From Sketch to Scene: Using GANs for Realistic Landscape Generation and 3D Reconstruction

Barnabas Juhasz (2043652), Charles Heron (2044203),
Diego Velotto (2071094) and Nadeem Kirolos (2053211)

Introduction

Our main goal was, given the limitations of time, data, and hardware to create a model capable of producing 3D mountain landscapes from a Sketch. To achieve this we first needed a dataset composed of mountain pictures, which we sourced from the Flickr website, using their free API. Next we processed and normalized the data.This included steps such as resizing, quantizing the colors, and extracting the main contours. Once the dataset was preprocessed, it was then fed into a Generative Adversarial Network (GAN). A GAN is an unsupervised learning algorithm that works by leveraging two separate deep learning models: a generator and a discriminator. The generator creates fake samples which are then passed to the discriminator. The discriminator is also given real samples from the dataset and determines whether each sample is real or fake.This process calculates a loss function, which is used to update the generator. This cycle continues until the generator is able to consistently fool the discriminator. Once the GAN generator was trained, we could move on to 3D rendering. Reconstructing a 3D scene from a single image is a challenging task. However, it was a challenge we wanted to tackle, as it has not been explored to the extent we desired. After extensive research, we developed methods for 3 different pipelines that suited the potential needs of our application. Ultimately, we were satisfied with the results.

---

1. Data

To train and build the model we required images of mountainous landscapes. Initially, the images had to fulfill the following requirements:

- Mountain scenes largely free from man made structures, people and animals
- Contains both mountain and sky (ie not a picture of only sky or ground)
- Free from borders, watermarks, or any other obvious digital alteration
- At least 256 pixels on the shortest edge

The original input data for the mountain image model was drawn from two sources. A dataset pulled from images.cv[**7**], and a dataset of images pulled from the flickr api using tags such as “mountain”, “alps” and “landscape”.

Both datasets were initially of low quality, with numerous images that bore little to no relevance to the requirements. As such, some time was spent manually screening them, keeping only the suitable images. Initially the dataset consisted of 612 images. This resulted in good results from the test set, however validation results were average at best. On the suspicion that we did not have enough data, we screened and added more, leaving a dataset of 1654 images.

Multiple initial runs were tried with different settings, however, even after seeing some marginal improvement with fine tuning, the generator model continued to produce mediocre results.

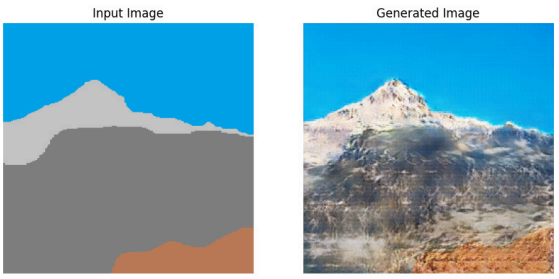

Figure 1: *Early input image and generated output*

As such, we decided to revisit the dataset to try to optimize it with the benefit of a clearer idea of what the model is likely to be able to achieve. We removed many images which, while mountainous, were not really within the scope of our goal. In the final pass we also removed:

- Desert scenes
- Oceanic island-like mountains
- Arty shots which had unusual perspectives, or used natural framing, such as trees or caves
- Night shots
- Gloomy mountainous shots
- Mirror lakes (where the scene is perfectly reflected)

This final pass resulted in 1397 images, and saw immediate improvements in generator model output on validation images.

2. Preprocessing

The image processing pipeline begins by identifying the pixel coordinates of four distinct colors within the input image, ensuring the image contains exactly four unique colors. Next, it calculates the average color of the specified pixel coordinates, defaulting to black if none are provided. To simplify the image's color palette, the colors are quantized into four distinct colors using techniques such as contrast adjustment, bilateral filtering, and k-means clustering. These colors, alongside the reduced depth image from the original image, are then used to produce a traced image.

