BABEŞ BOLYAI UNIVERSITY, CLUJ NAPOCA, ROMÂNIA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

From Pixel Art to Satellite Images: Enhancing Generation of Terrain using Generative Adversarial Nets

– Diploma thesis –

Author
Ovidiu-Calin Iosif, Computer Science, 934

2023

UNIVERSITATEA BABEŞ BOLYAI, CLUJ NAPOCA, ROMÂNIA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

De la Pixel Art la Imagini din Satelit: Îmbunătăţirea Generării de Relief folosind Reţele Generative Adversative

– Lucrare de licenţă –

Autor
Ovidiu-Calin Iosif, Informatică în Limba Engleză, 934

2023

Abstract

A novel approach for generating realistic terrain images from pixel art called pix2sat is introduced. The study explores the use of Generative Adversarial Networks (GANs), specifically the pix2pix model, and introduces modifications to further improve the quality of the generated images. The methodology involves using procedural noise functions to generate pixel art, which is then processed by the generator model to produce satellite-quality images. The modifications are evaluated using metrics such as FID, LPIPS, and SSIM. The pix2sat approach shows promising results in generating images perceptually close to real-life samples, with potential for further improvements and applications in various fields.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ever since computers became powerful enough to carry out varied and complex operations, the question of their ability to simulate the real world was raised. Could a human-built computation machine represent in one way or another the intricacies of mother nature? The first step in answering this question was made with the advent of procedural noise functions. Although not their original purpose, these functions were able to generate infinitely expanding matrices of numeric values, which could be interpreted as elevation points in a three-dimensional environment. Ever since they were made popular in the 80s, the problem of capturing the essence of natural landscapes appeared in numerous instances, from military applications to video games. With the continued rise of performance in Artificial Intelligence, along with Terrain Generation, as often happens in Computer Science, have been merged in order to create even more elaborate representations of the world around us. The vast amount of work completed with the shared purpose of creating simulated landscapes has led to solutions that cater to diverse use cases. Thus, the field of Terrain Generation has been growing in various dimensions since its genesis.

Regarding the position of this thesis in the field of Terrain Generation, it aims to present a new method of controlled realistic landscape generation, and as of my knowledge, no other similar method exists. While there is a multitude of methods for generating terrain structures with diverse sets of input mechanisms, the technique to be defined and expanded upon in this work was developed in order to fill a gap that was present in the space of related state-of-the-art methods. The idea started by analyzing the different ways control was introduced by related literature. While effective, all of them seemed to fail when it came to simplicity. Having this in mind, the proposal of using pixel art as a source of control was born. Pixel art is often considered an enjoyable form of art due to its simplicity. The grid-based structure provides a less intimidating drawing surface, where tangible results can be obtained relatively quickly. As far as user experience goes, in my opinion, this method allows the

highest level of accessibility.

Having pixel art as input, the technique essentially allows the transformation of these rudimentary overhead representations of terrain landscapes into hyper-realistic images that respect the semantic structure of the input. The elegance of this technique lies in its simplicity. The use cases for it are seemingly endless. One of the possible use cases that came into mind when first coming up with it was the case of fiction literature authors, more specifically J.R.R. Tolkien. I imagined that creating the geography of Middle-Earth, the world most of his novels are set in, would have been a more enjoyable process if Tolkien had access to a tool powered by this method.

This thesis comprises several chapters that collectively contribute to a comprehensive understanding of the landscape generation technique. The first chapter, **Introduction**, gives an introductory overview of where the technique lies in the space of other related literature and how the idea of the technique came about. The **Scientific Problem** chapter describes the method from a scientific point of view, revealing more details and expanding on the possible use cases. In the **Related Work** chapter, multiple diverse approaches to terrain generation are presented from simple rule-based methods to high-end AI-based methods. The **Proposed approach** chapter details the specific processes and mechanisms that collectively define the method. Moreover, the validation of the method and possible improvements can be found in the **Experiments** chapter. The method is incorporated into a software application, and the entirety of its development process, from draft to final product can be found in the **Software Application** chapter. Finally, the **Conclusions and future work** chapter provides an overview of the work presented in this thesis, while also setting up possible improvements that can be achieved in future publications.

The main contributions of this thesis in the field of Terrain Generation are:

- **Developed a novel method for transforming pixel art of landscapes into realistic satellite-quality images that makes use of the pix2pix GAN architecture, powered by a proprietary method for generating the required dataset**
- **Modified the pix2pix architecture by introducing a 34x34 PatchGAN Discriminator, further enhancing the quality of the resulting images.**
- **Addressed the issue of land sliver artifacts in the pix2pix model by introducing a Total Variation Loss, significantly reducing the presence of these artifacts and resulting in more visually coherent and realistic satellite-quality terrain images.**
- **Incorporated the method into an interactive, user-friendly software application tool**

Chapter 2

Scientific Problem

When put into scientific terms, the problem can be described as translating the images from the pixelated landscape image domain into the codomain of satellite-quality images. Since there is a considerably large gap between these domains, classical, non-AI methods would most likely fail to successfully approximate a function that maps the input domain to the output. I suppose that there is a probability that these kinds of methods could reach the same level of performance as AI methods, but they would need to be over-engineered and complex, and this could add a significant amount of overhead, which generally is not present when using AI models. It is true that the training process could be complicated, but at inference time, the models would, in my opinion, outperform the aforementioned classical methods, assuming that their level of quality is relatively similar.

The advantage of using AI methods, more specifically employing Generative Adversarial Nets, is that satellite images are made widely available thanks to the open-source nature of the scientific community. On the other hand, the diversity in the available satellite images could be considered a problem. The number of diverse features could be too much for the model to learn. When constructing the dataset, the balance between quality and quantity should be kept in mind. Another caveat when creating the dataset is the presence of unwanted structures, such as clouds, or urban components that could influence the way the model will eventually learn the terrain formation textures.

When considering the family of AI models that could be able to perform the translation task, the GAN architecture seems to be the most fit. Besides the fact that it is mathematically proven to work, in theory, nonetheless, it is also proven to be the method with the most control over the output due to its use of the latent space to represent the space of all possible representations.

Another important decision would be choosing the actual GAN architecture. Since it's more of a strategy rather than a concrete method, there are a lot of GANs that are built for specific tasks, and they might perform completely differently at the task at hand. Taking this decision should account

for a number of factors, ranging from the dataset all the way to available hardware resources. For the method to be considered useful, the model should be able to generalize, meaning that it should not overfit the training data, while it should also be able to provide a wide range of terrain patterns for specific textures.

Expanding on the subject of the use cases the method could have, the model could be used in transfer learning. More specifically, when faced with a task related to satellite imaging, the layers of the model could hold abstract feature information about specific textures, therefore it can be adapted to a specific task involving these types of images. Moreover, not necessarily the trained model, but the methodology could be used to learn other types of texture-based translations. For example, the model could be trained on terrain images from other planets such as Mars, and from the generated images, simulations could be built to calibrate and test exploration rovers.

As stated before, the method has the potential to be used in military contexts. Strategic planning plays a crucial role in military training, therefore having the ability to generate diverse terrain configurations would provide for a more immersive and extensive experience for military personnel. For instance, if there's a need for simulations of a specific region with a defined set of traits, the model could be trained on a dataset of images from that region, allowing for diverse yer region-specific images to be generated.

The entertainment industry could also benefit from the use of this model, more specifically the video game industry. In the context of strategy games, the world maps are usually hand-crafted, thus unchanging. This could pose a problem since players could, in fact, learn the specifics of each map, and rely on their knowledge of the landscape, instead of their strategic skill. Introducing a different map, generated using the translation model, could add not only realism but also variety to the maps, thus forcing players to rely on their strategic thinking alone.

This chapter explored the problem of translating pixel art images into satellite-quality images using Generative Adversarial Nets. The limitations of non-AI-based methods were pointed out, as well as the potential advantage of using AIs, more specifically GANs, for the defined task. The importance of careful dataset construction was raised, along with the need for the model to be able to generalize. The potential applications of the method, from transfer learning to military training highlight the potential impact of this work in diverse fields.

Chapter 3

Related work

Video games and entertainment are the main reasons for the need to generate vast and complex worlds, with smooth landscapes that also have a random feel to them, but at the same time realistic-looking. Among other uses are simulations and scientific research, therefore this problem has been tackled in many instances, with considerable diversity regarding the approach taken. Usually, terrain generation tasks begin with the generation of a height map, with values between either -1 and 1 or 0 and 1. The trivial method of obtaining such a map would be to use a random number generator, but in the end, the desired landscape should be smooth and continuous, and using random numbers generally doesn't address this. Furthermore, these procedural methods have been extended or even replaced by other approaches that use concepts from different fields of Computer Science.

3.1 Procedural Terrain Generation

Early publications date back to the late 20^{th} century, with the first notable contribution being the recursive approach based on fractals proposed by Fournier, Fussell, and Carpenter in 1982 [8], also more popularly known as the Diamond-Square Algorithm. Given the simplicity of the algorithm, which consists of recursively averaging values that define the edges of either square or diamond-shaped regions in a 2D grid, to the middle point of the region, and adding a random seed to it (the shape is taken in an alternating manner), the results are quite impressive for that time. Although realistic, the generated images present noticeable grid-like artifacts, thus the algorithm is considered flawed, as per Miller's analysis [20] of the algorithm. Further work was done in order to improve on the idea from [8] and to resolve the problem brought forth by [20], with the most notable contributions being presented and reviewed in [4].

In the following year, 1983, Ken Perlin informally introduced the concept of Perlin Noise [25], a

procedural noise function. The algorithm is based on Value Noise, which uses a hypercube-shaped grid to interpolate all the pixels contained in the grid based on the values of the grid points (given by a pseudo-random number generator) and the distances between each grid point and the pixels [2]. Perlin builds on this idea by using gradient vectors instead of height values from grid points and utilizing a permutation table instead of a random number function. Unlike their predecessors, Perlin Noise and its derivations are not fractal in nature. In order to compensate, they are used in combination with Fractional Brownian Motion (fBm), referring to the idea of summing up octaves of noise each with increased frequency and decreased amplitude, which can be interpreted as layers of noise that produce organic, mountain-like formations. Since the introduction of this type of noise, numerous variations and additions have been made to Perlins original implementation, and many different approaches tried to achieve better and more complex results, with the most prominent being Simplex [26], also authored by Perlin, that replaces the hypercube shape with simplex, the shape with the least corners in a given dimension (the triangle is a simplex in 2D, the pyramid is a simplex in 3D). Recently, more work has been put into filtering the generated height maps in order to achieve features such as caves, erosion, and vegetation, than in coming up with new noise functions, thus the state of the art of noise methods hasn't changed in a significant way and is accurately represented in [18].

In the original paper, Perlin proposes the concept of Pixel Stream Editor, a framework that helps designers synthesize textures when working with computer-generated 3D models. In order to give visual complexity to the output, the author introduced in the framework a function simply called Noise(), and in the paper, he presents various ways of using the function to give a torus-shaped object different textures. In one of the examples, the author points out the possibility of mapping different ranges of heights to colors jokingly calling it Bozo's Donut (see Figure 3.1). This method later coined as Simple Color Mapping, is used for Vegetation Modeling in Terrain Generation, one of the most basic ways of transforming a height map into something that resembles a landscape.

Following the introduction of Perlin Noise, given its ability to generate qualitative results fast using constant memory for any input size, it became the standard in industry and research. One of the best-selling games of all time, Minecraft [37], developed by Mojang Studios, uses a form of noise similar to Perlin to generate the terrain, as well as biomes, caves, and structure, each process using the noise in different ways. A notable mention here would also be Akalabeth: Wrath of Doom [32], which is considered to be one of the first-ever commercial video games to use procedural content generation. The game would ask the user for a secret phrase or number, and using that information as a seed it generated the dungeons along with all the enemies and items (see Figure 3.2).

As previously mentioned, the need to build worlds with complex and realistic features has never been

Figure 3.1: Bozo's Donut example from Ken Perlin's original paper [25]

higher, with developments in Virtual Reality and the so-called Metaverse. So naturally, a considerable amount of tools were developed in order to aid in crossing the informational barrier imposed on users by the mathematical knowledge needed for PCG. Some of these tools like Unity Terrain Tools [41] and Gaia [27] offer intuitive user interfaces and are considered accessible to novices. On the other hand, we have World Creator [38] and Gaea [28] are aimed at professionals and require some technical knowledge in order to use them proficiently.

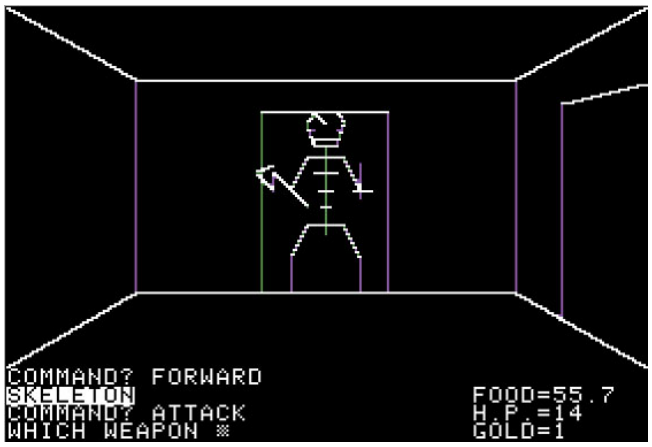


Figure 3.2: Snippet of gameplay from Akalabeth: Wrath of Doom [32]

3.2 Extended Procedural Methods

Starting with developments in terrain generation methods such as procedural noise functions, erosion algorithms, etc., intricate methods based on these fundamental concepts have been proposed, each one expanding this field of study into interesting directions. One of the major downsides with the fundamental methods is the fact that some mathematical knowledge is needed in order to use them

efficiently and as intended. Most algorithms depend on tweaking parameters and seeing what works, and more often than not these parameters don't have an intuitive meaning, therefore these methods, while showing impressive results, are not ideal. One of the most notable contributions to removing this barrier in the design process employs the use of a sketch map and a simplified set of parameters [45] that give meaningful control to the user. The proposed process consists of three steps, Initialization, Terrain Skeleton Generation, and Extension. In the initialization step the user-generated sketch map S is processed. The map S is an image with lines and polygons painted with different colors in order to represent the different types of terrain textures. Iterating through the pixels of S, each RGB value is checked against a set of predefined color values. When a pixel's RGB value is found in the set, its associated terrain type is mapped in a matrix M (having the same dimensions as S). A height map G (with the same size as S and M) is generated having all values set to unknown, the next step consisting of populating it. Iterating through G, each value is set by taking into account the corresponding feature type from M and the control parameters. Predefined terrain types such as $RIVER$, $LAKE$, and $PLAIN$ only have constant height values associated with them (values defined in the set of control parameters), while $MOUNTAIN$ and $CANYON$ types also have variation parameters, which are used to define an interval from which random values are generated and added to their respective constant values to obtain the final height. In the final step, Extension, ridges, and canyons generated in the previous step are extended to full mountains and canyons. Again iterating through G, for each point p, height values are generated for a set of neighboring points that lie on a line segment with the same orientation as the slope of p. Horn's algorithm is used to compute the slope, while the set of neighboring points is obtained by using Bresenham's algorithm. The height values are generated using the one-dimensional random midpoint displacement algorithm. An automated-sketch map version was also developed, where the map is generated based on another set of controllable parameters. The grid is initialized with the $PLAIN$ type and two algorithms are applied sequentially for $LAKE$ and $MOUNTAIN$ generation on the sketch map. This version has proven to be even more accessible to users, but the results lack the complexity since a small subset of terrain types are used. By performing user testing, the initial method has proven to be easy to use and intuitive, while also having above-average results.

While artistic input seems to give more satisfactory results than fundamental methods, the realism of the results depends heavily on the skill of the artist/designer, therefore it doesn't appeal to the general user base. Going in this realism direction, we find a simple yet elegant method described in [24]. It uses Value Noise, a derivation of Perlin Noise, in combination with real-world elevation data to generate landscapes that have similar height characteristics as the regions the elevation data was

sampled from. Approaching the problem statement from a geographical point of view, an analysis of elevation data (provided by the United States Geographical Service) reveals that on a small scale, elevation seems to have a somewhat normal distribution, but on a larger scale, abnormalities are more predominant. This analysis reveals what is already a consensus in geography, that elevation points are more correlated with points in their close vicinity rather than some other random point. In order to incorporate this behavior together with region characteristics in the final result, in the Value Noise function a custom distribution is used, obtained from real-world data, to generate the random height values, replacing the conventional normal and uniform distributions. This is done by building a height distribution table from data obtained from sources such as the USGS, reducing the original distribution of this data to a smaller, quantized version. The raw height values are first scaled to the range [-1, 1], then the number of height bands (intervals) is reduced. For each band $[\alpha, \beta)$, the number of values in the interval is given by $n = round(sl/p)$ where p is the number of data points, s is the number of initial elevation bands and l is the number of points where the scaled height lies in the interval $[\alpha, \beta)$. This method can be abstracted away from the user, allowing the use of different distributions from regions all around the world by simply downloading the data and feeding it to the program.

In some cases, the efficiency of a method is considered a high priority, and the standard methods tend to appeal to these needs, with procedural noise functions taking the lead by allowing maps to be generated on demand instead of being saved in memory. Although relatively fast on their own, the main practice when working with procedural noise functions is GPU parallelization. When time efficiency is the goal and relying on hardware is not an option, there is not much to do but look at other possible methods. One such method proposes using polynomials instead of the traditional noise functions [39]. In order to replace the traditional noise functions, a function h would need to associate a height value to each point of the domain, while also taking into account edge cases. Given this, the proposed form for function h is a multivariate polynomial of degree n:

$$h(x) = \sum_{a \in I} (c_a \prod_{k=1}^{D} x_k^{a_k}) \tag{3.1}$$

With I being the set of all vectors of the form $(a_1, a_2, ..., a_D)$ for which $0 \leq a_k \leq n$. A specific naming convention is used, $DdMmNn$, where d is the dimension, m is the order of the highest constrained derivative (usually $m = 1$ is used) and n is the degree of the polynomial. $D2M1N3$ is the minimum polynomial that is able to generate 2D terrain with height and gradient. For this specific configuration, the polynomial equation simplifies to:

$$h(x,y) = \sum_{i,j} c_{i,j} x^i y^j \text{ with } i,j \in \{0,1,2,3\} \tag{3.2}$$

Interpreting Perlin Noise as an order 5 polynomial, more specifically an order $2+s$ polynomial where s is the degree of the smooth step polynomial ($s = 3$ is the lowest order for the polynomial in order to satisfy constraints), it would seem that $D2M1N3$ would be a much better choice, however, Perlin allows for few computational steps due to its factorized form, yielding better performance. In order to compete with this, a derivation of $D2M1N3$ is introduced, zero-gradient $D2M1N3$. Choosing performance over generality, special gradient conditions are imposed resulting in a factorized form similar to the one Perlin Noise exhibits. While the zero-gradient variant is faster than Perlin Noise, it lacks the simplicity its slower adversary displays. Regarding result complexity and diversity, the general scheme $DdMmNn$ allows an arbitrary level of gradient smoothness. It is noted that the proposed method presents no difference in result quality when compared with the other methods (Perlin Noise, OpenSimplex Noise).

