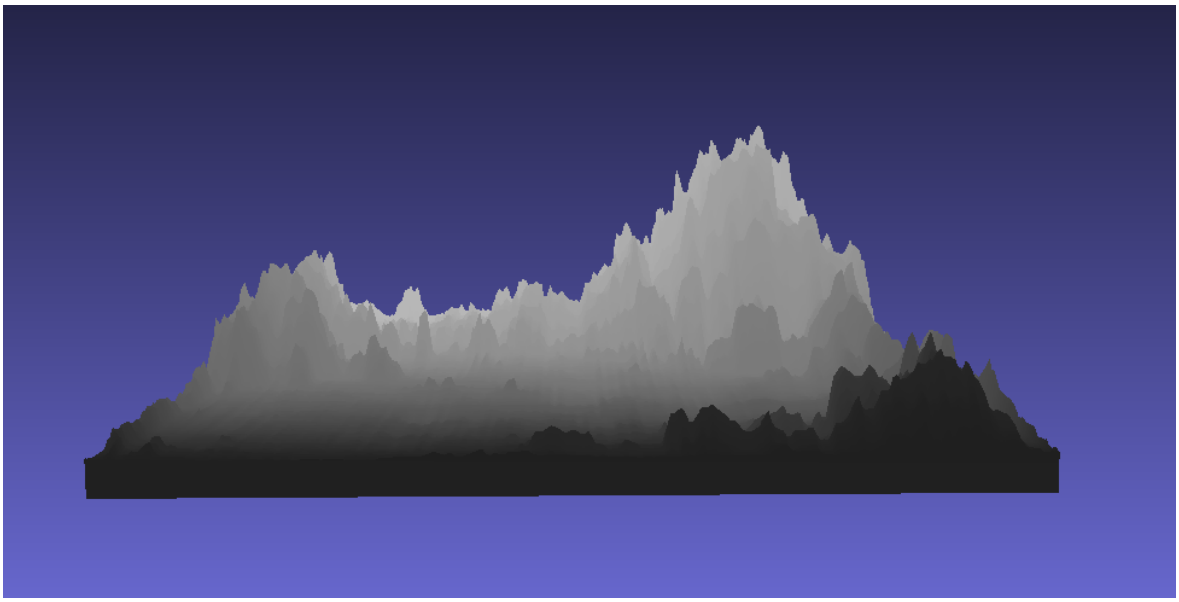

Figure 3.10: 3D Render of Heightmap Example 2

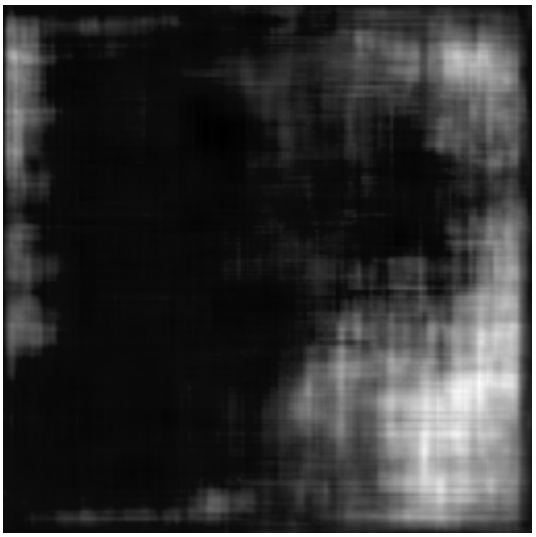

Figure 3.11: Heightmap Example 3

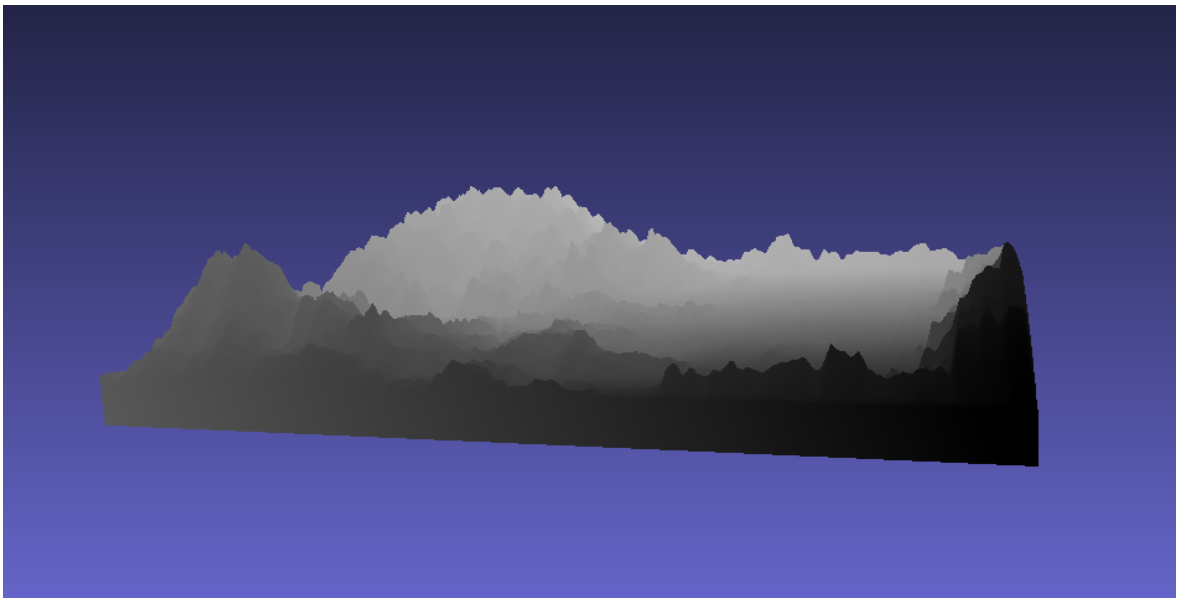

Figure 3.12: 3D Render of Heightmap Example 3

4 Experimentation Environment and Experiment Design

Generative Adversarial Networks as a design easily lend themselves to experimentation due to the high number of variables and configuration options available. While developing our GAN we experimented with a variety of approaches in the coding stage of the project, before landing on our final design which we still believe can be tweaked for further performance improvement.

We first experimented with the use of different libraries to implement our GAN in Python and made the joint decision to use the latest version of Tensorflow v2 and the Keras library. A suitable alternative would be the PyTorch library which has seen successful GAN implementations and use in different areas of machine and deep learning, we found it lacking in resources when compared to Tensorflow. Experimentation in our project was usually conducted in response to various issues we faced, with some relating to hardware.

4.1 Difficulty During Testing

4.1.1 Result Observation

Machine learning and the tasks performed by GANs are very resource intensive, requiring both a lot of processing power as well as storage space for the dataset as well as any output that will need to be saved. Due to the procedural nature of the process of generation with a GAN, it can take a large number of epochs to determine the accuracy of the current model. Depending on the efficiency and complexity of the implementation, the amount of time an epoch can take may be a substantial amount of time, as one of our early models took over 2 minutes per epoch on our personal computers. With models and hardware such as the ones described, this meant that on occasion it took near to 7 hours to observe the results of a model set. The use of resources also rendered our personal computers useless during the process if ran in a local environment.

Running the models without a Graphics Processing Unit (GPU) also had significant impact on the speed, with slightly above average GPU cards being able to run epochs over 10 times faster than the average CPU. Realising that developing and testing in such a manner was unsustainable we moved our development and experimentation environment online. We explored our options for online code execution, such as Amazon provided EC2 instances with access to GPU units for development, or Kaggle environments however developing on these services proved a challenge. We discovered the best balance between ease of development and was offered by the Google Colaboratory service, which allows users to execute code written in the style of Python Notebooks, as well as up to 40 GB of storage on the cloud machine, with easy Google Drive integration.

4.1.2 Availability of Information

Giving to the fact that GANs have only recently been an area of focus in neural network development the relative scarcity of resource compared to other coding subjects was an issue at first. Frameworks that were used in plenty of other prior work in the field are currently inoperable, with newer versions not supporting older implementations. We mainly had issues with loading our dataset and performing convolutions, however were able to solve these issues through prior experience in coding and through theoretical knowledge obtained from out other sources. The main source we used to learn the library we implemented our GAN with was the Tensorflow Keras documentation on the official website [21].