To train the model for generating realistic mountain drawings, the capabilities of the AutoModelForDepthEstimation were leveraged to understand and interpret the depth and spatial relationships within an image, mimicking how artists perceive and render three-dimensional landscapes. The color quantization provided the model with a basis for differentiating key elements in the landscape, such as varying shades and textures that constitute realistic mountain scenery. Furthermore, this process helped the model retain natural color variations while translating them into a simplified, drawing-like format. This method ensures that the model captures both the structural and aesthetic nuances of mountain images, thereby enhancing its ability to generate drawings that closely resemble authentic mountain scenes.

3. Training the model

The Pix2Pix[1] GAN model we used was based on the implementation provided by Aladdin Persson which uses the Conditional GAN architecture[9]. There are two differences when compared to the more traditional model, namely, the introduction of a loss function that does not have to be computed manually, this inturn means that the Generator's goal is to minimize its own loss, which can be achieved by maximizing the Discriminator's loss. The Generator is an encoder-decoder based on the "U-net" architecture, its key features being the concatenations, or "skips", between each downscaling step and its corresponding upscaling step. The input is downscaled until the bottleneck layer, which lives in a 1x1 dimensional space, then it is upscaled and outputted. The Discriminator uses a standard implementation, with the only difference being that it works on patches of the image, individually deciding if that patch is real or fake. Not only does this make it more efficient but it also allows the discriminator to work on arbitrarily sized images.

One of the important aspects of training a GAN is the fine-tuning of its hyperparameters, namely the Generator and Discriminator learning rates, as well as the batch size, and number of epochs the model will train for. In our first few trials we trained the model at a batch size of 8 and kept the Generator and Discriminator learning rates equal, but we

noticed that we were achieving convergence too quickly and that left us with low quality results filled with noticeable artifacting. Turning the learning rates we found that it was most optimal to use a value slightly higher for the discriminator of 0.00008 and 0.00007 for the generator. This fixed the premature convergence issue. However the generator was generalizing suboptimally and consequently inputs from the validation set would return incoherent results, meanwhile inputting an image from the training set would result in a near perfect copy. This was most likely alluding to possible model overfitting. To combat this we used a batch size of 1, which led to better generalization but was less efficient during the training needing more time per epoch.

4. 3D Scene generation

After the training of the model, the next stage of the project involved expanding into the third dimension, meaning to create either a point cloud or a mesh based on the output of our model. Reconstructing 3D scenes from images has been explored quite well already; one of the most recent findings uses Mip-NeRFs [2] to achieve amazing results, but the authors use multiple pictures of the same object from different angles. The main difficulty in our case comes from the fact that we only have a single image to work with [and this is not something we can improve on]. We had a couple of ideas on how to approach this problem, and we came up with 3 pipelines.

The pipelines all share a key idea, which is depth estimation. For this we use a pre-trained model: Depth Anything [3], a highly practical solution for robust monocular depth estimation. With that being said, the model was trained on real-life photos, therefore if our model generates some artifacts, these could propagate, but the occurrence of this is not common. See a visual example of the Depth Estimation below (figure 2):

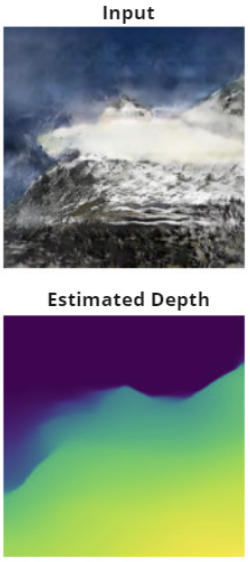

Figure 2: *from top to bottom: the output of our generator model, and the result of the depth estimation*

4.1 The basic pipeline

We inference the pre-trained model to acquire the depth image, using the open3d python library, we project the points of our image into 3D based on their depth, thus we acquire a point cloud. Optionally, we can perform normal estimation and generate a mesh using poisson surface reconstruction [4]. An important observation is that with the basic pipeline the sky of the image does not appear in the resulting point cloud/mesh, as it is considered to be infinitely far away by the depth estimator. This could of course be fixed, but it is not something we covered in the project. For the visualization of the basic pipeline, see figure 3.