When given a considerable number of constraints for the generated results, the fundamental methods perform badly, since control is limited. With terrain generation being used in video games, the main goal is to convince the player the world they navigate is close to real life. But what about strategy games? The map should be as balanced as possible so no single player has an advantage over the others from a location point of view. This is a real problem for Real-Time Strategy games, more specifically the maps used in these types of games are pre-built, therefore players could rely on memorizing strategies that work best for each map, instead of relying on strategic thinking on the spot, and the generated maps are unbalanced and inferior to the hand-crafted ones. Taking inspiration from Conway's Game of Life [6], one recent attempt at solving this problem employs the use of Cellular Automata, first introduced by the father of computing, John von Neumann [22], in order to generate height maps that could be extended to be used in RTS games [48]. Usually most CA have a static rule set throughout the whole process. The aforementioned method uses deterministic and stochastic rules, applied sequentially. Between iterations, the structure of the grid is modified by doubling the resolution in both dimensions (a cell is expanded into a group of four cells). In order to simulate the different height levels, CA grids for different types of terrain are stacked on top of each other (the $MOUNTAIN$ layer is the only one left out since it's generated later in the process). Regarding a CA grid's rule set, for each cell only the vertical and horizontal neighbors are considered (the same as von Neumanns original version), and as in most CA, a cell could either be alive or dead (on or off). Gathering these constraints into a single rule set yields 10 possible neighbor-state combinations

($\{0,1,2,3,4\} \times \{0,1\}$). A deterministic set can be represented by a set of 10 boolean values, while a stochastic set can be defined as a set of 10 probabilities. Coming back to the stack of CA-generated grids, a height map is generated by adding for each alive cell from each layer a value corresponding to the layer type (e.g. if a cell is alive in the $GRASS$ layer the predefined value for that layer will be added at the position of the cell in the height map). Thermal erosion is then used to smoothen the map out before a stochastic erosion rule set, identical to the ones from the previous step, is applied to generate the $MOUNTAIN$ layer (Hydraulic erosion could be used instead). The last step in this process is porting the height map to the desired format used by your RTS game of choice, adding details such as rocks, trees, player markers, and resources. For evaluation, the authors used the method to create maps for an RTS game called Supreme Commander, a study was conducted with users testing human-generated maps and CA-generated maps. While the generated maps prompted players to come up with new strategies based on the terrain they were working with, the general consensus was that the human-generated maps were more aesthetically pleasing than the CA ones, furthermore, players reported that the CA maps were more balanced among both winning and losing parties.

3.3 Evolutionary Methods

In the field of Terrain Generation, any process that yields acceptable results in a reasonable time is considered a good solution, therefore the exact path to obtaining the result is not important as long as the aforementioned conditions hold true. The previous subsection presents methods that approach the problem statement from different angles, but the common denominator among them is that a rule set is established and the terrain is generated within those constraints. But what if we look at the problem from another point of view? Instead, we could think of a generated terrain as being a single point in the domain space of all possible terrain. Given this, the generation process would actually be a search in this space. This approach is adopted by a handful of landmark papers all using Evolutionary Algorithms as their search mechanism. Six papers have approached the problem statement with Evolutionary Algorithms in mind, all of them described in an extensive survey [31] that still describes the state of the art in the domain since the trend nowadays focused on Deep Learning based methods. The first known proposal for using EA in Terrain Generation is by *Ong et al.* [23][36], the work culminating in a program called *Terrainosaurus*. The algorithm consists of two steps. Firstly, generating a terrain outline by applying distortion to a user-generated sketch. And secondly, the evolutionary step takes place. The initial population is generated by sampling from user-provided data. In order to make changes to the terrain, control points are placed on the map. The chromosome

representation is a list of operators applied to each point, while also influencing the vicinity of the points. The fitness function measures the similarity of individuals to the given sample data. While this method produces satisfactory results, the major disadvantage of this method is the reliance on user-sourced data.

Taking inspiration from the fractal nature exhibited by the growth process of plants, *Ashlock et al.* [3] propose using an L-system, which is a type of formal grammar that has the ability to generate realistic models of natural patterns such as trees, plants, etc. A modified L-system is used in order to establish correspondence between the system and a height map. EA is used to fine-tune the parameters of the system, having the fitness function based on comparing the individuals with user-provided data, suffering from the same constraint as in the case of *Ong et al.*, having a considerable reduction effect on the explored space.

Walsh and Gade [43] proposed using EA to fine-tune the parameters of an already existing terrain generator. The genotype is represented as a set of parameter values each stored on 8 bits. Regarding the fitness function, they use a somewhat interactive method where the score is given by users in real-time. Crossover is used in order to speed up convergence. While no contribution is made towards a new method of generation, the idea holds a lot of potential if powerful generators are used.

In order to create height functions, *Frade et al.* [11][10][9][33] use a genetic algorithm having the genotype representation as a tree of operators, evolving by adding/removing operators from it. These can range from simple arithmetic operators to trigonometric functions or even handmade functions. Similar to the previous paper, a user-based fitness function was used initially, later experimenting with two fitness functions based on Accessibility (making the world easily traversable for the player) and Obstacle Edge Length (introducing obstacles for the player to navigate around) measurements.

Thinking of terrain generation for a video game as a multi-objective task, *Togelius et al.* [40] set out to first generate maps suitable for games, and only after that to create the terrain around the gameplay elements in the map. In order to do this, the map is initialized with flat values, and mountain spikes are raised together with ridges constructed on Gaussian curves. The information encoded in the chromosomes represents the characteristics of the mountains, such as the standard deviations, x, and y values for the mountain's position, and h, the height of the mountain. Multiple fitness functions are used to ensure a balanced playing field by having equal access to resources, spread out bases, and accessibility between these.

Raffe et al. [30] came up with a patch based solution. EA is applied to generate terrain from smaller patches of height maps. The square patches are obtained by user-provided sample data. This method uses a matrix of patch identification numbers as genotype representation combined with a

uniform crossover mechanism that works by copying the genetic structure of one of the parents in the offspring and probabilistically changing patches in the structure with the corresponding ones from the other parents' structure. A simple form of mutation is also used, randomly setting a patch to an arbitrary one.

3.4 Generative Adversarial Networks

Ever since AlexNet proved the superiority of neural networks over traditional machine learning methods for image classification in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [35] in 2012, classifier models have been strikingly successful, with results surpassing even those of the average human. At the same time as these models that were breaking records year after year, generative models were complex and had at most satisfactory results. In 2014, Ian Goodfellow proposed a simple yet elegant architecture, Generative Adversarial Nets [12], that aimed to harness the power of classifiers and channel it toward a generative task. Taking inspiration from the game theory concept of a minimax game, the architecture consists of two models in a contest with each other, a discriminator model D and a generator model G. The discriminator estimates the probability that a sample came from the real dataset or from the fake dataset generated by G. The goal of the generator is to fool the discriminator by generating samples closer to the real dataset. While training, they each get progressively better, up until the point where the discriminator is not able to differentiate between the real and the fake samples ($D(x) = 0.5$). After training, the discriminator can be discarded and what is left is a very powerful generator model. While initial results were not better than current methods, they did show that adversarial models have potential.

Since the only input given to the generator is a noise vector (often called latent vector or random vector), there is no real control of the output. Conditional Generative Adversarial Nets (cGANs) [21] are a variation of GANs that introduce the ability to condition the output. The discriminator and the generator can be extended to conditional models by adding some extra information y that can be from any modality (labels, images, etc.). In the generator, y is combined with the noise vector, while in the discriminator y and the sample x are considered as input.

The use of Convolutional Neural Networks in the GAN architecture proved to be extremely successful, the first iteration of GANs having used fully connected feedforward neural networks. Deep Convolutional Generative Adversarial Networks (DCGANs) [29] are a family of CNNs with certain properties that make them suitable for unsupervised learning. Some of these properties include the pooling layers being replaced by strided convolutions in the discriminator and fractionally-strided con-

volutions in the generator, batch normalization in both models, no fully connected layers in either model, *ReLU* activations for all layers except for the output (which uses *tanh*) in the generator, and *LeakyReLU* activations for all layers in the discriminator. The latent space can be thought of as a hyperdimensional space of features, therefore taking a walk in this space can result in semantic changes to the generated images, this being a sign that the model has learned relevant representations. One of the first examples of latent space linear arithmetic occurring in unsupervised models was presented along with the DCGAN family of nets, proving the ability of CNNs to accurately model the feature space (see Figure 3.3).

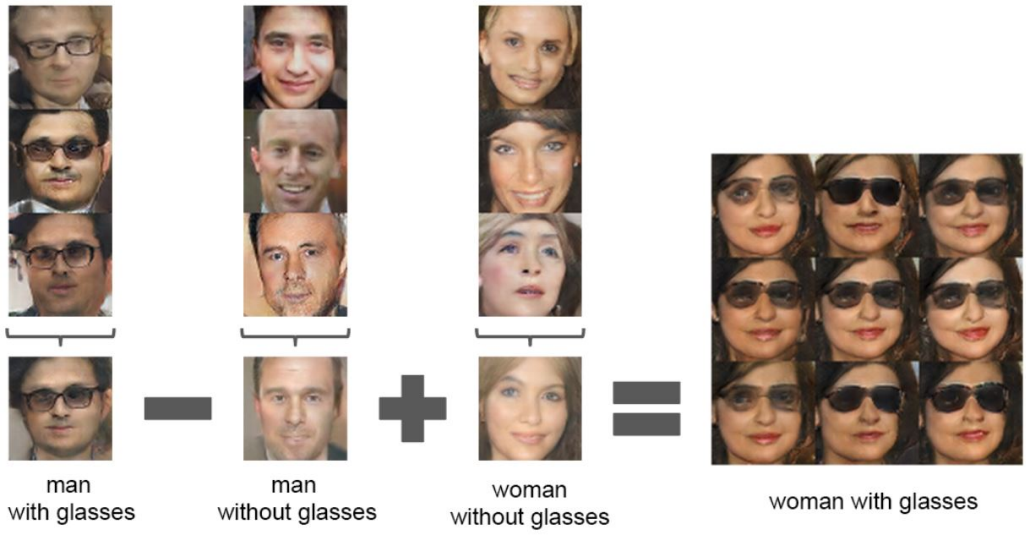


Figure 3.3: An example of linear arithmetic in latent space from [29]. Average points were taken in the latent space for 'man with sunglasses', 'man without sunglasses', and 'woman without sunglasses', the result of the equation representing the average point for 'woman with sunglasses'

Another interesting method to measure the generator's ability to learn significant representations was presented in [29]. The method involves predicting if an activation is on a window or not in the photo in order to generate images with and without the feature maps. This resulted in the generator replacing windows with other objects or removing them entirely.

Pix2pix [16] is a DCGAN architecture extended to a cGAN, conditioning the models in input images, used for image-to-image translation tasks. The generator in pix2pix uses a U-Net [34] based architecture, unlike past work, while the discriminator is a convolutional PatchGAN classifier. In past work, the GAN objective was used in conjunction with a traditional loss applied on the generator G. The authors used $L1$ loss, stating that $L2$ loss exhibited a blurring effect on the result when compared to $L1$. Instead of introducing noise as an input vector to G, pix2pix uses a different approach by employing dropout on the generator at both training and test time. A new strategy was used for the discriminator, PatchGAN. The model tries to classify each NxN patch as real or fake, running it in

a convolutional manner across the image and averaging all the results. The U-Net architecture used for the generator allows information to flow across the network throught the skip connections, hence its better than the encoder-decoder architecture (similar to U-Net, without the skip connections) in general, not necessarily specific to the cGAN case. Pix2pix proves its versatility through its ability to generate decent results with rather small datasets. Human evaluation of real and fake images was used along with the FCN-score. The latter method employs a model for semantic segmentation trained on real data. The images generated by G are scored based on the labels they were generated from and the identified labels. A few tests were conducted to determine the best patch size for the PatchGAN. $16x16$ PatchGAN yields good results, but it leads to tiling artifacts in the generated output. $70x70$ improves on this by removing the artifacts and maintaining the same quality. Scaling to the full image ($286x286$ PatchGAN, also called ImageGAN), yields less qualitative results, a probable cause being that the number of learnable parameters is greater, therefore the model becomes harder to train. The generality of pix2pix's approach is demonstrated by a series of translation experiments. The most notable ones are semantic labels to photos, architectural labels to photos, and map to aerial photos, all showing impressive results, using the same architecture more or less (in some experiments the architecture is slightly modified), the difference being only the data sets used.

The main downside of the pix2pix method is the fact that the dataset must be paired, thus making it harder to create large datasets. Shortly after pix2pix, an architecture called CycleGAN [47] took a different approach to the image to image translation problem, which allows the use of unpaired datasets. The goal of CycleGAN is to train a generator $G : X \to Y$ to generate photos indistinguishable from the real photos, identical to the classic GAN strategy, but CycleGAN introduces the use of a generator $F : Y \to X$ together with a cycle consistency loss in order to constrain the mapping and help with reducing modal collapse (the objective of the loss is to push $F(G(X)) \approx X$ and vice versa). CycleGAN also introduces discriminators two discriminators, D_X which distinguishes between images from X and $F(X)$, and D_Y, which distinguishes between images from Y and $G(X)$. There are two types of losses used for CycleGAN, adversarial loss, which is the classical GAN-loss applied to both mappings, and cycle consistency loss. Regarding the latter type of loss, from a mathematical point of view, the goal is to produce a pair of inverse function to each other, so practically for any image x from X, it should be possible to transform the image into y from Y with G, and bring back x by using F (the opposite should also be the same). This is the intuition behind the cycle consistency loss, and expressed in the mathematical form it would look something like this:

$$L_{Cycle}(G, F) = E_{x \sim p(X)}[||F(G(x)) - x||_1] \tag{3.3}$$

A version of this loss with the form of adversarial loss has been experimented with but no performance improvement was recorded. After experimenting, replacing the adversarial log-likelihood loss with least square loss seemed to lead to performance stability. Instead of updating the discriminators based on the latest generated images, a historic batch of 50 photos is used. An interesting additional loss, called identity loss, was used in the painting to photo experiment in order to preserve key aspects that should be translated to the other domain. The reasoning behind this type of loss is that having given a photo from the codomain as input to one of the generator, it should output the same photo since it is already in the codomain. This idea is expressed in the following loss function

$$L_{Identity}(G, F) = E_{y \sim p(Y)}[||G(y) - y||_1] + E_{x \sim p(X)}[||F(x) - x||_1] \tag{3.4}$$

While CycleGAN has an advantage over pix2pix regarding the dataset, the former might fail for more complex translation domains.

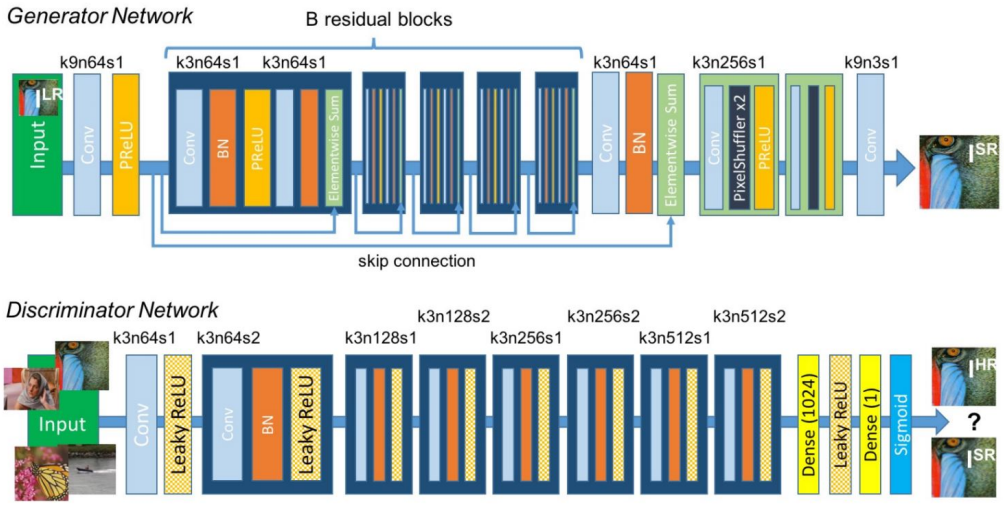


Figure 3.4: Generator and Discriminator architectures taken from the original SRGAN paper [19]

Pix2pix and CycleGAN both generalize the task of image-to-image translations, demonstrating their ability to find translations between a diverse set of domains. While they can be used by anyone with no knowledge about the inner workings of the models, having to only change the datasets, they are not expected to perform the same for any pair of datasets. Some translations are more complex, therefore harder to represent by the models. One such translation is Super Resolution, a conversion of low-resolution images to high-resolution. The SRGAN architecture [19], uses insight specific to the translation, yielding results more qualitative than generalized methods. In the effort of generating images with a 4x up-scaling factor, SRGAN uses a perceptual loss, composed of a content loss and an adversarial loss. Regarding the former, pixel-wise mean square error loss was widely used for image

super-resolution on which state-of-the-art methods relied. However, this type of loss caused overly-smooth textures. SRGAN uses a different type of loss, VGG loss, defined as the Euclidean distance between the feature representation of a reconstructed image and the reference image. A separate, pre-trained VGG network is used (in general the VGG-19 model), trained on a large dataset used for image classification such as ImageNet [35]. The adversarial loss is the same classic GAN loss described before. The generator architecture employs residual blocks similar to those used in ResNet [14], allowing for a deeper network (16 residual blocks are used in [19]) and thus having the ability to represent features of higher abstraction. The discriminator architecture is fairly standard, having a sequence of convolutional layers (convolutions followed by Batch Normalization and activation such as Leaky ReLU), a dense layer after the last convolutional layer, and finally a dense layer with one value as output (for an in-depth look at both architectures see Figure 3.4). The method used to measure SRGAN's performance is called the Mean Opinion Score (MOS), based on the feedback given by a group of human evaluators. Compared to state-of-the-art methods, SRGAN scores higher than all of them.

3.5 GAN-based Methods

The advent of Generative Adversarial Nets opened a new dimension in the field of Terrain Generation. Shortly after the original paper on GANs was published in 2014, their potential use in generating realistic representations of the real world was recognized. Among the first methods to harness the power of this architecture made use of heightmap data provided by NASA's Visible Earth project to obtain artificial heightmaps that closely resembled elevation patterns from the surface of the Earth [5]. Random 512 by 512 patches from a world heightmap are used to generate random artificial heightmaps, along with corresponding texture maps. In order to achieve this, a DCGAN model is used, and in order to infer the texture map, a pix2pix model is employed. In the paper, an example of a linear interpolation between two heightmap outputs is presented as a means to introduce the idea of control over the heightmaps by leveraging the power of the latent space. Apart from having slight stability issues in training, the generated outputs seem to sometimes contain some sort of artifacts. To address this, a Gaussian kernel convolution is used to give the image a slight blur in order to smoothen out these artifacts. The pix2pix model, used for texture generation, outputs images that roughly match their respective maps. An important observation was made during the training stage, that the model would seem to confuse the desert and the snow textures, often using a combination of both when only one should be applied. A possible improvement pointed out by the authors addresses the out-of-sync

behavior between the two models, suggesting that further research should concentrate on a method to train the models together in order to form a deeper connection between the heightmaps and the textures.

Later, the idea of using satellite images as a source of terrain features that could be used to synthesize realistic representations became more common. One method makes use of all the types of data that are usually recorded by satellites in orbit. Using a ProGAN as a base architecture, the method aims at generating multispectral satellite images [1]. The main advantage of the ProGAN architecture is the fact that during training, the resolution of the images is increased progressively, a strategy that has shown improvements in the stability and speed of the process. For updating the weights, a strategy similar to that of the Wasserstein GAN with gradient penalty is used. Due to the fact that the original ProGAN architecture was designed to handle 8-bits-per-pixel images, modifications were made to support 16-bits-per-pixel images in order to accommodate for the multispectral input data, which encompasses more bands than the ones of visible light.