The issue of availability on the topic of heightmaps was also a large obstacle for development, as recently the United States passed a bill making USGS terrain data private. This meant we had to find alternative sources for information, that were of sub optimal quality and harder to obtain. If there was more terrain data and heightmap imaging freely available to the general public a more advanced model of the GAN would be able to be trained.

4.2 Experimentation Environment

The resource requirements for the training process can be prohibitive for work in the field of GANs to be carried out unless a powerful machine is available, or a significant budget for online resources is allocated. The generation process specifically requires a lot of processing power, which is most efficiently supplied by GPUs. For a large part of the project we did not have access to a physical device with a strong GPU, so performed most of the development and testing on the aforementioned cloud development services. For the final stages of development we gained access to an Nvidia RTX 2060M which offered us performance that was over 10 times quicker than that of the online services.

Once our experiment with a certain model had finished running for over 500 epochs we inspected the output of the model manually, and decided through the accuracy of its results if it was an improvement or not. We also used methods like making animated GIFs of each epoch output of a model to visualize the learning progress of any model, as well as checking loss values to determine system failure in unsuccessful trials to cut them early.

4.3 Experiment Design

4.3.1 Generator Experimentation

The main component of the GAN that we experimented with and trialled multiple versions of was the generator architecture. Our initial design for the project used transposed convolution layers, otherwise known as deconvolution layers, through Keras'

Conv2DTranspose function. Early models that used this method were more efficient in terms of performance, with faster epoch times on average, however the results suffered in quality.

Using Conv2DTranspose the main issue we encountered was heavy artifacting and noise on the resulting image outputs. This topic is explained in further detail by Odena et. al in their 2016 study [22]. While explaining that the problem can be near impossible to remedy completely, they offer approaches to mitigate the loss in quality caused by the 'checkerboard' artifacts. Following the discovery of this issue with GANs we aimed at solving this problem by removing Conv2DTranspose from our generator structure. We replaced the usage of this function by performing 2D convolution, batch normalization and finally an upscale function to achieve the desired output dimensions. While this replacement alleviated the artifacts for the most part, the overlapping layers of convolutions still left us with some undesirable vertical artifacts in a majority of our models.

4.3.2 Discriminator Experimentation

Our design process for the discriminator did not include as much experimentation as that of the generator, as the convolution process has less alternatives than deconvolution. We experimented with different activation functions such as sigmoid and softmax however found that we achieved the best results with tanh activation. We also experimented with linearity layers, and found again that Leaky_ReLU yielded the best results when compared to other options. The main experiment we made was to add Gaussian noise to the input of our discriminator following the results of Salimans et. al 2016 study [23] showing that it may improve results, however it had little no effect on our results.

5 Comparative Evaluation and Discussion

5.1 System Design

The design of a GAN in this project was an implementation of a DCGAN through the use of the Python programming language. The choice of library was Tensorflow, specifically version 2.7 with use of the Keras layers library. The basic flow of the code is as follows:

- Take the input images as a dataset
- Define generator and discriminator models
- Set optimizers and other run-time variables
- Train for the desired number of epochs
- Save and display the output

5.1.1 Training Parameters

Before the training process for the network begins the user must first set various parameters which highly affect the results, mainly the loss and optimizer values. In order to help calculate loss we made use of the *BinaryCrossEntropy* function in Keras, which computes the cross-entropy loss between the generated images and real images from the dataset. The loss for the generator and discriminator are calculated similarly as show in the python code following this paragraph, with the correct behaviour for each being compared to an array of 1's for validity, with the incorrect being compared to an array of 0's. This method of dual testing for the discriminator checks both how well it is at discerning real images, with 1's being compared against real images, as well as how good it is at detecting fakes with the 0's being compared against a run of fake images. The lower the loss value for each model, the better the model is functioning.

```
1 def discriminator_loss(real_output, fake_output):
2     # [8][9] Using soft labels
3     real_values = np.random.uniform(0.95, 1.0, size=
    real_output.get_shape())
4     fake_values = np.random.uniform(0.0, 0.05, size=
    fake_output.get_shape())
5
6     real_loss = cross_entropy(real_values, real_output)
7     fake_loss = cross_entropy(fake_values, fake_output)
8
9     total_loss = real_loss + fake_loss
10    return total_loss
11 discriminator_optimizer = tf.keras.optimizers.Adam(1e-5)
12
```

```
13 def generator_loss(fake_output):
14     return cross_entropy(tf.ones_like(fake_output),
     fake_output)
15 generator_optimizer = tf.keras.optimizers.Adam(2e-4)
```