The basic pipeline has some advantages compared to the others, namely: it is considerably faster, as it is the simplest and least intensive computationally.

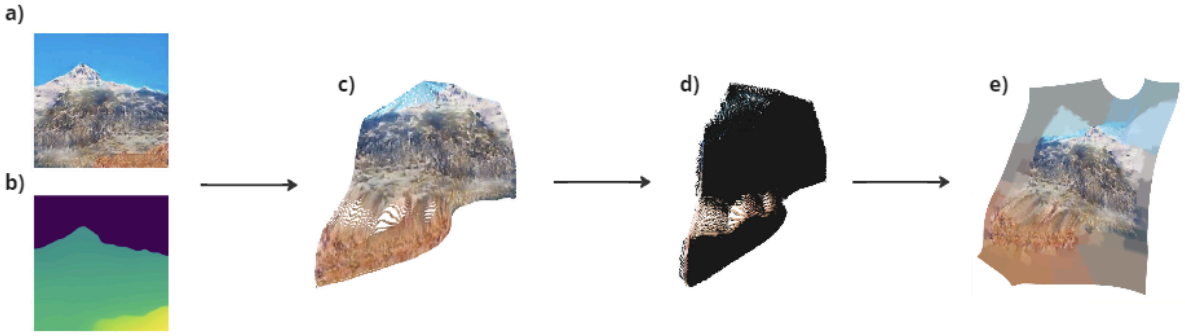

***Figure 3**: The basic pipeline. **a)** The output of our model. **b)** The generated depth. **c)** Estimated point cloud. **d)** Estimated normals. **e)** Final mesh*

4.2 The extended pipeline

The extended pipeline is very similar to the basic pipeline. In order to have a larger image to work with [and therefore larger point cloud as a result], before the depth estimation, we first extend the image horizontally to be twice the width of the original. This is done by the use of the pre-trained model: Boundless [5].

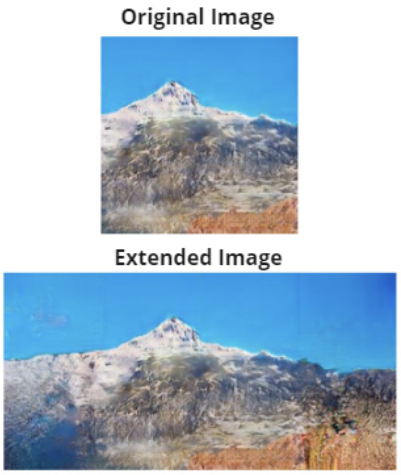

***Figure 4**: from top to bottom: the output of our generator model, and the output of the boundless model*

The results of this pipeline are of course similar to the results of the basic pipeline, except we have a larger surface as a result. Since some of the artifacts potentially propagate to depth estimation, the sky also might be part of the resulting mesh. Figure 4 depicts the visualization of the extended pipeline.

4.3 The noise-based pipeline

The noise-based pipeline is essentially an extension of the basic pipeline. The main idea is to create a larger point cloud or mesh, with the use of the one coming from the basic pipeline. To do so, we use a complex perlin noise obtained by stacking multiple noise maps on top of eachother, where each noisemap has different number of octaves and frequency. We can also scale the resulting noisemap in order to resemble the terrain shape of the point cloud obtained by the basic pipeline. The noise-based pipeline starts with depth estimation and projecting into 3D. We proceed with downsampling our estimated point cloud. For this, we use a factor of 0.0025: from around 150,000-200,000 points we keep around 375-500, while keeping the shape of the mountain intact. The results of this procedure can be seen in figure 4 part d. The point cloud is now prepared to be projected onto 2D, which creates a heightmap in the shape of the mountain. This heightmap is then mixed with perlin noise, with the application of biharmonic smoothing and blur. Finally, we can project the new heightmap back into 3D, and we obtain a terrain which fits our mountain. For the visual noise-based pipeline, see figure 5.