Regarding the dataset, SEN12MS, a large0-scale dataset, was used, consisting of 180662 image triples. From the triples, only the images taken by the Sentinel 2 satellites were used, and preprocessed to remove any cloud formations. Given the fact that the Sentinel satellites cover a majority of the Earth's surface, and there are images taken during different seasons, the diversity of the dataset in ensured. The images are 256 by 256 in size, having 13 bands.

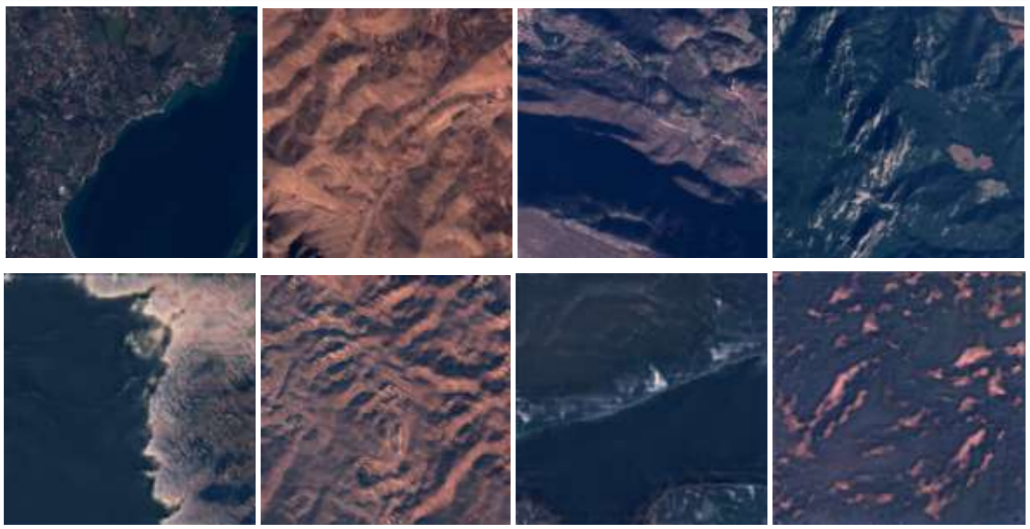


Figure 3.5: Results generated by a ProGAN (adapted from [1]). The top row images are real satellite images, while the bottom row images are generated using the model

For the experiments, the whole dataset is considered for training the GAN. The training was closely monitored, the process being stopped when no visible improvements were observed. When doubling

the resolution of the images, part of the ProGAN strategy, the batch size is halved. The results were evaluated by visual analysis of the image quality and the structure of the spectral bands. For a better assessment of the results, the multispectral images were reduced to the RGB bands (3.5). The Generator model is able to mimic the SEN12MS images up to a certain degree. Moreover, a spectral analysis, of the rest of the bands indicated that for a set of specific textures, the generated bands seem to match the structure and patterns of those from real-world data.

At the time of writing, the method called GAN-terrain [42] produces the most accurate representations of terrain, obtained from satellite images. The input, having a simplified structure, consists of 2D scatter plots containing so-called Points of Interest (PoI) in the form of an image. The term used to describe these images is "altitude image". The output is a 2D image resembling satellite images, the colors representing diverse real-world terrain textures.

The main advantage of the method is the fact that after the Generator model is obtained by training the GAN, inputs are relatively easy to generate using any image processing tools. Even with the arbitrary input images, the output still generates modestly realistic images (3.6).

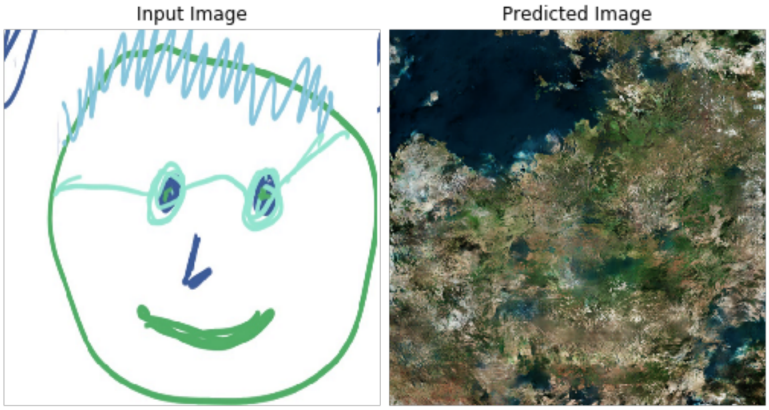


Figure 3.6: An image of a smiley face cartoon transformed into a realistic terrain image by using the GAN-terrain method (adapted from [42])

GAN-terrain consists of training a pix2pix model to learn the mapping from the 2D scatter plots to the realistic images. At inference time, similar altitude images can be fed to the Generator to obtain 2D terrain images with rich colorful textures (prominently displayed in Figure 3.7). For training, geographic patches were collected, and for each patch, a random number of PoIs falling within the patch were collected along with the corresponding satellite image of the patch. The PoIs are used to generate an altitude image. The initial image is a white canvas to which these points are added in the geometrically identical positions from the patch. The color of each point encodes the altitude value. The value could be interpreted on a grayscale or as an RGB value. The visual features in the realistic images are correlated with the color-coded values of the PoIs. The Generator automatically fills in

the rest of the details, built around the position and information of each PoI. Augmentation is also explored as a means for diversity in the output, performed by applying rotations on the input images only. For the dataset, 11.2 million world PoIs were collected to generate a dataset of 4300 patches from around the world, and Microsoft Bing Maps API was used to obtain the corresponding satellite images.

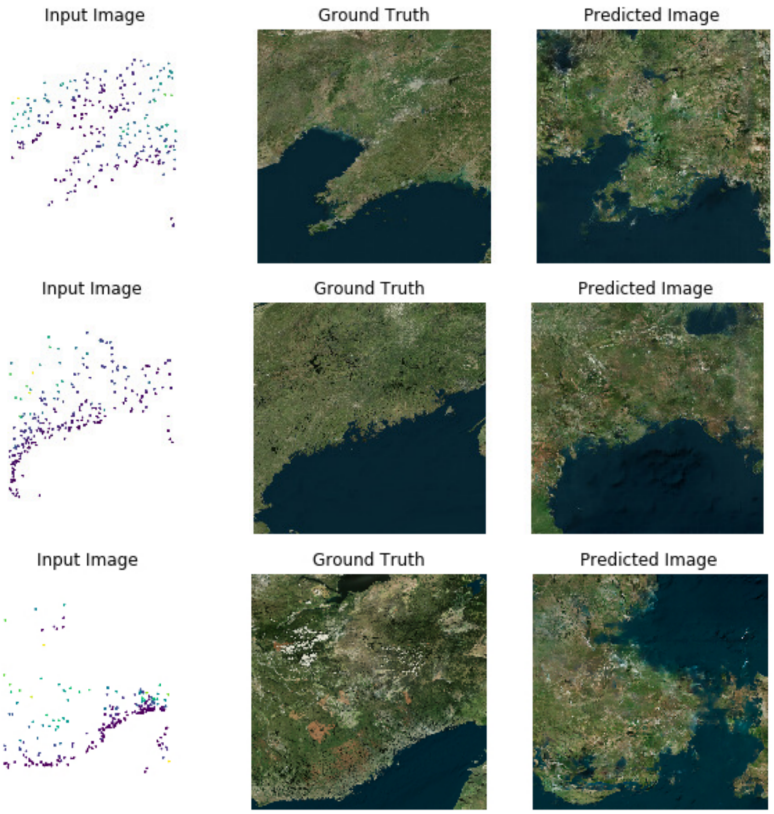


Figure 3.7: Four examples of GAN-terrain input/ground-truth/prediction triplets (adapted from [42])

The network was trained on a set of 3000 images for 300 epochs, and tested on the remaining 1300 images. Both color and grayscale models were trained, with the color version yielding far better results, therefore further experiments continued with it as a base model. The evaluation was done using the Normalized Histogram Intersection similarity score (bounded between 0 and 1) between the generated GAN-terrain prediction and the ground truth image. The overall score on the test set was relatively high, 0.7665. Metric scores such as plausibility and correspondence were obtained by subjective observer testing on the set of 40 images, half real and half fake. For each image, each observer had to give a value between 1 and 5 for both metrics. The main takeaway was the fact that the images were indistinguishable from one another for human observers. Further evaluation was done for the models with and without augmentation by using the GIST global image description vectors. The analysis of the overall values revealed a tradeoff between semantic concordance and output diversity.

Chapter 4

Proposed approach: pix2sat

Looking back at state-of-the-art methodologies that use Generative Adversarial Nets, the main focus seems to be on the quality and complexity of the generated outputs, focusing on capturing terrain features from a certain niche, such as multispectral information or elevation data. Their predecessors, methods that used evolutionary algorithms and rule-based algorithms, seemed to incorporate in their structure meaningful and intuitive control over the generated landscapes. While there is a certain degree of control present in the current GAN-based solutions, it does not seem to cater to the needs of the average user, instead focusing on broader use cases.

As a general overview, there seems to be a considerably large gap in the current state of the Terrain Generation field. Pre-AI solutions lack realism in their outputs since classic procedural methods can only try to simulate real-world environments, but they compensate by allowing for different degrees of control, either by parametrization or crude sketch-based inputs. AI-based solutions on the other hand have the advantage over quality since models can accurately capture complex terrain details in their feature maps, but are less concerned with providing a way for the user to guide the generation process. The following methodology is designed to fill this gap by combining classical procedural terrain generation with modern GAN-based landscape generation, built around the idea of reintroducing the focus on user experience in the domain of modern Terrain Generation.

4.1 Methodology

Taking inspiration from the chronological flow of the evolution of Terrain Generation, the first step in the proposed approach (4.1) is generating an initial height map. In order to obtain the map, which in essence is a matrix of values from 0 to 1, a procedural noise function can be employed, such as Perlin, Value, ValueCubic, or OpenSimplex. Any one of these can be used, with the only constraint being

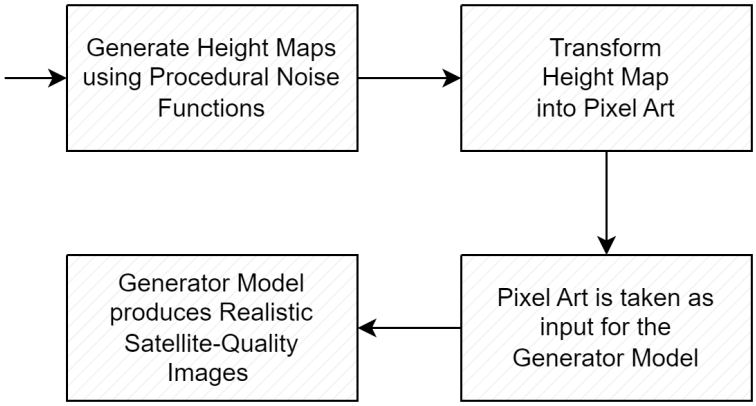


Figure 4.1: Proposed Methodology

the size of the obtained height map, which should be $(32, 32)$. Having generated a height map with the necessary size property, the next step would be transforming it into an image, resembling pixel art of a colorful landscape (4.2). The values from the map also called elevation points, are mapped to a certain color that represents an elevation-specific terrain texture. In order to achieve this, the interval $(0, 1)$ is split into sequential sub-intervals, one for each color. To provide a reasonable amount of texture diversity, the set of colors, which will be referred to as the color palette moving on, contains 7 colors that represent generic terrain types. The colors, taken in order from elevation 0 to 1, are Dark Blue (Ocean), Beige (Sand), Light Green (Hardwood Forest), Light Brown (Rough Hill), Dark Green (Conifer Forest), Dark Brown (Mountain), and White (Snow). Having mapped each elevation point in the height map to an RGB value from the color palette, we effectively obtained a matrix of colors. To synthesize the final pixel art image, the color matrix of size $(32, 32)$ is up-scaled by replacing each color value with a constant color sub-matrix of size $(8, 8)$, obtaining a final matrix of size $(256, 256)$, which is essentially our pixel art image, having 'pixels' of size $(8, 8)$.

$$\begin{bmatrix} 0.336288 & 0.313749 & \cdots & 0.603502 & 0.56443 \\ 0.337361 & 0.316478 & \cdots & 0.58653 & 0.550057 \\ 0.34851 & 0.32921 & \cdots & 0.571232 & 0.538802 \\ 0.368976 & 0.351231 & \cdots & 0.557611 & 0.530456 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0.495421 & 0.475395 & \cdots & 0.636905 & 0.618829 \\ 0.49555 & 0.469544 & \cdots & 0.667557 & 0.648769 \\ 0.336288 & 0.313749 & \cdots & 0.603502 & 0.56443 \\ 0.337361 & 0.316478 & \cdots & 0.58653 & 0.550057 \end{bmatrix} \longrightarrow$$

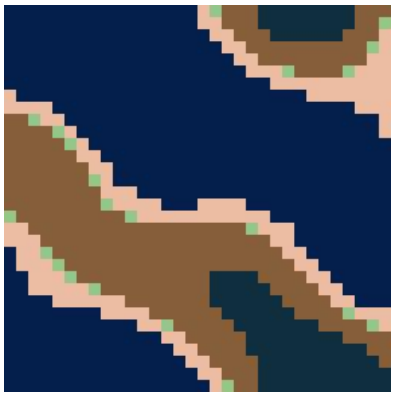


Figure 4.2: Transforming a height map into pixel art

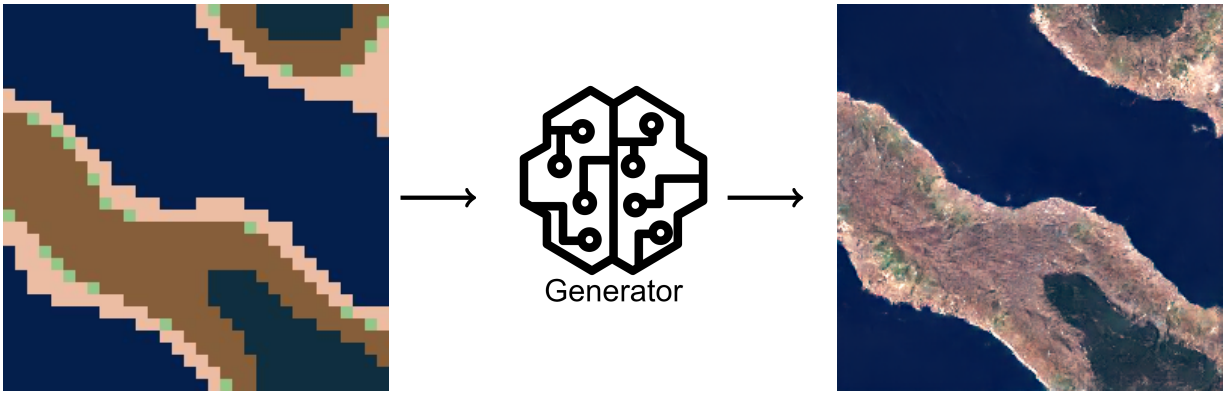


Figure 4.3: Applying the generator on the pixel art image, obtaining a realistic terrain image

Moving forward, we have reached the point where the GAN Generator model is employed. The Generator is designed and trained in such a way that it is able to take the pixel art image as input and transform it into a realistic satellite-quality image having the same size as the input image, $(256, 256)$ (4.3). The chosen model architecture uses images of this size, therefore all the aforementioned dimensions were chosen to accommodate this.

Having described the general methodology, you might be wondering *'Where is the control you were talking about?'*. Well, the central and most intuitive point of control is between the pixel art generation and feeding it to the generator. In between, the image can easily be modified, in an image editing program, using the predefined color palette by changing the colors of the 8 by 8 blocks. It would be somewhat challenging to do this using a program like Paint, but a simple GUI application, which allows the user to change the color of a single 'pixel' in the pixel grid image, can be built to streamline this editing process. Moreover, if artistic input is not enough, variables such as the procedural noise function and its parameters and the color sub-intervals can be adjusted in the initial step to modify the resulting pixel art image, offering an unparalleled level of control and customization.

4.2 Model

The problem statement can be interpreted as image-to-image translation, basically, this task would require a model that is able to transform an image from a domain X into another domain, Y, where the X domain is the space of all pixelated images of landscapes using the predefined color palette, and the Y domain is the space of all satellite images. The model can be trivially defined as:

$$f : X \to Y \tag{4.1}$$

Choosing a GAN architecture for this task requires an in-depth analysis of the actual translation.

After a general review of the state-of-the-art models that would be able to perform such a task, three contenders for the position of the base architecture stood out, pix2pix, CycleGAN, and SRGAN.

The first model, pix2pix, has proven itself to be a powerful translation architecture given the diversity of translations it is able to capture. The original paper [16] presented a wide range of domain pairs that pix2pix was able to bridge the gap between (4.4), solidifying its status as the main contender for the base architecture. One downside of pix2pix is the fact that in order to train the model, a paired dataset is needed, and sometimes it is hard to obtain.

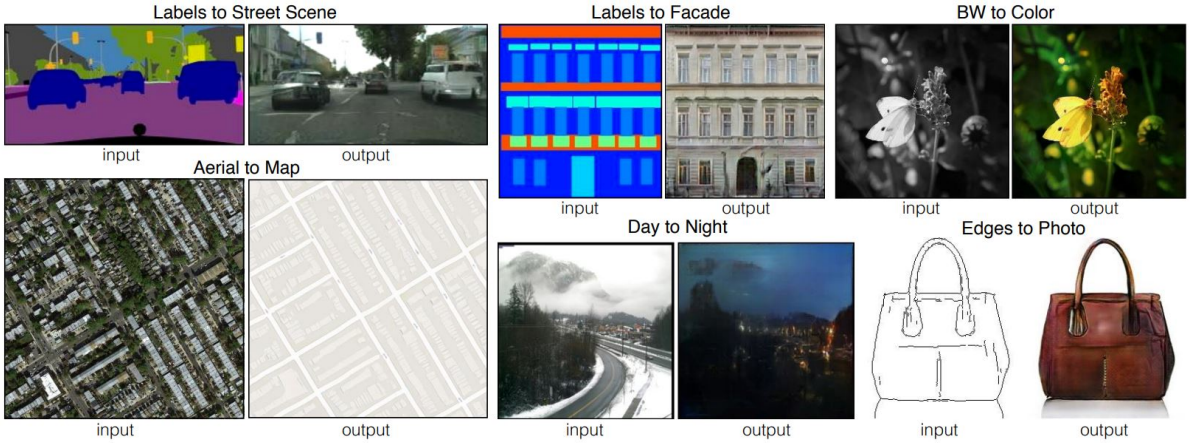


Figure 4.4: Translation examples taken from the original pix2pix paper [16]

The next contender is CycleGAN, inspired by pix2pix, it builds on the idea of improving image-to-image translation by providing a way to perform this task without a paired dataset, unlike pix2pix. Powered by an interesting trick, cycle consistency, it yields a performance close to that of pix2pis. The main reason this model was considered is the fact that in the case a paired dataset would be difficult to create, CycleGAN would be the next best architecture.

The last model, SRGAN is specifically designed for single-image super-resolution. The reason this model was considered is the fact that the translation problem at hand can be viewed from a super-resolution point of view. Using SRGAN, the 'pixelated' images are up-scaled to a much more detailed version, which is a broad description that fits the translation task.