Optimizers are a vital component of any neural network, with the main objective of the optimizer being to adjust the weights of a model to achieve a lower error, or loss rate. The function is used in order to optimize gradient descent of each model, with the optimization function implemented being Adam optimizer. Following the results of the 2018 research carried out by Srivastava et. al [24], we used the Adam optimizer function in this project. This is widely regarded in the deep-learning communities as the most efficient optimizer function for neural network projects.

5.2 Comparative Evaluation

This project is an advancement on the approach of using Perlin noise [8] to generate terrain, as it uses an input dataset to generate more realistic terrain than that what is generated from random noise. Models generated using Perlin noise are not configurable and will be at the mercy of the noise generator, while our implementation of a GAN allows users to train the network in a biased sense, by giving it a dataset of the desired specification.

While all GAN implementations are similar structurally, they differ in the details based on the task expected of the network. This project was built on the basis of a DCGAN that is explained in the Tensorflow Keras documentation [21], with heavy modifications. Due to new versions of Tensorflow being released, the code required updates in the input and generator phases. Once updated, we modified the generator, discriminator and pre-processing functions. Our project is different to many others in this aspect as we use 256x256 pixels as a resolution for our input images, while most others opt to use a resolution of 28x28 pixels, following the most popular implementation GAN that is the MINST numbers GAN.

This project is most comparable to the GAN designed by Beckham and Pal [6], as the final aim is to generate terrain with a GAN in both. The Beckham and Pal GAN differs from ours in terms of input and output, as we focused purely on generating terrain from an input dataset of heightmaps, whereas their GAN included terrain texture details and colour, which was handled through the implementation of the 'pix2pix' [15] GAN design.

6 Conclusion and Future Work

This project focused on the use of GANs in the area of terrain generation. The goal of being able to generate a 3D terrain object that can be textured or explored, using an input dataset and a GAN was achieved. Efficiency goals were also met, with the caveat being it hard to measure quantitive results as the performance of different implementations has varying efficiency depending on the architecture it is run on. In our project we managed to successfully determine the type of GAN we would need to implement for the desired results, and tweaked the configuration until it yielded satisfactory results.

The conclusion we drew from the work performed in this project is that manualy generated terrain will not remain industry standard for long. While it will always be part as results can be altered to the developers desired detail, the overall process of generating terrain is much faster when performed through a GAN. Studios looking to build worlds could pour significantly more resources into a similar project, and with a much larger dataset and a more sophisticated implementation of a GAN, we believe it is possible to generated terrain that would be virtually indistinguishable from that drawn by hand.

6.1 Future Work

The potential for GANs and projects utilising them is vast, and this project is no exception. There are a multitude of developments that we aim to make to this project in the future when the time and budget becomes available to us. Ranging from performance improvements to new features and integrations, this project is easily modifiable to anyone with a basic understanding of GAN architecture.

6.1.1 Result Quality

We aim to improve the results of this project in the future, as we discovered the main issue with our implementation is the lack of a fleshed out dataset. We were able to gather over one thousand high quality height maps, and modified them to result in a dataset of over five thousand images, however this is not enough data for high quality output. In future implementations we would like to run this GAN implementation with datasets preferably of over ten thousand images, that may also be classified depending on the terrain structure that the heightmap corresponds to. Obtaining such a dataset however is a very time and resource intensive process, yet having a GAN be able to generate and classify high quality output for a multitude of terrain formations, such as valleys, plateaus, mountain ranges etc., would be of great use to those looking to build terrain.