4.4 Conclusion on pipelines

All 3 of the pipelines have something unique, which amounts to a drastic change in the resulting point cloud / mesh. They also have different advantages and disadvantages. The basic pipeline, being the most simple and straightforward, is the fastest of the proposed pipelines. In comparison, the extended pipeline is prone to generate artifacts but produces a

larger surface. With that being said, our favorite remains to be the noise-based pipeline. There is definitely space for improvement, especially where the noise is getting mixed with the heightmap. For this part, there was also an idea of using a modified Wave Function Collapse algorithm, with custom feature extraction in order to fit our special needs. After the features were extracted, we would have generated a larger height map based on our original projection. This larger height map would have had the same features, resulting in shapes similar to our original mountain. Unfortunately, there was not enough time for us to explore this version of the pipeline.

5. Paint tool

A simple tool, originally created by Github user mdoege[8], was modified to contain the exact featureset required for the mountain image model. This allows users to generate images in the style expected of the model, with a suitable color palette that captures all of the scenes we expect to find in the data set.

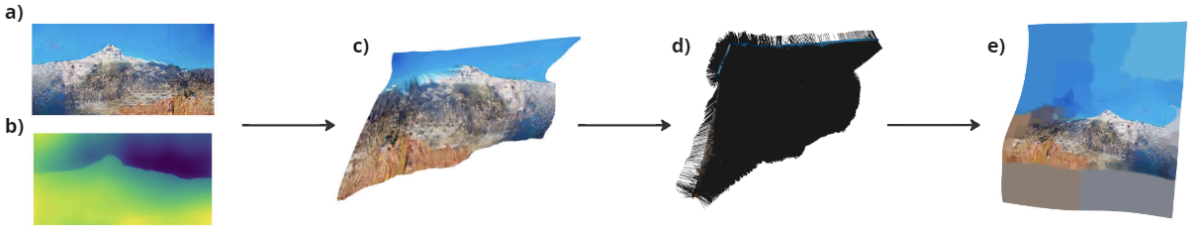

***Figure 5**: The extended pipeline. **a)** The extended version of the output of our model. **b)** The generated depth. **c)** Estimated point cloud. **d)** Estimated normals. **e)** Final mesh.*

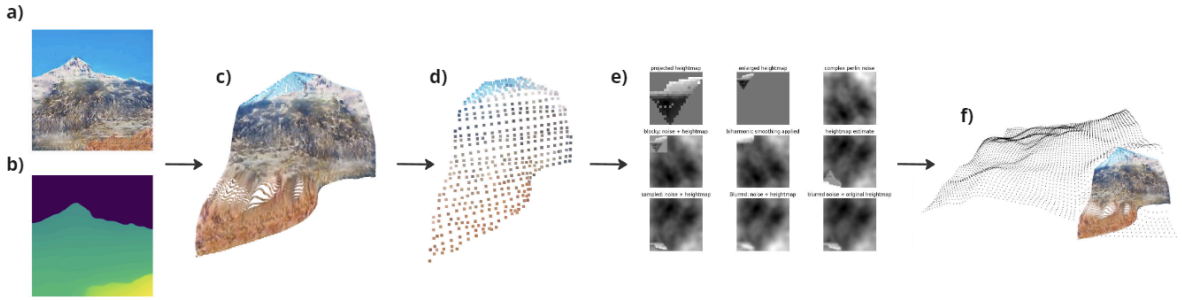

***Figure 6**: The noise-based pipeline. **a)** The output of our model. **b)** The generated depth. **c)** Estimated point cloud. **d)** Downsampled point cloud. **e)** Projecting onto 2D and mixing the obtained heightmap with complex perlin noise. **f)** Projecting back into 3D and visualizing the terrain together with the estimated point cloud.*

6. Results

After optimizing the model, dataset, pre and post-processing stages the model performed well, consistently delivering good results, even when the input was not in the style of the original pre-processed images, as shown in Fig 7. When the input style accurately matched the original preprocessing the models were able to output some extremely good results, see Fig 8.