After a series of crude experiments, training each model on a small handcrafted dataset, the model that showed the most potential in the visual quality of the generated images was, indeed, pix2pix, with CycleGAN coming in close second place. These models were both considered, until the viability of a paired dataset was demonstrated, as detailed in the next section. Therefore, pix2pix was ultimately chosen to be used as the base architecture.

4.3 Dataset

The quality of the resulting output images is highly dependent on the quality and structure of the dataset. Essentially, the dataset is comprised of 2 subsets, the input images, the pixelated landscape images, and the output images, the real satellite images. Given the nature of the translation objective, it is only logical that a paired dataset would yield more compelling results compared to its unpaired version. Starting with the real satellite images subset, the images were obtained by leveraging the use of the Google Earth Engine API [13], a cloud-based tool that enables the processing and analysis of large geospatial datasets. The API provides access to a diverse range of datasets, from surface temperature to land cover, the actual images representing a small part of the available data. Having examined all the available datasets, the one that stood out was Sentinel.

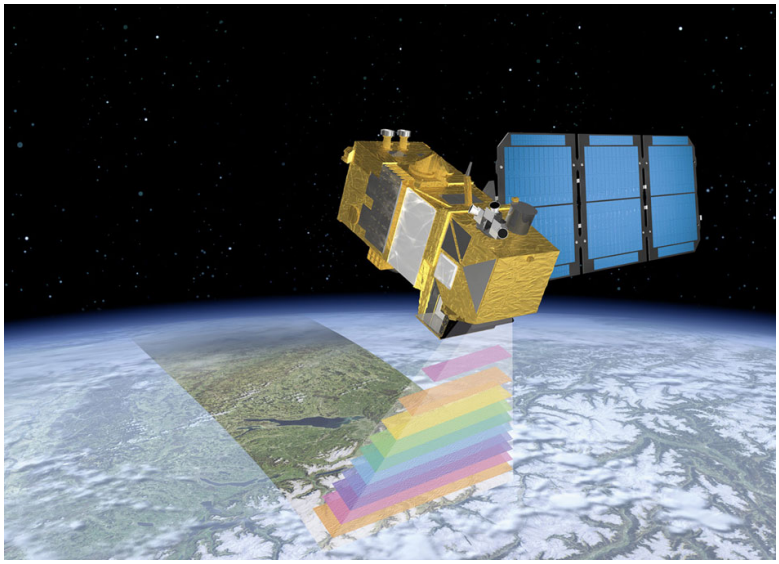


Figure 4.5: Sentinel 2A Satellite. Image courtesy of the European Space Agency (www.esa.int)

The Sentinel satellites (4.5), Sentinel-2A, and 2B launched by the European Space Agency (ESA), provide high-resolution optical images, from an altitude at which terrain features seem to be highly detailed. The API provides a multitude of mechanisms one could use to build a highly customizable dataset. The initial iteration of this set of images was sampled from random locations on the globe, which immediately proved problematic. Since the Earth is 70% covered by water, the majority of images did not include any land. A first attempt at solving this was to constrain the coordinate generation by defining rectangular regions from the south of Europe and the north of Africa, as bounds for the latitude and longitude. These regions were chosen for their diversity, having mountainous regions from the north of Italy, islands from the Greek archipelago, deep forests from the Balkan area, and

deserts from Tunisia and Egypt. Although the number of images containing only water bodies was reduced, they still represented a large percentage of the dataset. In order to have complete control of this number, the power of the Earth Engine API was leveraged. The MODIS LandCover dataset contains information regarding the distribution of land and water on the surface of the globe, from which segmentation maps can be obtained. These maps were introduced as filtering criteria, basically for each image, the segmentation map is checked in order to determine the presence of land, and in the contrary case, other locations are chosen until one satisfies the conditions. The transition from water to land represents an important feature that should be present in the dataset, therefore the presence of water is also checked, in 50% of the images. To further improve the quality of the images, a cloud percentage of at most 5% was introduced as a constraint, to reduce the possibility of cloud-like artifacts in the model output as well as the possibility of the model confusing clouds with snow and vice versa.

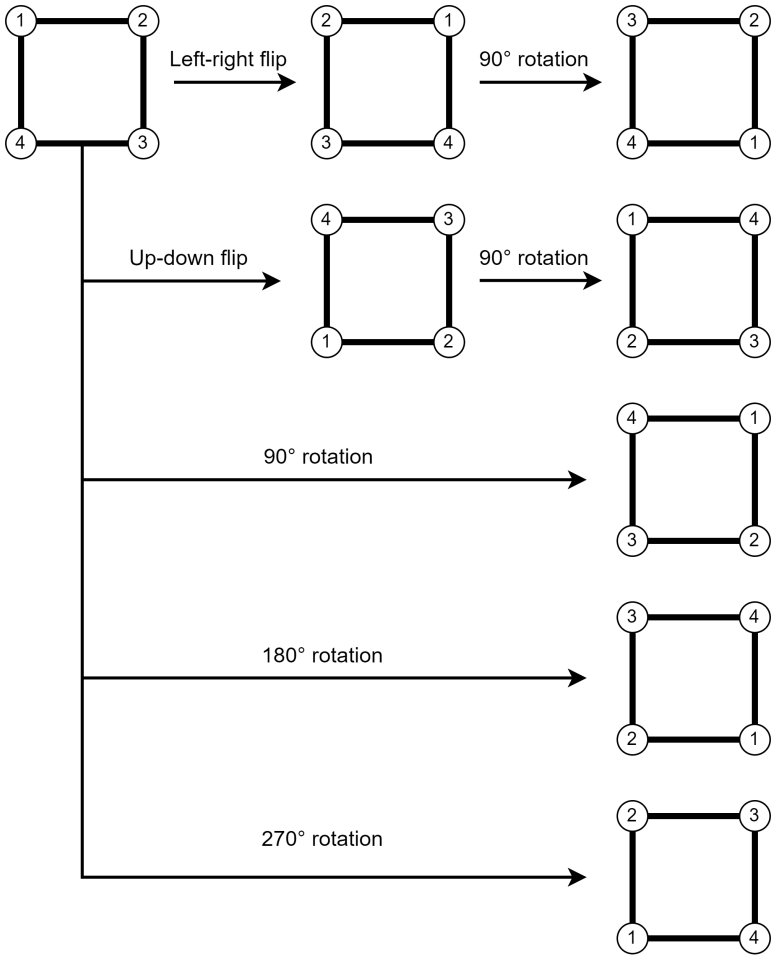
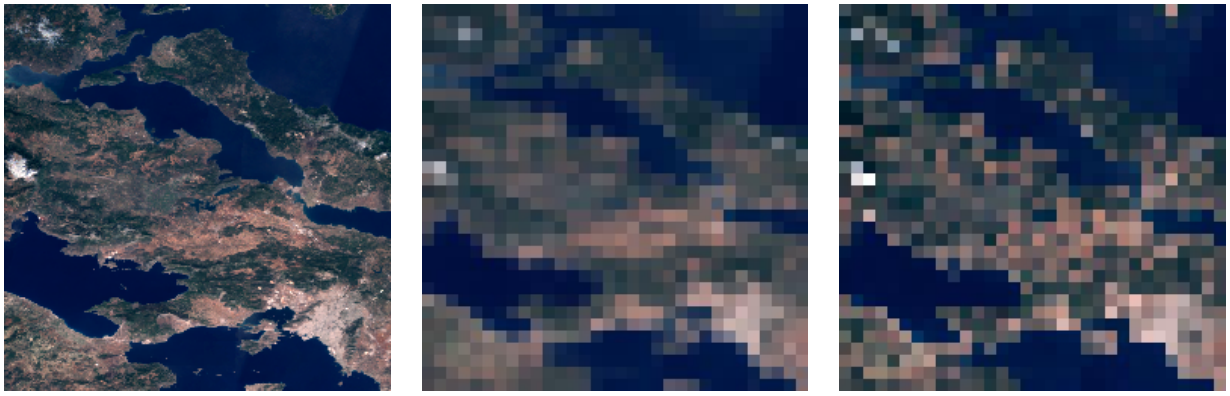


Figure 4.6: Data Augmentation scheme used to expand the dataset

Due to the number of constraints imposed on the images, downloading a single image of size

$(256, 256)$ takes approximately 20 seconds on average. Having this bottleneck in the downloading process resulted in obtaining a set of only 4000 unique images. For image-to-image translation tasks, this dataset size would barely qualify as useful, therefore data augmentation, consisting of rotations and flips (4.6), was used to increase the size by a factor of 8 to a final size of 32000 images.

Having dealt with obtaining the satellite images, the next concern was obtaining the corresponding pixel art versions of the images. This translation is effectively the inverse of the one the model would eventually learn, therefore it would need to preserve the information from one domain to the other as accurately as possible. The first step in this process is to pixelate the images using a downscaling algorithm in order to obtain the necessary structure with pixel blocks with size $(8, 8)$. The two algorithms that were considered are Nearest Neighbors and Average Color. The former takes the top-leftmost pixel in a given block and sets the rest of the pixels in the block to its color, while the latter computes the average color of the block and sets all the pixels from it to the average. Comparing visual results, both methods seem to produce results that contain the overall information of the input, but Nearest Neighbors seems to produce noisy results, while Average Color presents smoother and more defined results, as seen in figure (4.8). Therefore, Average Color was chosen for the downscaling step of the process.



(a) Original (b) Average Color (c) Nearest Neighbors

Figure 4.7: Downscaling(Pixelation) algorithms comparison

The second and last step in obtaining the pixel art subset consists of taking the downscaled version of the original image and normalizing each block's color to one of the 7 from the predefined color palette. While developing this step, the RGB values of the color palette were calibrated based on the colors present in the set of satellite images. Basically, normalizing consists of computing the distances from the color of the block to the colors from the palette and replacing the original color with the closes one to it. In the current context, the distance represents how close the colors are perceptually, therefore the RGB colors would need to be converted to a color space in which the visual difference between

two colors corresponds to the spatial distance between them. The chosen color space is CIELab since it was specifically designed to represent the way humans visually perceive colors.

The final output of the process defined above yields outputs that preserve the semantic features of the original satellite images, while also being part of the distribution of images generated using the color palette and procedural noise. The final dataset contains 32000 satellite images and their corresponding pixel art images.

(a) Satellite Image (b) Intermediary Downscaled Image (c) Final Pixel Art Image

Figure 4.8: Generating the corresponding pixel art of a satellite image

As an optimization of the dataset generation process, instead of applying the previously defined process to the final 32000 images, the original 4000 satellite images are processed to obtain their pixel art versions, and the augmentation is applied afterward to both sets.

Chapter 5

Experiments

The initial experiment validates the proposed use of the pix2pix GAN architecture to translate the pixelated landscape images into highly-detailed satellite images. Given the generalization nature of pix2pix, one could make the case that modifications can be made to the base architecture that would improve the performance on a specific translation task. Therefore, additional experiments are performed in order to determine specific configurations which could yield more qualitative results.

5.1 Specifications

In order to evaluate the performance of the models, the dataset is divided into two subsets: a training set comprising 90% of the data and a test set comprising of the remaining 10%. Regarding training, all the models are trained for 50 epochs, with a batch size of 32. For both the Generator and Discriminator models, Adam Optimizer is used with a constant learning rate of 0.0002 and momentum $\beta_1 = 0.5$, as well as Batch Normalization. Additionally, the Generator contains Dropout layers with a probability of 0.5. One deviation from the pix2pix methodology is the fact that no preprocessing is done on the images. All models were trained on an NVIDIA GeForce RTX 3090 GPU.

5.2 pix2pix Experiment

The purpose of this experiment is to validate the use of pix2pix as a viable architecture for the given image translation task. Early in training, the model seems to make out the general semantic structure it should represent, although textures such as ocean surfaces are not learned initially, therefore grid-like artifacts are present in their place (5.1).

Figure 5.1: Training image from step 180800 from the pix2pix Experiment

After a few epochs have passed, the model has gone through the textures enough times to come up with visually aesthetic textures to fill in the gaps. As seen in figure (5.2), given a large area of constant color, the model is able to fill it with complex and coherent terrain structures that not only adhere to the type of texture the color is meant to represent but also contain patterns that seem natural. There are some small visual artifacts present in the form of white dot clusters scattered around the terrain. Analyzing the real image reveals the fact that these artifacts could be caused by the sun reflecting from human-built structures directly into the lens of the camera aboard the satellite.

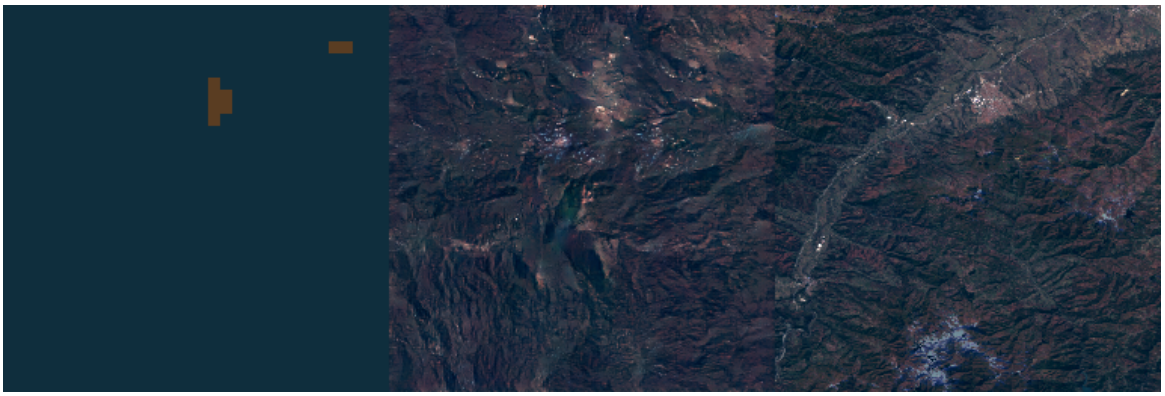


Figure 5.2: Training image from step 404800 from the pix2pix Experiment

In the middle of training, there is already evidence of smooth transitions between textures, without any noise whatsoever (5.3). Although the textures seem to blend in a more harmonious manner, the aforementioned white dot cluster artifacts are still present, in the same manner as before. Similar to state-of-the-art models that deal with desert and snow textures, distinguishing between the two seems to be challenging for the model. The short distance between the two texture colors in the color space could be the main factor of confusion.

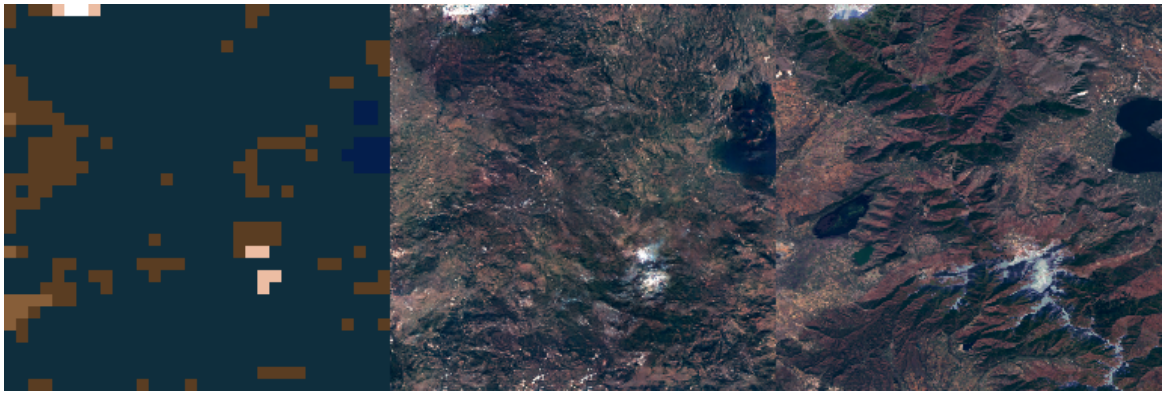


Figure 5.3: Training image from step 1085600 from the pix2pix Experiment

Having completed the training, the final model seems to have improved significantly. The white dot artifacts as well as the grid-like artifacts are entirely removed. The resulting outputs are extremely similar to their real counterparts aesthetically speaking and also respect the semantic structure imposed by the pixel art input. Although there are no major artifacts left, there is a specific type of small artifacts present in transitions from land to ocean. These appear as small slivers of land where there should be none (5.5).

Figure 5.4: Training image from step 1393600 from the pix2pix Experiment

Analyzing the loss graphs reveals that the training process was stable, with the GAN loss slowly decreasing. The Discriminator losses increased significantly at the beginning, later steadily oscillating around a plateau value, giving the Generator the opportunity to learn, as shown by the L1 loss of the Generator which after the 400k step mark reduced linearly.

To fully grasp the performance of the model, samples of pixel art were generated using the previously defined method based on procedural noise (5.6). Analyzing these outputs indicates that the model is able to generalize and is not overfit on the training data. Just like the textures from training, the result

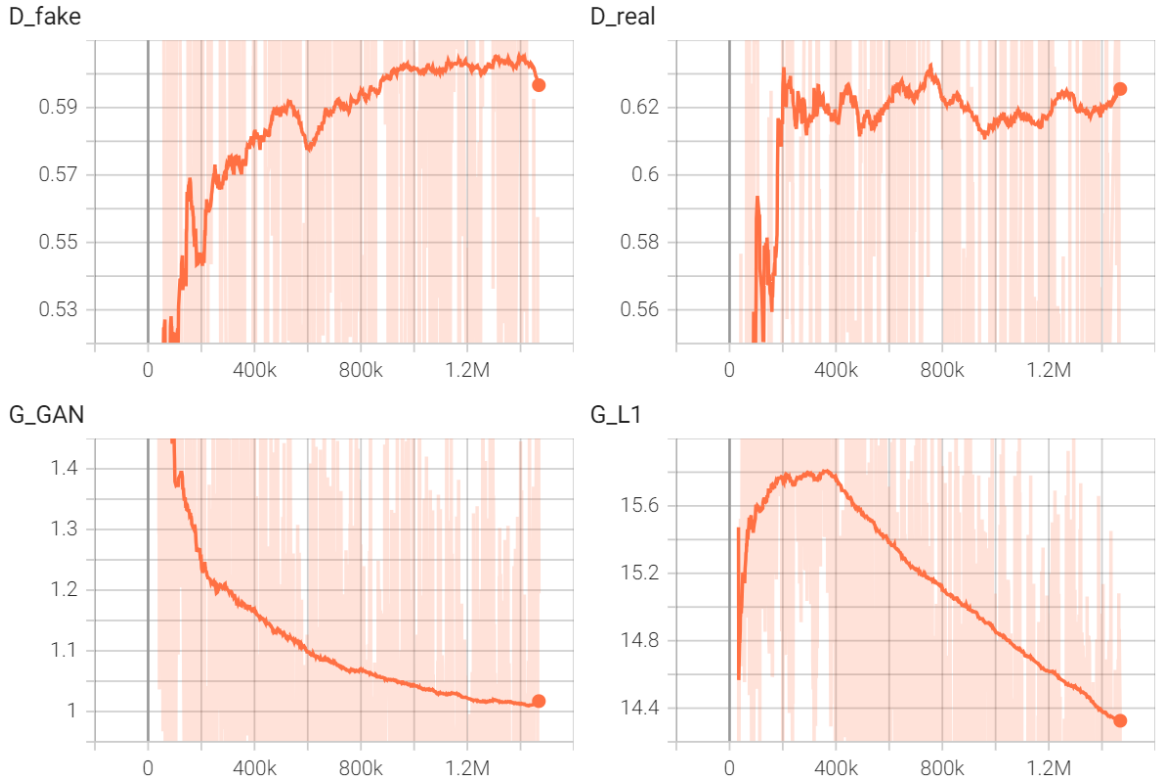


Figure 5.5: Loss graphs for the pix2pix experiment

Input Pixel Art Output Realistic Image

Figure 5.6: Example of a realistic image generated from procedurally generated pixel art using the pix2pix model

is highly detailed, filled with patterns that look natural, and achieving a balance between respecting the structure of the input and introducing diversity in the landscape. A problem that seems to persist is the presence of the previously identified sliver artifacts, which is not necessarily a significant flaw since it does not ruin the overall aesthetic of the images, but removing them would lead to more

polished and qualitative images.