6.1.2 Texturing

Inspired by the impressive results of the study and work performed by Beckham and Pal in 2017 [6], we intend to enhance our project by adding a second GAN that is able to generate a texture map for the resulting terrain. This GAN could be able to classify terrain based on climate, or latitude and generate the according texture. This classification system would be an important part of this proposed enhancement as geological formations that are unalike in texture, such as deserts and tundra, may have near identical heightmaps.

6.1.3 GAN Performance

Considering that GANs are a relatively young neural network, with more research being done in the field in the coming years we believe new advances will be made toward making GANs more efficient. We aim to follow these advances, such as the updates to Python neural network libraries and follow research papers on the topic in order to keep up to date with the latest information. When new methods or functions easing the generation or training process are released, we plan to implement them and update our GAN model for better results.

6.1.4 Software Package

All of the tools that we developed in order to complete this project can theoretically be bundled into a single software package. This project could serve as a standalone application for the personal computer, where a user may be able to pick a location in the world, or a terrain formation that they would like to generate similar versions of to their desired specification. A frontend could be designed that will allow end users to manipulate parameters used in the generation process, such as the output file dimensions, texturing the resulting terrain or optimizers to allow for greater detail, without requiring the knowledge of the software architecture, and allow them to visualize results. This would yield a real world use for our project in that anyone can obtain generated world files of their choice.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[2] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[3] H. R. Team. (2021) The top 50 best-selling video games of all time. [Online]. Available: https://www.hp.com/us-en/shop/tech-takes/top-50-best-selling-video-games-all-time

[4] H. Games. (2020) Game screenshot. [Online]. Available: https://nmswp.azureedge.net/app/uploads/2020/10/No-Mans-Sky-Next-Gen-Bug.png

[5] A. Bielik, "Adjusting to job demands: The role of work self-determination and job control in predicting burnout," *Animation World Network*, 2007. [Online]. Available: https://www.awn.com/vfxworld/navigating-golden-compass-part-2

[6] C. Beckham and C. J. Pal, "A step towards procedural terrain generation with gans," *ArXiv*, vol. abs/1707.03383, 2018.

[7] S. Mo, M. Cho, and J. Shin, "Freeze the discriminator: a simple baseline for fine-tuning gans," 2020.

[8] S. Turner. (2019) Perlin noise, procedural content generation, and interesting space. [Online]. Available: https://heredragonsabound.blogspot.com/2019/02/perlin-noise-procedural-content.html

[9] W. Eck and M. Lamers, "Biological content generation: Evolving game terrains through living organisms," vol. 9027, 04 2015.

[10] A. Odena, "Semi-supervised learning with generative adversarial networks," 2016.

[11] Y. Clarke, "Deep fakes accountability act," *116th US Congress, House - Judiciary; Energy and Commerce; Homeland Security*, 2019.

[12] K. Perlin, "An image synthesizer," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 287–296, jul 1985. [Online]. Available: https://doi.org/10.1145/325165.325247

[13] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[14] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. Metaxas, "Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks," 2017.

[15] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," 2018.

[16] R. J. Spick and J. A. Walker, "Realistic and textured terrain generation using gans," October 2019. [Online]. Available: https://eprints.whiterose.ac.uk/153088/

[17] J. Patryk. (2021) Heightmap generator git. [Online]. Available: https://github.com/sysoppl/Cities-Skylines-heightmap-generator

[18] ——. (2021) Heightmap generator website. [Online]. Available: https://heightmap.skydark.pl

[19] J. W. . M. development team. (2016) Basemap documentation. [Online]. Available: https://matplotlib.org/basemap/

[20] J. Dietrich. (2013) Heightmap2stl tool. [Online]. Available: http://adv-geo-research.blogspot.com/2013/10/converting-dems-to-stl-files-for-3d.html

[21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[22] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, 2016. [Online]. Available: http://distill.pub/2016/deconv-checkerboard

[23] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," 2016.

[24] M. Srivastava, S. Pallavi, S. Chandra, and G. Geetha, "Comparison of optimizers implemented in generative adversarial network (gan)," *International Journal of Pure and Applied Mathematic*, vol. 119, no. 12, 2017.