Finally, to tie the multiple aspects of this project together, a simple application was created allowing the user to draw and immediately generate a mountain scene.

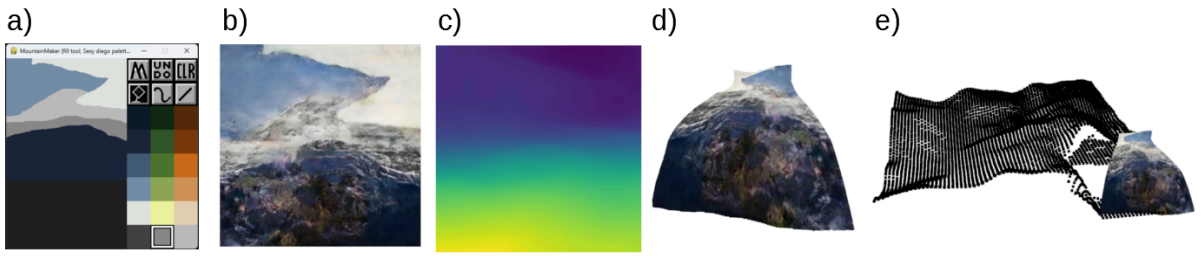

***Figure 7**: Performance with input differing from original dataset. a) Paint tool b) Generated image c) Generated depth map of generated image d) Estimated point cloud e) Estimated point cloud with generated terrain*

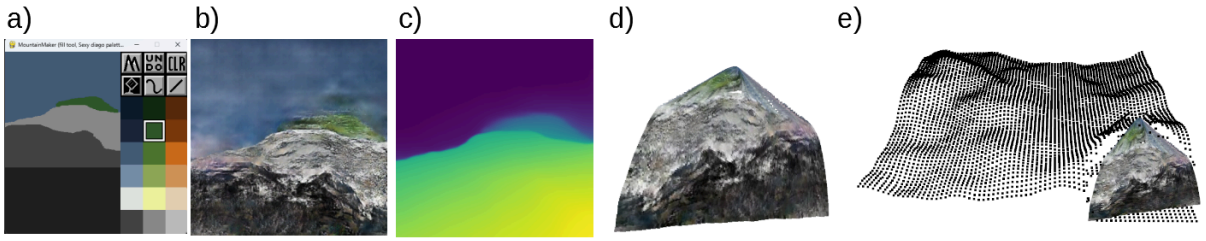

***Figure 8**: Performance with input in similar style to original dataset. a) Paint tool b) Generated image c) Generated depth map of generated image d) Estimated point cloud e) Estimated point cloud with generated terrain*

References

[1] Isola, Phillip & Zhu, Jun-Yan & Zhou, Tinghui & Efros, Alexei. (2017). Image-to-Image Translation with Conditional Adversarial Networks.

[2] Jonathan T., Ben M., Matthew T., Peter H., Ricardo M., Pratul P.: Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. ICCV (2021)

[3] Lihe Y., Bingyi K., Zilong H., Xiaogang X., Jiashi F., Hengshuang Z.: Depth Anything: Unleashing the Power of Large-Scale Unlabeled Data (2024)

[4] Michael K., Matthew B., Hugues H.: Poisson Surface Reconstruction (2006)

[5] Piotr T., Aaron S., Dilip K., Aaron M., David B., Ce L., William T.: Boundless: Generative Adversarial Networks for Image Extension ()

[7]https://images.cv/dataset/mountain-image-classification-dataset

[8] https://github.com/mdoege/PyPaint/tree/master

[9]https://github.com/aladdinpersson/Machine-Learning-Collection