The overall results of the base experiment solidify the proposed use of pix2pix for the image-to-image translation at hand. The following experiments aim to further improve the performance of the architecture as a whole, by adjusting its structure to better fit the specific task at hand.

5.3 ResNet Experiment

Regarding the Generator model from the pix2pix architecture, a U-Net model is used. U-Net was initially proposed for the purpose of image segmentation in the biomedical field [34]. The overall structure of U-Net resembles an encoder-decoder type of architecture, distinguishing itself from other similar models by employing the use of skip connections. These connections are present between corresponding layers from the encoder and the decoder parts of the model. Considering its use as a Generator in pix2pix, the skip connections might allow early features extracted from the input to have a greater impact on the output, hence the skip connections could be the main reason why the output is semantically similar to the input.

Based on this reasoning, other architectures that present this type of information flow through the network might compare or even show greater performance than the current model. A close relative of U-Net is ResNet [14], a model that was initially created to solve the problem of vanishing/exploding gradient, allowing for deeper networks to be trained in image classification tasks, more specifically in the context of the ImageNet competition, which it won in 2015. In some particular cases, the outputs seem to be constrained by the pixelated nature of the input, and sharp edges can appear, making the generated landscape seem unnatural. This may be caused by the initial skip connections, which could theoretically pass the initial rigid structure of the inputs as feature information to the final output. Unlike U-Net, the skip connections used in ResNet are applied sequentially (5.7), therefore removing the potential cause of this anomaly in the structure of the results.

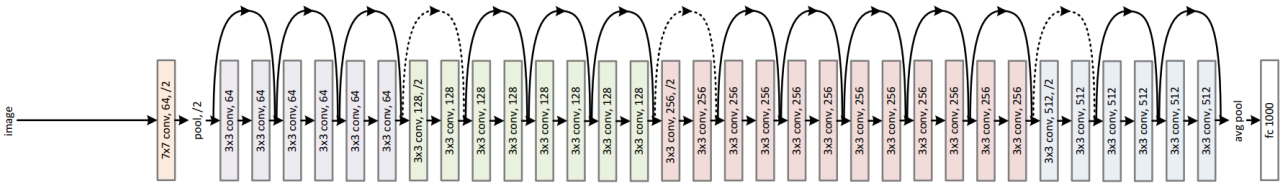


Figure 5.7: Architecture Diagram of ResNet from the original paper [14]

Given the previous hypothesis, the aim of this experiment is to verify if ResNet is capable of generating images with the same quality as U-Net does, and moreover to establish if this model architecture

indeed solves the aforementioned structure anomaly in the outputs. Setting up this experiment, the default U-Net Generator from pix2pix is replaced with a ResNet Generator consisting of 9 residual blocks. The number of residual blocks was chosen based on hardware limitations. Adding more residual blocks would increase the training time to unacceptable periods of time.

Similar to the initial experiment, the model seems to make out the general structure of the landscape it should depict. At this stage, the model seems to keep up with the performance of its U-Net counterpart, but with slight noise in the landscape patterns. Another visible similarity between the two generators is the fact that the ocean texture presents heavy artifacts, evidently displayed in (5.8).

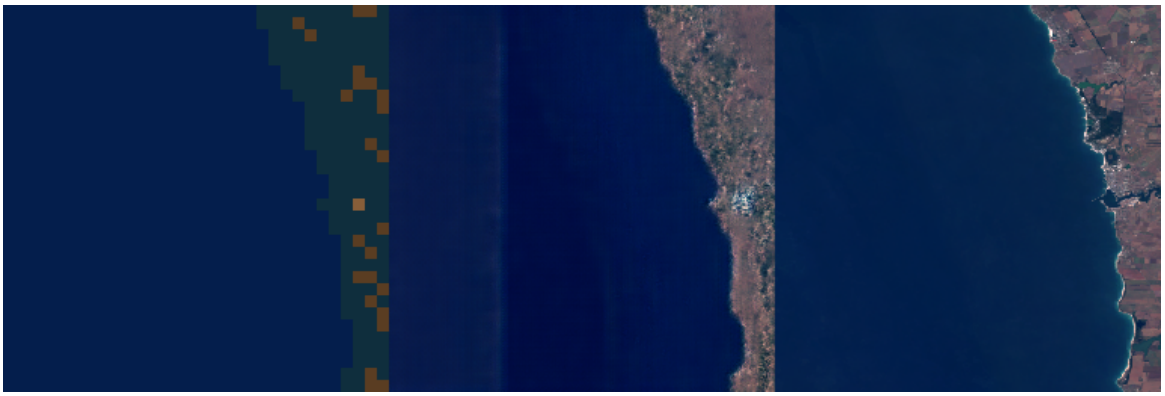


Figure 5.8: Training image from step 180800 from the ResNet Experiment

At step 404800 (5.9), it is clear that ResNet is struggling to apply complex features based on the input textures. While the output textures do appear to be noisy and grainy, there are signs of crude pattern replication happening, a sign that the model could eventually learn at the level of its counterpart, but at a much slower pace.

Figure 5.9: Training image from step 404800 from the ResNet Experiment

Although there is a persistent grid effect on the images throughout the training process outputs,

by step 1393600 (5.10) the results are as detailed as the ones generated by U-Net. They exhibit the same range of diversity in texture transition and the terrain features seem as natural as can be, but the initially observed noise effect is still present, therefore they are no match for the high-resolution outputs of the original Generator. Moreover, the confusion between the snow and desert textures is present, to a greater degree than encountered before.

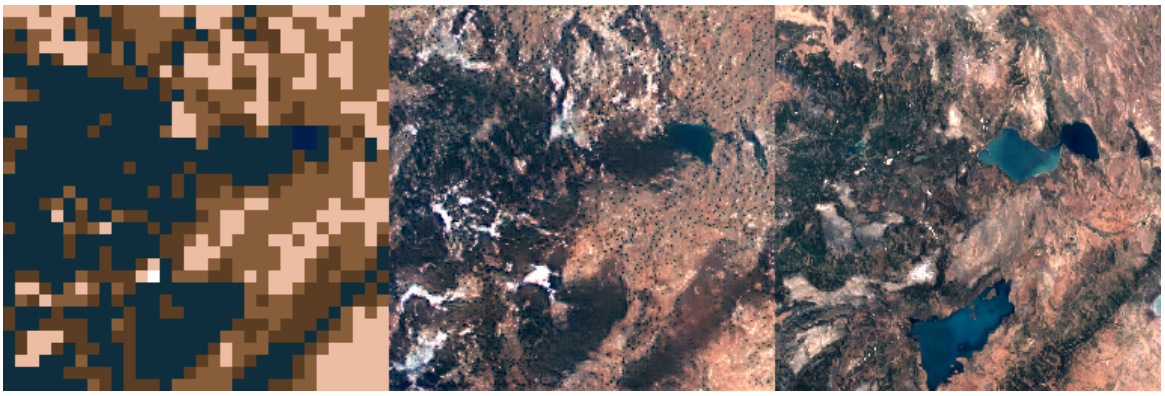


Figure 5.10: Training image from step 1393600 from the ResNet Experiment

Concluding the training process, the proposed hypothesis is somewhat confirmed, ResNet is indeed capable of obtaining a relatively close level of detail and complexity to U-Net, but to a lesser degree of realism. Intuition would indicate that training the model for longer or increasing the number of residual blocks would push the performance of the model to something comparable or even more qualitative than U-Net.

The loss graphs (5.11) present a different story than the images from training. They depict a training process in which the Discriminator quickly learns to distinguish between the fake and real images, not giving any time for the Generator to catch up and continue the min-max game. Moreover, the GAN loss is increasing with no sign of plateauing. While the L1 loss on the Generator seems to be decreasing faster than in the case of the U-Net Generator, it does not directly correlate to the performance of the Generator.

Overall, while ResNet seems like a worthy candidate for a Generator model, the 9-block ResNet was not able to come up with results comparable to those of U-Net. While further experimentation with deeper networks and longer training times could possibly lead to satisfying results, the computational overhead ResNet presents in training is more than enough to dismiss it as a candidate in favor of U-Net, which is easier and more affordable to train, from a computational point of view.

In order to fully disregard this model as a candidate for the Generator model, the model was visually evaluated on pixel art samples generated with procedural noise. ResNet does not have the

ability to generalize on the whole input domain, evident by the smudged textures and heavy grid and white dot clusters artifacts (5.19).

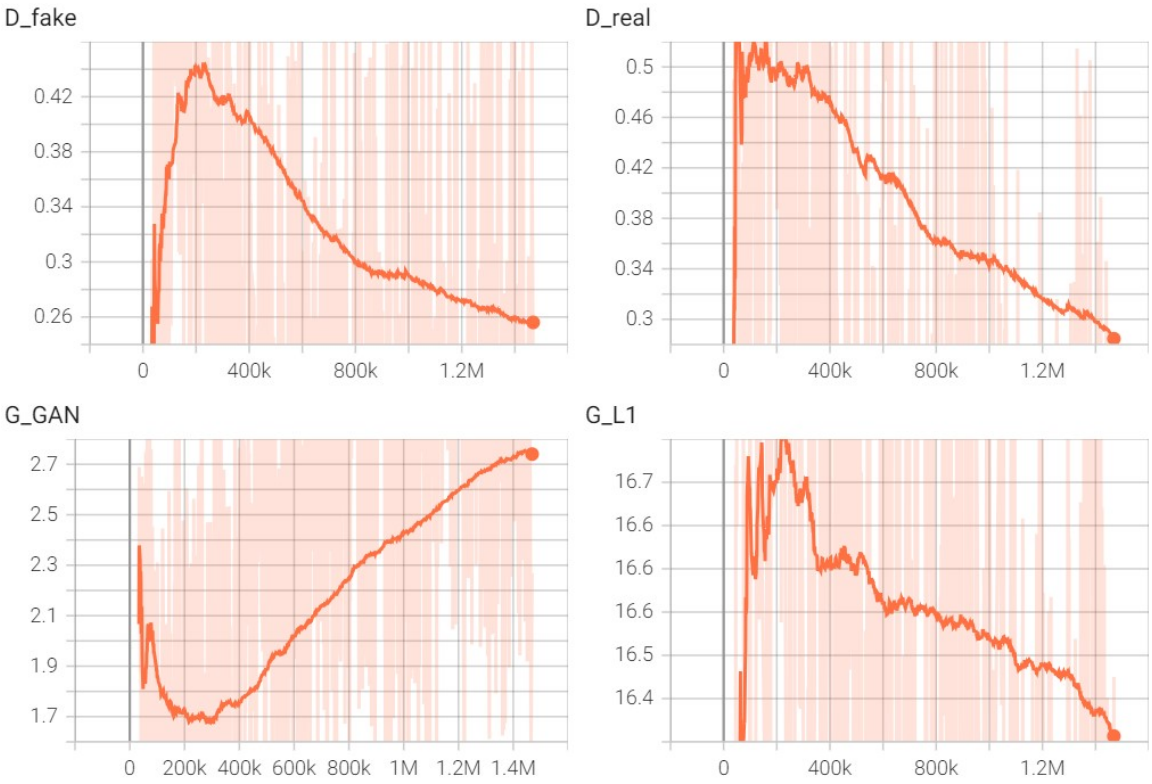
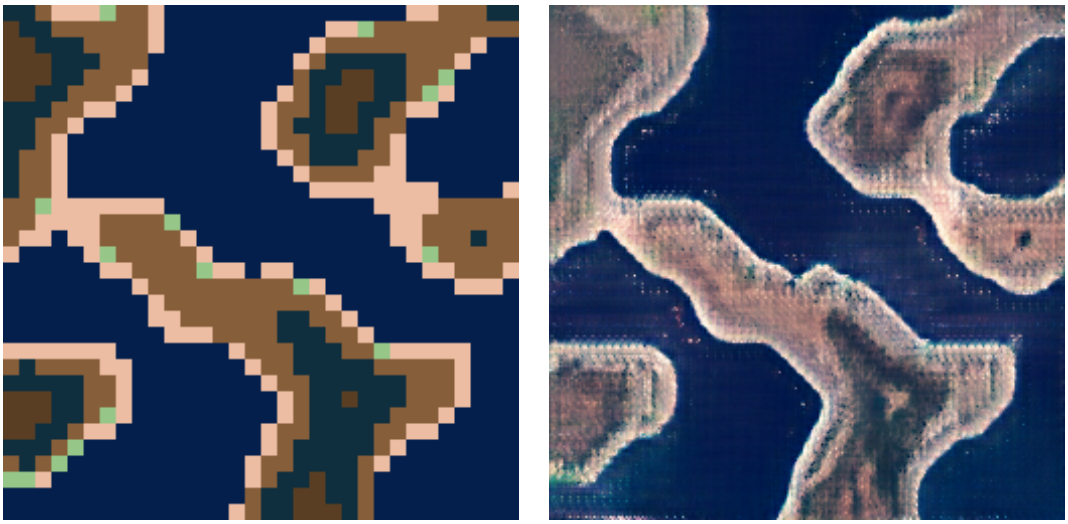


Figure 5.11: Loss graphs for the pix2sat ResNet experiment



Input Pixel Art Output Realistic Image

Figure 5.12: Example of an image generated from procedurally generated pixel art using the pix2sat ResNet model

5.4 34x34 PatchGAN Experiment

The initial intuition when interacting with GANs is that the most important component is the Generator since it is the only component used after training, the Discriminator being discarded in the majority of cases. The truth is, both models are equally important for the performance of the final Generator. The Discriminator is basically the only reason the Generator has any considerable incentive to produce qualitative images, therefore the focus should not be on one of the two, both model architectures should be thought about as one unit, interacting through a process that resembles a classic min-max game.

Given the shared importance of both the adversarial models and the fact that an experiment regarding the Generator has been completed, an experiment arises from the question of how the structure of the Discriminator impacts the results of the Generator. In the original pix2pix architecture, the used Discriminator is PatchGAN (5.13). Compared with other classifier models, which provide a single probability value for the whole input image, PatchGAN effectively computes multiple probabilities, one for each patch of the input, and computes an average that represents the final probability for the whole image. Recalling the input size of the Generator, $(256, 256)$, and considering the default PatchGAN patch size of 70×70, it seems logical that this value was chosen for the sake of generalization. The size is right in the middle of the patch sizes allowed by the structure of the PatchGAN. These sizes are 16×16, 34×34, 70×70, and 142×142. These are bounded by the PixelGAN, with 1×1 patch size, and ImageGAN takes the whole size of the image as the designated patch size.

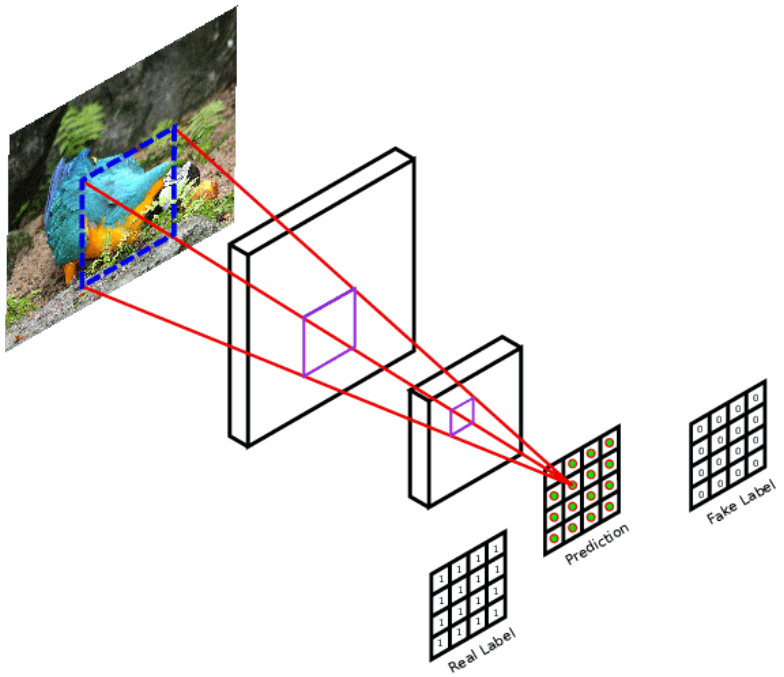


Figure 5.13: PatchGAN Discriminator (adapted from [7])

This experiment is based on the hypothesis that if the Discriminator would be fitted with a smaller patch size, 34×34, the output images would be more detailed, since the model would effectively 'look closer' at the images it's supposed to classify as real or fake, therefore forcing the Generator to come up with even more detailed results. The 16×16 was not chosen for this experiment due to it leading to tilling artifacts, as concluded by experiments from the original paper that introduced the PatchGAN [16].

Figure 5.14: Training image from step 404800 from the 34×34 PatchGAN Experiment

Compared with the results from early training of the 70×70 PatchGAN version of the architecture, the initial training images show much more complex and defined structures (5.14). It is one of the few encountered cases in training where the fake image seems to be more realistic than the actual image. With results as qualitative as this in early training, it is a good first indicator that the proposed hypothesis could be validated.

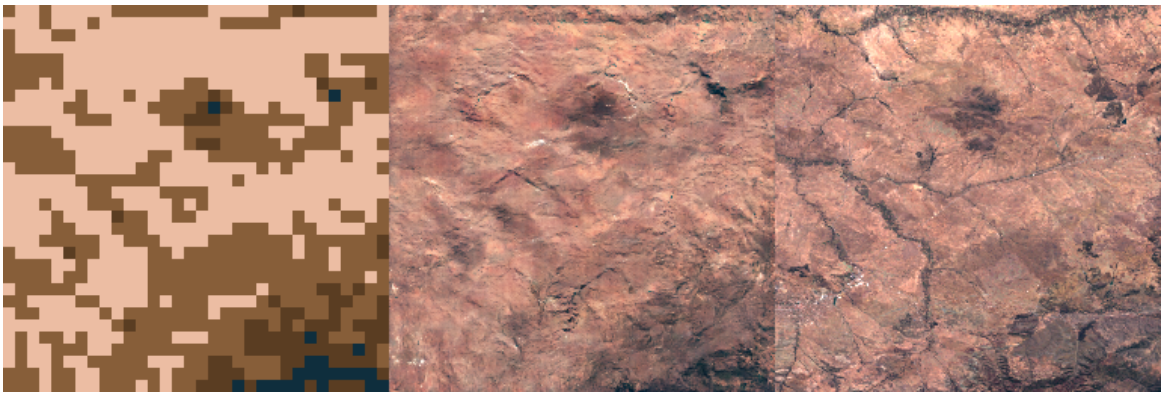


Figure 5.15: Training image from step 1085600 from the 34×34 PatchGAN Experiment

The visual quality and realism of the results increased somewhat linearly, up until the one million step mark when it reached a point that far surpasses the results obtained with the bigger patch

size PatchGAN. As can be observed in (5.15), (5.16), and (5.17), changing the size of the patch in the Discriminator architecture prompted the Generator to come up with highly detailed landscape patterns, with an incomparable level of realism.

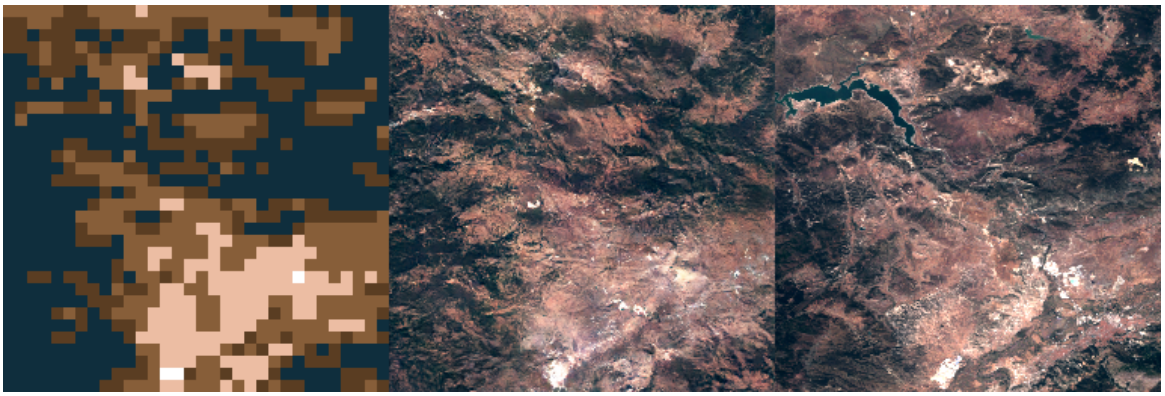


Figure 5.16: Training image from step 1349600 from the 34×34 PatchGAN Experiment

Figure 5.17: Training image from step 1393600 from the 34×34 PatchGAN Experiment

To gain a deeper insight into the training process, we turn to the loss graphs (5.18). The loss behavior is similar to that of the original patch size version of the model, therefore changing the structure of the Discriminator slightly did not have any reverberating effect on the stability of the training process. The GAN loss declines inverse exponentially, while the L1 Generator loss again shows a linear descent. Plateauing can be observed in the Discriminator loss graphs, indicating that the Generator was given enough time to come up with results that could fool the Discriminator. Overall, the training process from the point of view of the loss values has not changed in any significant way.

While there is no doubt that the model is able to generalize on the procedurally generated sample set, it is worth examining the difference between both patch size versions of the model in order to fully grasp the difference in quality between them. The textures in the proposed patch size version seem to

be sharper. Moreover, it is evident that there is a clear difference in the way both methods respect the semantic structure of the input. Using a smaller patch might also allow for a more detailed structure translation, while the bigger patch might cause the Generator to skip over these kinds of details. The problem of land slivers seems to be present in all experiments so far. Despite the fact that the smaller patch size facilitates a completely different level of structural and visual complexity, it does seem to cause a larger number of these kinds of artifacts to appear in the ocean texture of the output.

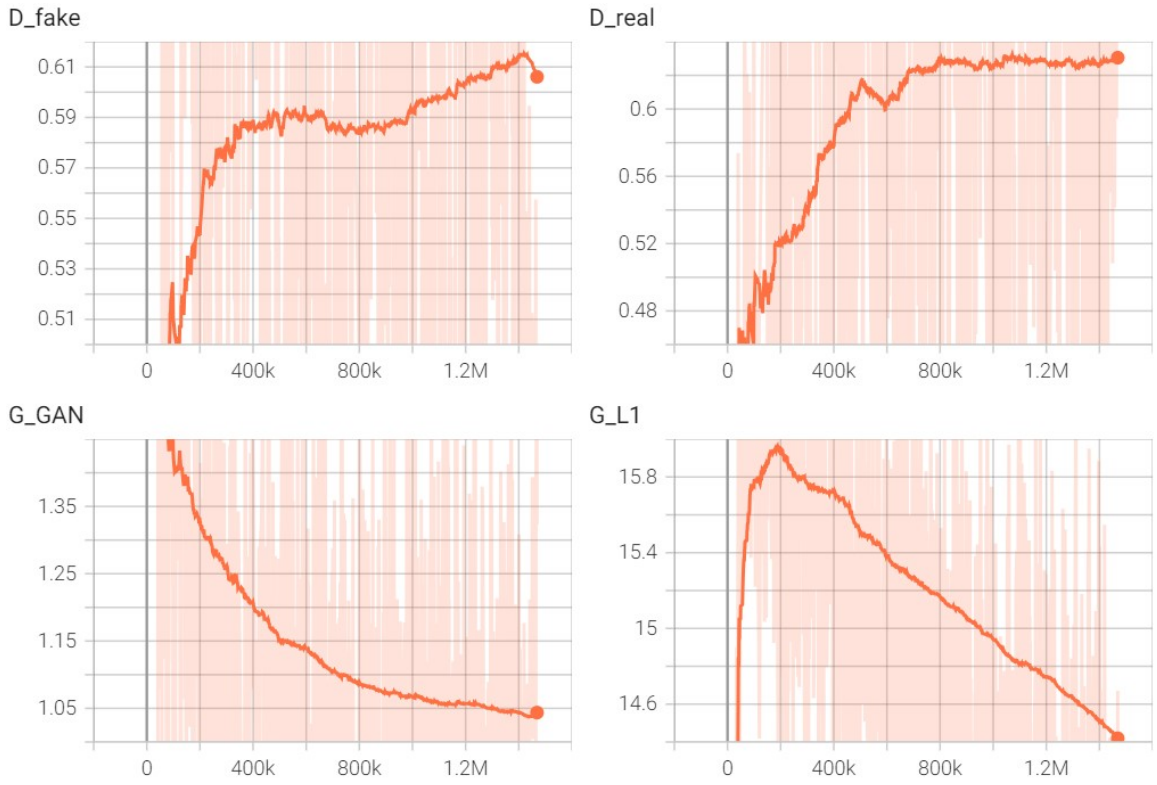


Figure 5.18: Loss graphs for the pix2sat 34×34 PatchGAN experiment

Input Pixel Art | 34×34 PatchGAN | 70×70 PatchGAN

Figure 5.19: Comparison on procedurally generated input of different size PatchGANs

Concluding the experiment, it is clear that using a smaller, 34×34 patch size for the PatchGAN Discriminator is able to facilitate the Generator to produce more detailed, natural, and complex landscapes than before. Coupled with a method to remove the aforementioned artifacts, it would greatly improve the quality of the proposed pix2sat architecture.

5.5 Total Variation Loss Experiment

Thus far, the only prominent visual feature that was consistent throughout all experiments and can be considered as a mild flaw was the land sliver artifacts present along coastal areas. While their presence does not take away from the visual aesthetic quality of the images, since their shape and color resemble small island formations which are terrain formations that occur frequently in nature, reducing the frequency of these artifacts would result in more polished images.

Essentially, total variation loss measures the amount of "roughness" or "variation" in a given image. Minimizing this loss would effectively lead to smoother and less noisy images. Since these slivers can be classified as noise in the image, enforcing this loss on the Generator could lead to reducing their occurrence.

As stated before, this loss would need to be applied to the Generator model directly, with a given weight λ_{TV}. In order to determine the most fit values for this weight parameter, a series of experiments were conducted with different values.

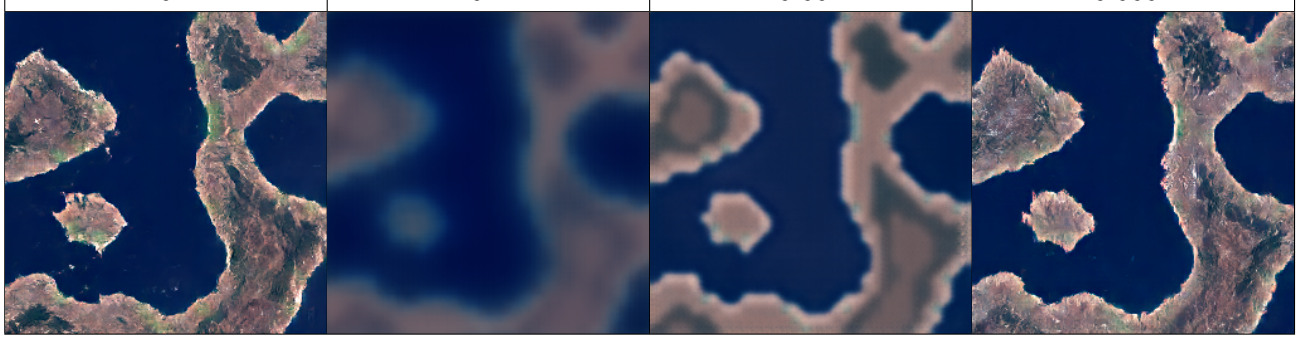
λ_{TV}			
0	0.1	0.001	0.0001
			

Table 5.1: Visual comparison of generated images for different values of λ_{TV}

When the value of the λ_{TV} parameter is 0, meaning the default model without the loss, there are visible artifacts in the image, basically, the problem described above. The initial test was conducted with $\lambda_{TV} = 0.1$. During training, the addition of this loss seemed to destabilize the process. After only a few epochs, the Discriminator losses were quickly decreasing towards 0 (5.20), meaning that the Discriminator quickly figured out which images were fake and which were real. This was most likely

caused by the initial value being too large, which in turn caused the Generator to output blurry, overly polished images. Due to the behavior of the previously mentioned losses, training was stopped after 5 epochs.

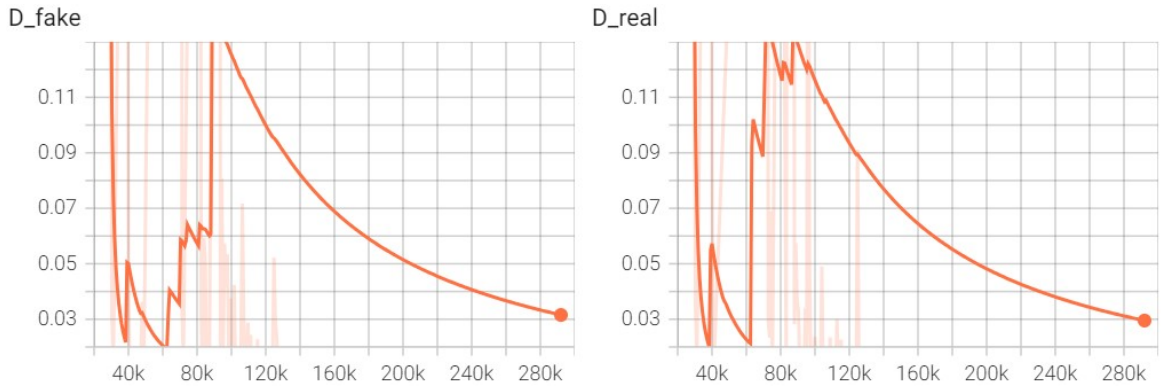


Figure 5.20: Discriminator Losses for $\lambda_{TV} = 0.1$

The heavy blur generated by $\lambda_{TV} = 0.1$ determined the next test to consider a considerably smaller value, $\lambda_{TV} = 0.001$. Visually, the blurring effect was diminished and the textures became more clear. Even though there was some improvement, the quality of the images was inferior when compared with the results of the base model. During training, the Discriminator losses behaved exactly in the exact same manner as in the previous test. The slight improvement in detail, although small, indicated that a suitable value for λ_{TV} was in close vicinity.

The next value that was tested, $\lambda_{TV} = 0.0001$, successfully eliminated the artifacts from the input. This was proof that the Total Variation loss, when applied with a fitting weight on the Generator, could be a viable solution for removing the sliver artifacts to a satisfactory degree. When comparing the output of the model with $\lambda_{TV} = 0.001$ and $\lambda_{TV} = 0$, besides the fact that the artifacts are not present in the former version, there is a slight blurring effect on the other textures. With further experimentation with hyperparameter tuning methods, a value for λ_{TV} that is in the close vicinity of 0.0001 can be obtained that would balance the removal of the artifacts and the blurring effect on the outputs.

A final test was conducted with $\lambda_{TV} = 0.00001$ in order to set a crude lower bound for the aforementioned vicinity. As expected, such a small value for λ_{TV} does not have a significant impact on the Generator, resulting in the reappearance of the artifacts.

It should be noted that while this method is considerably effective in removing a large number of artifacts of this type from the input, and even if the most fitting value for λ_{TV} is found, there might be few cases where the artifacts might be present, although smaller and less frequently than before.

5.6 Numerical Validation

Following the series of experiments performed to determine possible improvements in the pix2pix architecture, the proposed modifications are compared by using numerical evaluation metrics. Using these types of metrics offers deeper insight into the performance of each derived architecture. The metrics are used to evaluate different aspects and characteristics that would not be evident by performing only visual analysis. The specific metrics used are FID, LPIPS, and SSIM.

Fréchet Inception Distance [15], commonly referred to as FID is a metric commonly used to assess the output images of generative models, more specifically in the current context of Generative Adversarial Nets. The numerical value of the FID score essentially represents the similarity between two given sets of images, namely real images and fake images. Instead of performing a direct comparison between the two, for each image the distribution of features is extracted and compared. An Inception model, which is a model designed for image classification tasks, is used to obtain these distributions. This metric can be used to understand the degree to which the fake and real images share common abstract features. A lower FID score would indicate an increased similarity index between the images.

Learned Perceptual Image Patch Similarity [46], LPIPS for short, is a metric developed by a team of researchers from UC Berkeley with the help of researchers at Adobe to measure the perceptual difference between two given images, based on a certain understanding of how human perception works. Similar to FID, LPIPS uses an image classifier such as AlexNet or VGG to obtain feature layers which are used in a linear combination to compute the perceptual distance between them the images. Given the fact that the score is considered as a distance, a positive value, the lower the score the more similar the images are. Having another method based on a pre-trained model leads to a better understanding of the perceptual similarity between the generated images and the real ones.

Structural Similarity Index Measure [44], or SSIM, is considered a sophisticated metric that measures the similarity between two given images by comparing features such as structure, luminance, and contrast. The score is calculated by considering patches, commonly referred to as windows in the context of SSIM, of an image. The window of one image is compared with the corresponding image from the other. The final index is computed using a multiplicative combination of the 3 previously mentioned features, obtaining a value that is bounded between -1 and 1. The closer the value is to -1, the more perceptual difference there are between the two images, and naturally a value closer to 1 would indicate a high degree of similarity. Since the previously described metrics employ the use of pre-trained models to compute the score, this metric was chosen for the fact that it uses concrete features in computing the score.

Experiment	FID Score	LPIPS Score	SSIM Score
pix2pix	45.80	**0.239**	0.3989
pix2sat ResNet	111.60	0.369	0.3784
pix2sat 34×34 PatchGAN	**42.71**	0.245	0.3966
pix2sat Total Variation Loss ($\lambda_{TV} = 0.0001$)	46.94	0.241	**0.3994**

Table 5.2: Metric values for each experimental model

All the score values are prominently displayed in Table 5.2. A closer look at the FID column confirms the observation made on the ResNet model, that the model is not able to capture the same level of detail and quality as U-Net when used as a Generator, therefore obtaining a relatively big FID score when compared with the rest of the experimental model architectures. Looking at the FID score related to the 34×34 PatchGAN experiment, choosing a smaller patch size does indeed lead to the Generator outputting images that are more perceptually closer to the real-life samples than its 70×70 patch size counterpart.

While the LPIPS scores seem to be relatively close to each other, the ResNet outlier helps decide if the score differences translate to significant differences in the quality and behavior of the models. Given that the differences between the base pix2pix experiment score and the rest, excluding ResNet, are of magnitude 0.001, and the difference between the ResNet score and the rest is of magnitude 0.1, we can conclude that aside from the terrible performance of the ResNet model as a Generator, the LPIPS scores don't necessarily offer any extra useful information.

The experiment that seems to have the best SSIM score is the Total Variation Loss. This is also in accordance with the visual analysis which yielded the fact that the use of the loss has a great reduction effect on the number of land sliver artifacts. The removal of these artifacts would make the structure of the final image slightly similar to the structure of the real image. This slight increase in similarity is indeed reflected in the SSIM score.

A final observation to be made about the metrics table refers again to the Total Variation loss experiment, compared with the initial experiment. During the experimentation phase, the Total Variation loss was classified as a viable solution to removing the artifacts, this being confirmed by the SSIM score, as described above. Aside from this, another observation was made, that the generated images are slightly less complex and detailed than the ones produced by the base model. This remark is accurately reflected by the small difference between the corresponding FID and LPIPS scores, hence these metrics could be used as calibration factors in the process of fine-tuning the value of λ_{Tv}.

Chapter 6

Software Application

In this chapter, the design and development process of a simple and intuitive desktop software application called Pix2Sat is detailed. Written in C++ using the Qt GUI framework, this application enables users to create pixel art representations of overhead terrain and transform them into realistic satellite-like images. By providing a user-friendly interface, the application allows for a wide range of control, from generating initial pixelated terrain using customizable procedural noise functions, through refining the output with personal modifications using the Paint-like interface, to choosing from a range of pre-trained AI models for the final transformation.

6.1 Abstract Design

Given the wide range of actions the user should be able to do, there are a lot of use cases that should be considered (6.1). The most important use cases are related to the user being able to draw pixel art on a canvas since the main point of control of the final output is in this stage. Considering this, "Draw On Canvas" is a fundamental use case, representing the ultimate modality of control from the user's perspective. This use case would not be complete without "Choose Brush Color", adding variety to the user's modifications by providing a set of colors that would later be translated to terrain textures. While these use cases represent the building blocks of the application, additional use cases such as "Undo Drawing Action" and its complement "Redo Drawing Action" contribute to the idea of making the application as user-friendly as possible by allowing a higher degree of control over the canvas.

Considering users who may not possess extensive artistic abilities or inclinations, the "Generate Procedural Landscape" use case might be the most accessible alternative. Coupled with the "Modify Procedural Function Parameters", these use cases provide another direction for the user to take in the pixel art creation process, building on the accessibility of the application.

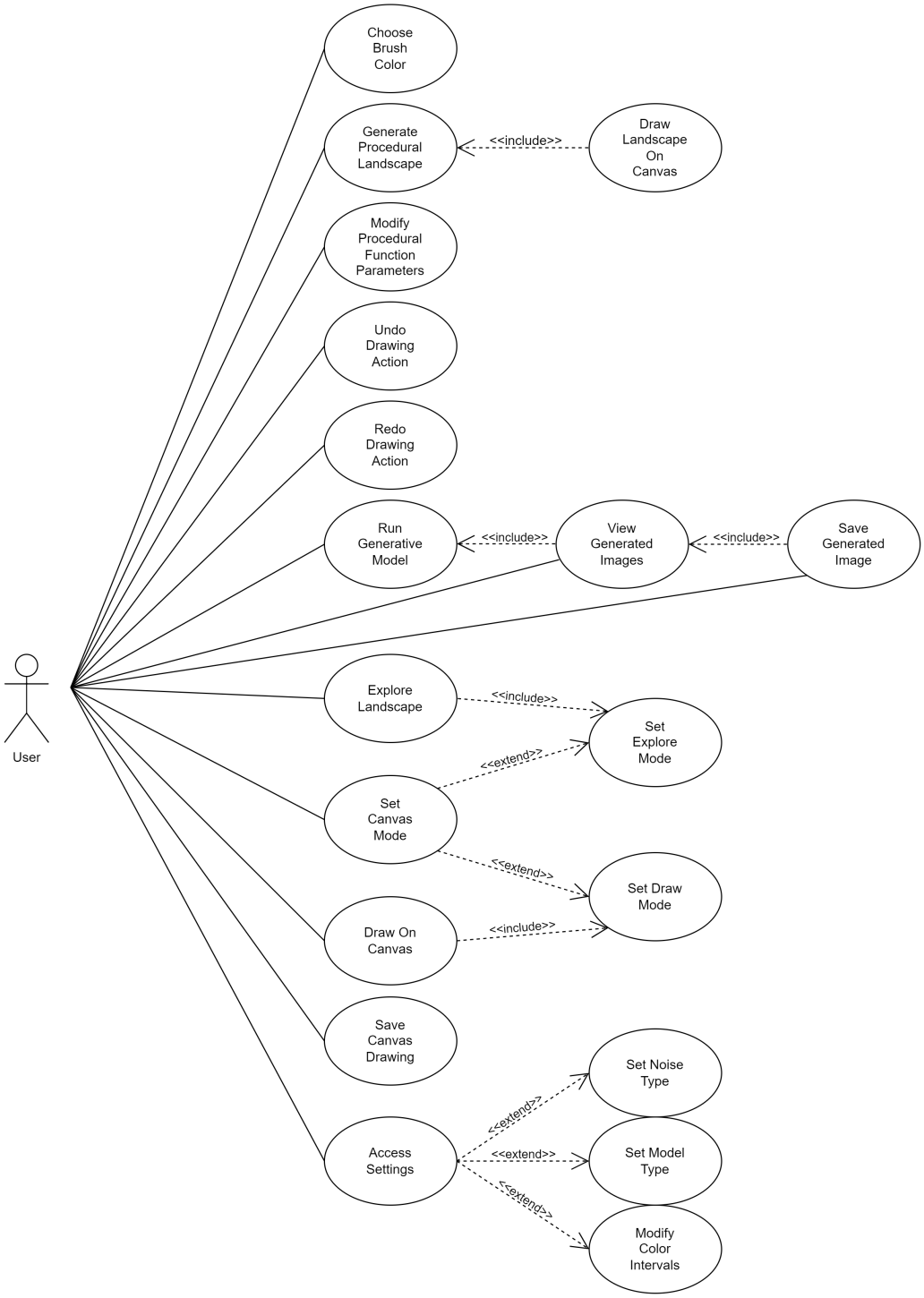


Figure 6.1: Use Case Diagram for Pix2Sat

Given the ability of procedural noise functions to produce infinitely expanding maps, a use case worth mentioning when discussing user control is "Explore Landscape". Having generated a landscape, the user can explore without bounds, giving a variety of scenes they can capture and use. Switching between these methods of building the pixelated world is described by the "Set Canvas Mode" use case, which is extended by the "Set Draw Mode" and "Set Explore Mode" use cases, providing a smooth transition from one to the other from the user's point of view.

The final step in the process is described by the "Run Generative Model" use case. Similar to "Draw On Canvas", it is one of the fundamental cases, the most important functionality of the application. Together with the "View Generated Images" and "Save Generated Images" use cases, it defines a smooth functionality flow, representing the last piece in the overall flow of the application use process. Additionally, the use case "Access Settings" is extended with "Set Noise Type", "Set Model Type" and "Modify Color Intervals", contributing to the user's ability to fine-tune the outcome of the overall flow.

While all use cases provide valuable insight into the functionalities of the application, some of them warrant a more in-depth analysis due to the intricate communication flow between components. To further understand these use cases, sequence diagrams help illustrate the chronological interactions between components.

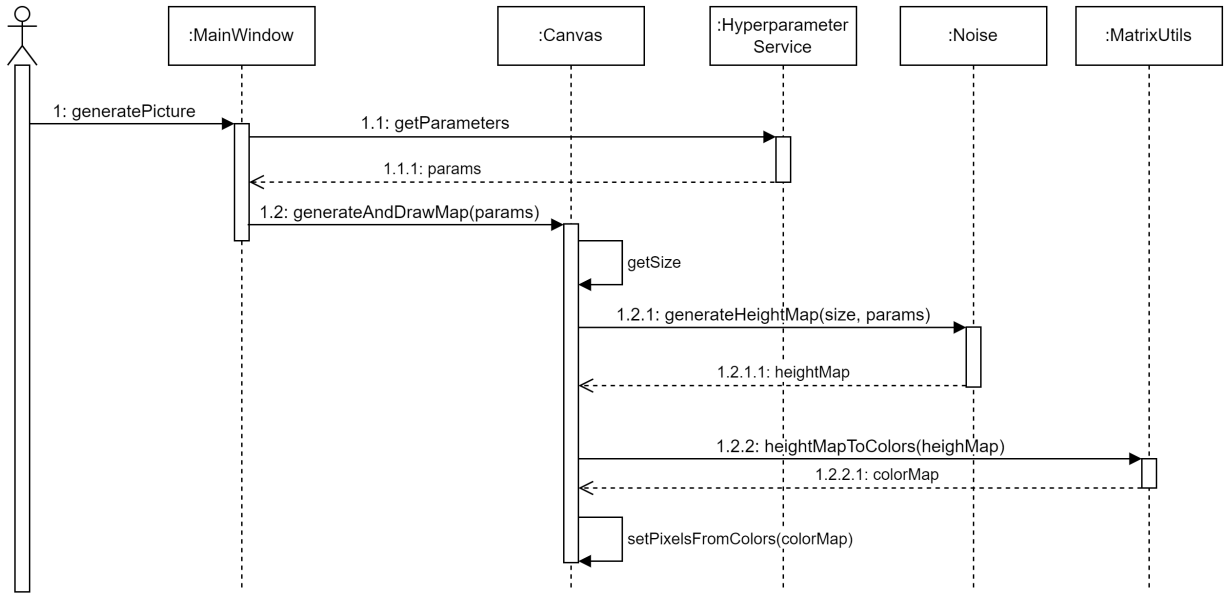


Figure 6.2: Sequence Diagram for the Generate Procedural Landscape Use Case

One such use case with a more intricate interaction pattern between components is "Generate Procedural Landscape". This use case encapsulated the application's ability to produce procedurally generated pixelated terrain. Analyzing the sequence diagram (6.2), the process is initiated by the

user, which sends a message `generatePicture` to the MainWindow component. Triggered by this message, it fetches the parameters needed for the procedural noise function down the line from the HyperparamService component by sending the `getParameters` message. MainWindow messages the Canvas component, also providing the necessary parameters, leaving the Canvas to handle the rest of the interactions. When beginning its lifeline, the Canvas component messages itself in order to obtain the `size` value from its own state, and passes the `size` and the received `params` from MainWindow together with the `generateHeightMap` message to the Noise Component, which applies the procedural noise function and returns the `heightMap`. In order to convert this returned map to something the Canvas component can work with, it messages the MatrixUtils component, `heightMapToColors`, in order to transform the `heightMap` into `colorMap`. Finally, the Canvas sends the `setPixelsFromColors` message along with the `colorMap` to itself in order to update its internal state and display the generated image.

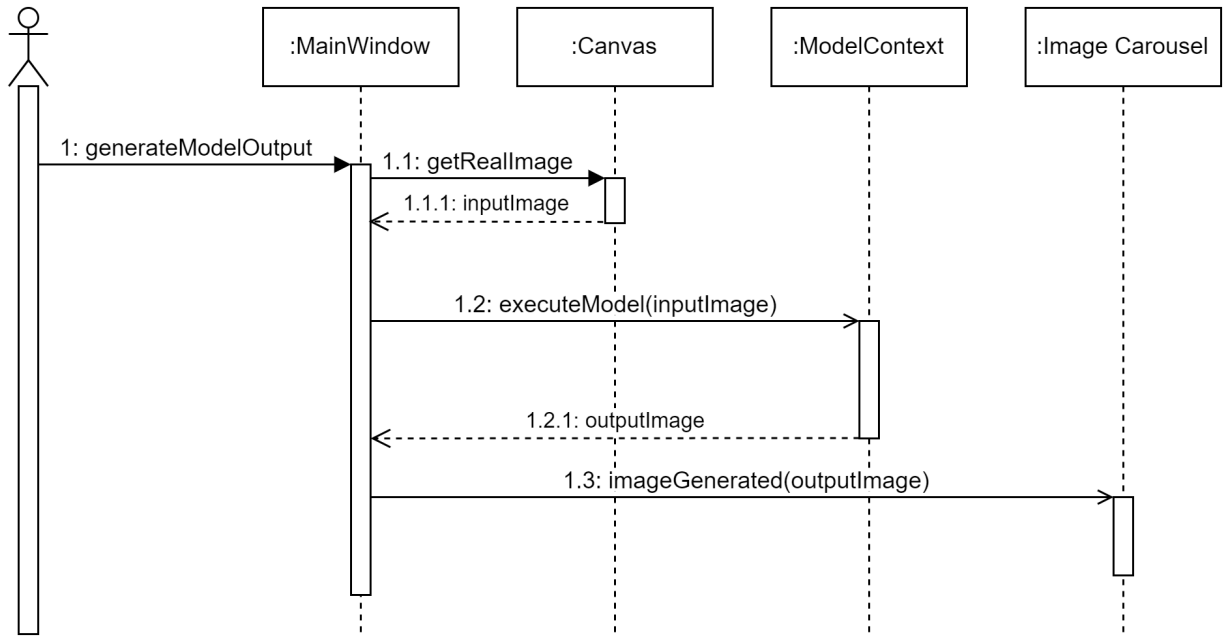


Figure 6.3: Sequence Diagram for the Run Generative Model Use Case

Another pivotal use case worth delving into is "Run Generative Model" (6.3), as it outlines the user's interactions that extend to the very core of the application, the model. As before, the initial message, `generateModelOutput`, is sent from the user to the MainWindow component. In order to retrieve the `inputImage` needed for the model down the line, MainWindow sends `getRealImage` to the Canvas. From this point forward, all the performed operations are asynchronous, so as to not block the MainWindow, which would cause a visible blockage of the app for the user, since AI models are known to have longer inference times. Running async, the `executeModel(inputImage)` message is sent to the

ModelContext component. The returned `outputImage` message is then rerouted by MainWindow to the ImageCarousel component in order for it to be displayed to the user.

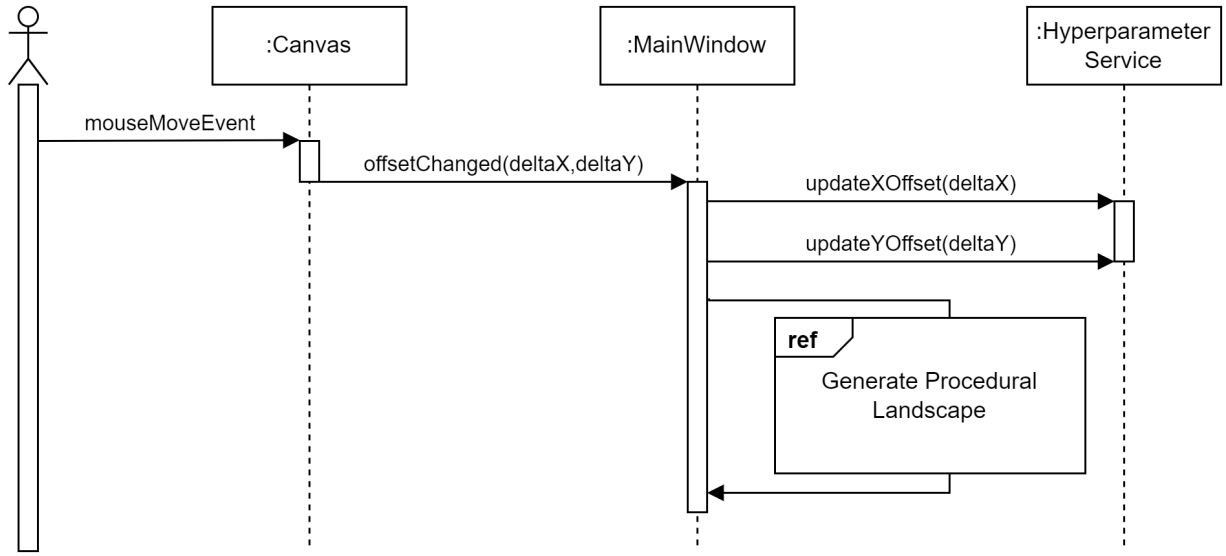


Figure 6.4: Sequence Diagram for the Explore Landscape Use Case

Although the sequence diagram for the "Explore Landscape" (6.4) is not as complex as the rest, it's worth going over due to its significance from a user experience point of view. The user sends the initial message to the Canvas in the form of a mouse move over it. This triggers the Canvas to calculate the offsets from the last move and to send them to the MainWindow. These offsets are sent to the HyperparamService component which stores them. Those values are immediately reused when the MainWindow component triggers the sequence described in the sequence diagram for the "Generate Procedural Landscape", thus the landscape is again generated, but taking into account the new offsets. This sequence describes a process that gives the user the impression that they are exploring the generated landscape.

Having gone through the sequence diagrams of a few specific use cases, a detailed view of the flow of operations and interactions was presented. While sequence diagrams are helpful for understanding the behavior of a system from a time-oriented point of view, it is also important to understand the modular structure of the system. This leads to an examination of the component diagram (6.5). The MainWindow component is the absolute core of the application, everything is directly or indirectly linked to it. It's responsible for making all the connections between the different types of components, while also containing the other Widget components in its layout and managing their state. Looking at the components from a layered point of view, we first have the components related to the GUI, the Widget components. The BrushWidget component is one of the simplest visual components since it only displays the current brush color to the user. Linked to this component is the ColorPickerWidget,

which can be used by the user to change the current brush color. Going towards more complex Widget components, we have ToolsWidget. This component allows the user to set the Canvas Mode (Explore or Draw, as stated before) and to undo or redo Canvas actions. Another component that offers control to the user is the NoiseControlsWidget, which deals with the different values the user might set for the procedural noise functions. The Image Carousel is the final destination for the model-generated images. It displays the images in the order they were generated. Moving on to the SettingsDialog, this component is used independently of the MainWindow, even if it is launched by it. This component contains simple options for the user, but one of the settings has a much more complex nature, and cannot be represented using conventional input methods, therefore an entire Widget component, ColorIntervalsWidget is used inside the SettingsDialog component. The most important component of the visual layer is the Canvas. The Canvas is comprised of a surface that the user can use to draw pixel art, with a behavior similar to any generic painting application. Going a layer deeper, we look at the logic/business layer. The Settings and ColorPalette components are used throughout the application, serving as data-centric components. The HyperparamService also presents this kind of behavior, being in charge of storing the parameters used for the procedural noise functions in the pixelated landscape generation process. The component that deals with running the model on the input image is the ModelContext.

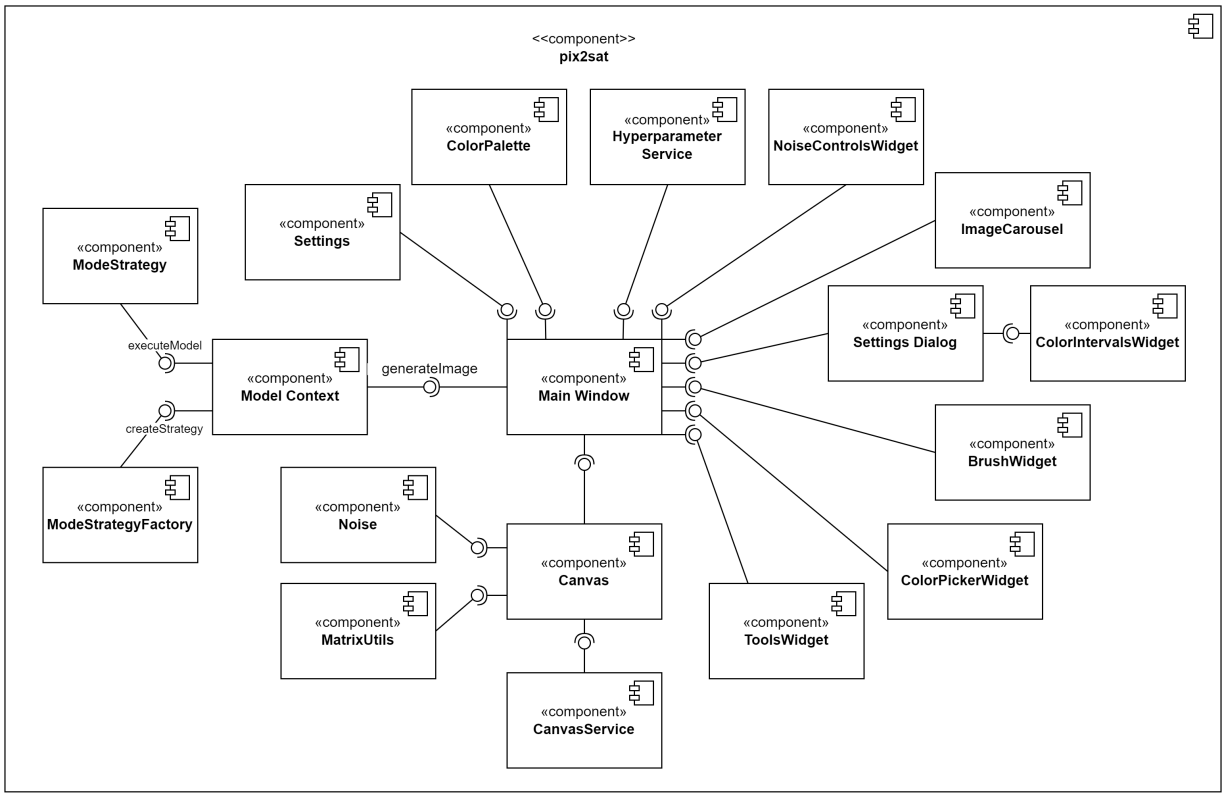


Figure 6.5: Component Diagram for Pix2Sat

Up until now, all components mentioned were directly providing interfaces to the core component, MainWindow. The ModelContext component requires interfaces from the ModelStrategy component, which represents the different models that can be applied, and the ModelStrategyFactory component, which is used by the ModelContext when a different strategy(or model from the user's point of view) is selected. Being a more complex visual component, the Canvas also requires interfaces for some logical components. The CanvasManager component handles the undo/redo logic, while the Noise and the MatrixUtils components are used by the Canvas component in the pixelated landscape generation process.

Reflecting on the analysis of the diagrams, an abstract understanding of the application's inner workings was laid out. Each component and the interactions between them play a distinct yet interconnected role, collectively contributing to achieving the greater goal of the application. Having presented this skeleton of the system, we will focus next on the actual implementation of the Pix2Sat application.

6.2 Implementation

The initiation of the 'Pix2Sat' software application's implementation phase presents a critical decision-the selection of an appropriate technology stack. This decision sets the foundation for following development activities, impacting the efficiency, maintainability, and performance of the final product. Therefore, this decision was made given the requirements of the application.

The technology stack for this application is a combination of Qt with C++ for the desktop interface, and Python for the local server that hosts the models. The Qt GUI framework was chosen for its extensive support for diverse graphical capabilities, facilitating the creation of a highly interactive and intuitive user experience. Moreover, besides UI design, it includes pre-built modules for tasks such as event handling, image processing, and 2D graphics rendering, as well as inter-process communication. This modular nature provides built-in scalability, meaning that developing in this environment not only addresses its current requirements but also allows for potential extension and enhancement of the application. Since it supports a range of platforms, it offers the flexibility to deploy applications across different environments. Being that the framework is available for a multitude of programming languages, it is also important to consider the upsides and downsides of each. C++ was selected for its exceptional performance characteristics, making it the ideal candidate for implementing an application that requires a high standard for the speed at which operations are completed. Combined, they provide a responsive and immersive user experience, delivering high-quality results in a timely manner.

The choice of Python for the local server was mainly driven by its extensive support within the AI community (with libraries like TensorFlow and Pytorch being among the most popular). Therefore integrating the models into the local server is seamless, allowing for a smooth transition from training the model all the way to deployment. Since both Python and C++ support the Qt framework, communication between the two components is supported by the inter-process communication module provided by Qt.

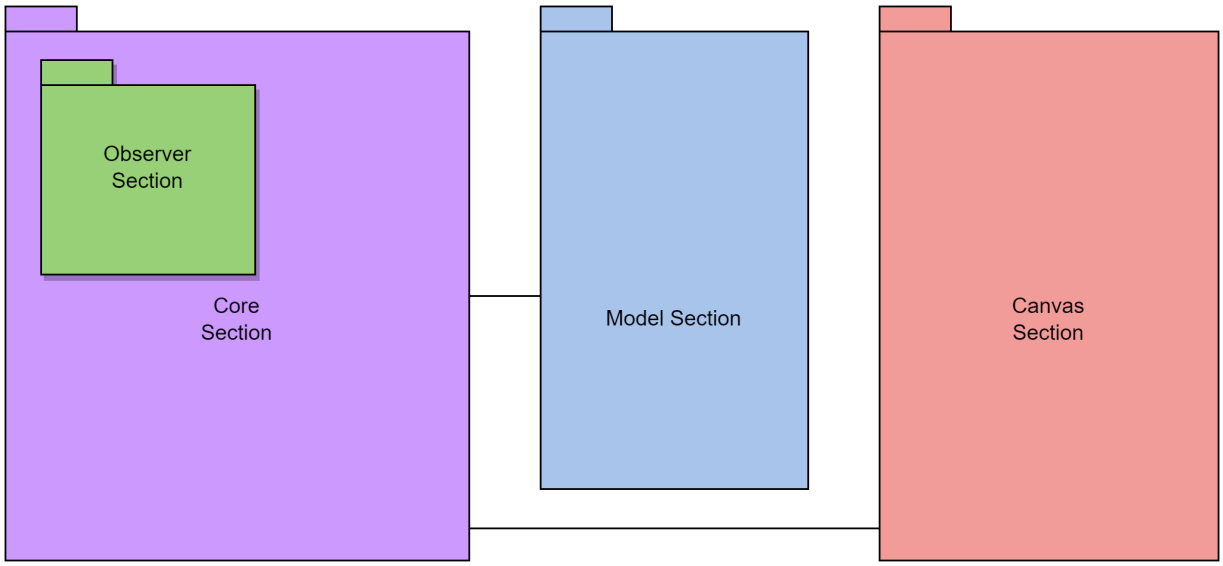


Figure 6.6: Class Diagram divided into 4 major sections

With the technology stack fixed, the focus now turns to the organization and structure of the system. To gain a more concrete understanding of its intricacies, the class diagram has been divided into four major sections (6.6), allowing for a detailed examination and analysis of the architecture.

The Canvas Section (6.7) encapsulates the connections between the different classes, forming a cohesive unit that enables the diverse functionalities of the drawing area. The central class of this section is, evidently, the Canvas class. The Canvas class is a subclass of the QWidget class, therefore it can implement methods that are called on mouse action events. The `mousePressEvent`, `mouseMoveEvent`, and the `mouseReleaseEvent` methods are used in combination with the `drawing` boolean attribute in order to simulate the drawing effect on the canvas. The `drawing` boolean is toggled by the `mousePressEvent` and `mouseReleseEvent` methods when the left button is clicked on the mouse. The `cellSize` attribute is used in the painting process since it dictates the size of the pixels on the canvas. When the `mouseMoveEvent` is called and the `drawing` attribute is true, a square with sides equal to `cellSize` is drawn using the color stored in the `brushColor` attribute in the corresponding patch in the drawing area. The actual drawing area is represented by the `pixmap` attribute, which basically is an object that

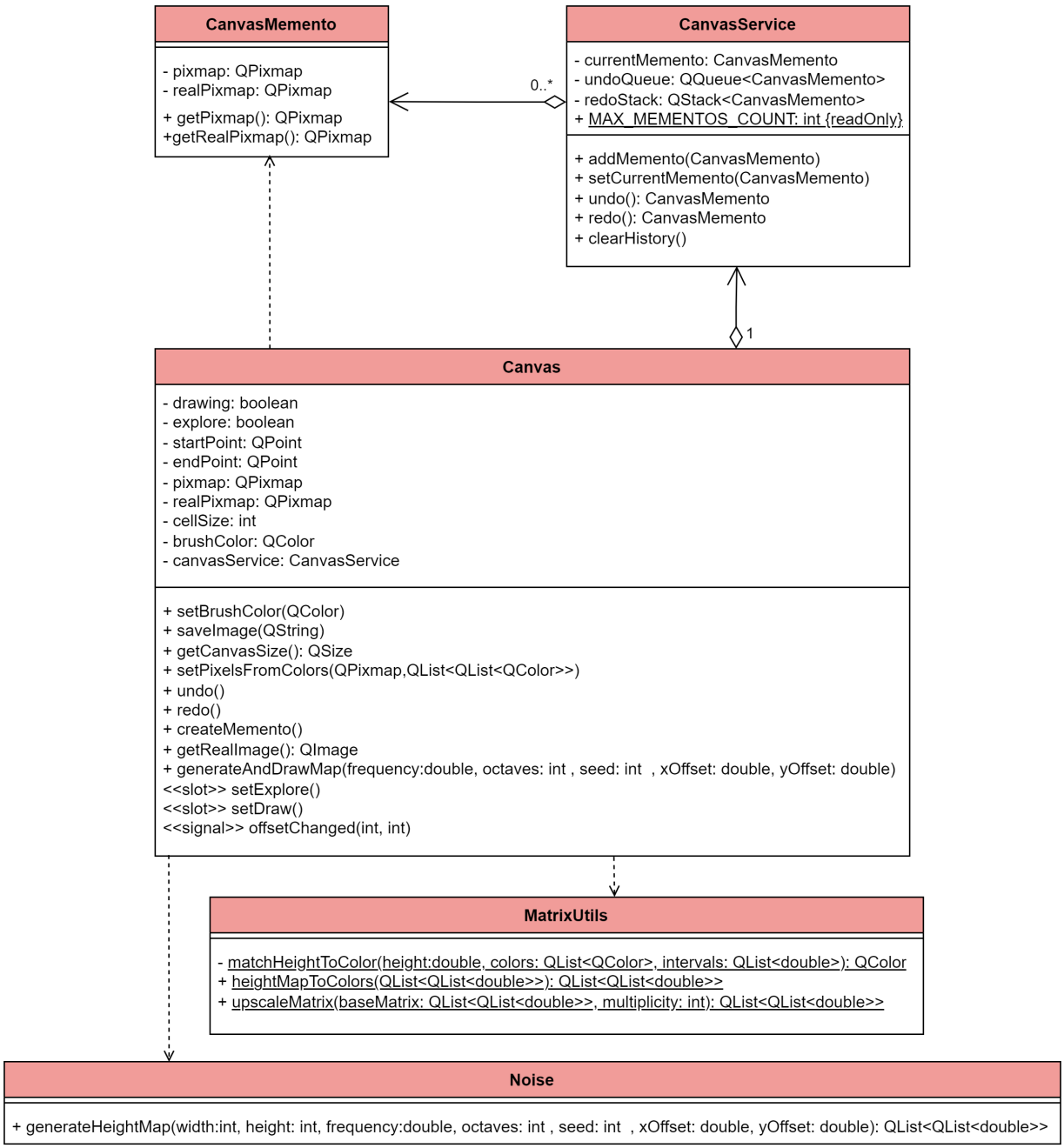


Figure 6.7: Canvas Section from the Class Diagram

can store an off-screen image. All the drawing operations are actually applied to this attribute, and the image is painted on each update in the Canvas layout. The AI models used to generate realistic images take images of size 256 by 256 as input, so if the drawing area would have the same size, it would be too small to draw on effectively. In order to provide a good user experience and meet the size requirement of the models, the `pixmap` is 512 by 512 in size with a `cellSize` of 16, big enough for the user to draw on, and a separate QPixmap is stored, `realPixmap`, with a reduced size to fit the model requirements, 256 by 256, and the cell size 8 in order to maintain the same ratio between the two QPixmaps. All operations done on the `pixmap` are also mirrored on the `realPixmap` in order to maintain consistency, and when generating the realistic image, `realPixmap` is transformed into an image and sent to the model. Other methods to bridge this gap between the models and the UI components sizes were attempted, such as encoding the image into a matrix of colors and decoding it in the local server, but all methods failed to preserve the quality of the image and generated visual artifacts due to lossy compression. The Canvas class deals with all the logic behind generating the pixelated procedural landscape. All the logic is encapsulated in the `generateAndDrawMap`, which requires the parameters for the procedural function, but also some coordinate offsets. The Canvas class uses the static Noise class, which in turn in its implementation uses functions from the FastNoiseLite [17] library, an extremely portable open-source noise generation library. After obtaining the matrix of doubles, also known as a height map, from the Nosie class, the MatrixUtils class is used to post-process the map so it can be displayed in the `pixmap`. First, the height map is upscaled, meaning that each element in the matrix is expanded to a `multiplicty x multiplicity` submatrix, each submatrix consisting of repeating the value of the corresponding element in the matrix. This is done in order to match each height value to a pixel in the `pixmap`. Moreover, the upscaled matrix of doubles is mapped to a matrix of colors. Since the height values are between 0 and 1, each color has a corresponding subinterval in this range, so it goes that each height value is mapped to the color corresponding to the intervals it falls in. It's worth mentioning that the intervals are stored in the ColorPalette singleton class, and can be modified to the users liking. Finally, the Canvas class uses its own static method `setPixelsFromColors` to set each pixel in the `pixmap` to its corresponding color from the color map. The `startPoint`, `endPoint`, and `explore` attributes are used in the explore landscape functionality, where the user can drag across the canvas and explore the outer bounds. The `explore` boolean is used to distinguish between the exploring mode and drawing mode of the Canvas. When the mouse is pressed, the point is saved in `startPoint`, and each time the mouse is moved, the point it moved to is saved in `endpoint`. These points are used to compute the offset by which the landscape should be shifted. The Canvas uses the slots and signals mechanism from Qt in order to signal the MainWindow that a drag operation was

done and by what amount. The MainWindow class passes the offsets to the HyperparamService which adds them to the overall offsets (which are initially 0). Then the MainWindow triggers a landscape generation action with the new offsets, prompting the Canvas to draw the landscape slightly shifted in the opposite direction of the user's drag. Since this event loop happens on each slight mouse move, the user is given the impression that the canvas is a sliding window that only shows a small portion of the infinitely large procedural landscape. This illusion is possible due to the aforementioned performance of the C++ language. Also in the Canvas Section, we can find the implementation for the undo/redo drawing actions mechanism. In order to implement it, the Memento Design Pattern was used. The Canvas acts as the Originator, CanvasMemento acts as the Memento, and CanvasService acts as the Caretaker. The CanvasMemento class is meant to be a snapshot of the Canvas, therefore it contains a `pixmap` and its corresponding `realPixmap`. CanvasService deals with the actual undo/redo logic. Normally, an undo/redo implementation uses two stacks and an intermediary value between them (in this case `currentMemento`), but in order to impose a limit on the number of saved states (set by the `MAX_MEMENTOS_COUNT` static constant), the undo stack is actually a queue, and when the maximum amount is reached, when adding a new CanvasMemento the oldest one is discarded, having a sliding window effect on the `undoQueue`. On each call of the `mousePressEvent`, the Canvas class calls the `createMemento` method, which copies both QPixmaps and packages them into a CanvasMemento, and sends it to the CanvasService. When the undo/redo method of the Canvas is called, it, in turn, calls the undo/redo method to retrieve the corresponding CanvasMemento, from which it updates its state.

The Core Section (6.8) contains the central Widget of the application, the MainWindow. This class is the main point of control for all the interactions in the system since all the signal and slot connections are established by it. Starting from an already familiar class, the Canvas class as stated before signals the `handleOffsetChange` slot from the MainWindow class. The offsets are then passed to the HyperparamService where the offsets are aggregated into the overall offsets. The HyperparamService is a class meant for managing the values needed for the procedural noise function and the aforementioned offset values. Similarly, the Settings class is a singleton class that holds the `noiseType` value which is an enum representing the type of procedural noise function the Noise class should use when generating height maps, and the `modelContext` which is used to gain access to the AI models. The Settings class is used in different parts of the application, so making it a singleton is beneficial. Another singleton class that is used in multiple places in the application is the ColorPalette class. It encapsulates the colors that the user can use to draw on the canvas along with all the information related to colors, what texture each color represents in `colorDescriptions` (e.g. the blue color represents `OCEAN`), and the color intervals (from `colorIntervals`) needed in the procedural landscape generation process from

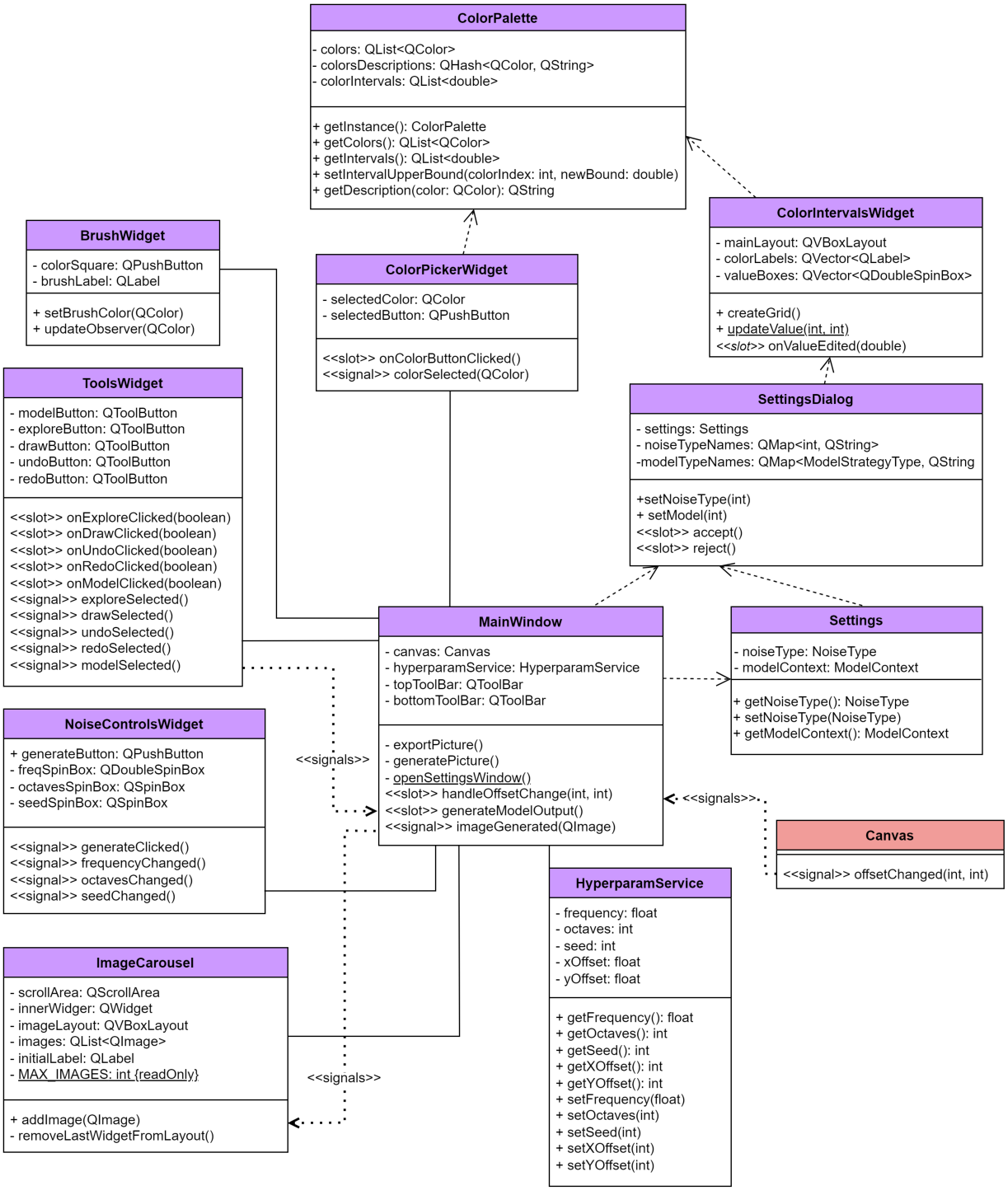


Figure 6.8: Core Section from the Class Diagram

the Canvas Section. Connected to these singleton classes and the MainWindow is the SettingsDialog, which is a widget that is launched by MainWindow and contains input components for setting the noise type and for setting the model type. Since the color intervals are a more complex form of information, common input widgets cannot be used, so a different custom widget was developed, ColorIntervalsWidget. What the intervals actually are is a simple list of increasing upper bounds between 0 and 1. The list is one element shorter than the list of colors since the upper bound for the last color is always 1. For each upper bound, there is a corresponding QDoubleSpinBox that is linked to its value. Like the ColorIntervalsWidget, the ColorPickerWidget also depends on the ColorPalette singleton. This widget is used to change the `brushColor` attribute from the Canvas class by sending the `colorSelected` signal to the `setBrushColor` method. The BrushWidget is a simple component that is used to display the current color of the brush from the Canvas. The ToolsWidget class acts as a bottleneck, providing an interface that exposes the Canvas functionalities to the user, all of them linked using the Qt slots and signals mechanism. Unlike these, the generative model functionality is linked directly to the MainWindow, sending the `modelSelected` signal to the `generateModelOutput` slot. From this point onward, the sequence of events is described in 6.3. The NoiseControlsWidget contains the input components for the `frequency`, `octaves`, and `seed` values from the HyperparamService. For this Widget, the signals for each variable change are mapped in the MainWindow class to lambda functions that receive the new value and update it in the HyperparamService. This is done in order to decouple the dependency on the HyperparamService of this Widget. Finally, we have the ImageCarousel class, a scrollable Widget that stores the model outputs. When the MainWindow asynchronously receives the generated output image from the local server, it emits the `imageGenerated` signal along with the output image to the `addImage` method in ImageCarousel. The image is transformed into a QLabel when displayed, and the actual QImage is stored in a QList inside the Widget. Similar to the CanvasService, the ImageCarousel can only hold a maximum amount of generated images (indicated by the `MAX_IMAGES` static constant). When the maximum amount is reached, for each image added to the list the oldest one is deleted by calling the `removeLastWidgetFromLayout` and removing the actual image from the `images` QList. For each image, the `clicked` signal is linked to a lambda function that opens a menu option to save the selected image.

The Model Section (6.9) encapsulates all the functionalities related to the AI models. A solution was required that would allow the dynamic selection of models at runtime, encapsulate each model's communication logic and ensure that the system could easily be extended with additional models in the future. This situation is exemplary of the Strategy Design Patter. These needs are a good fit for the Strategy Design Pattern, making it a suitable choice for this situation. In the case of

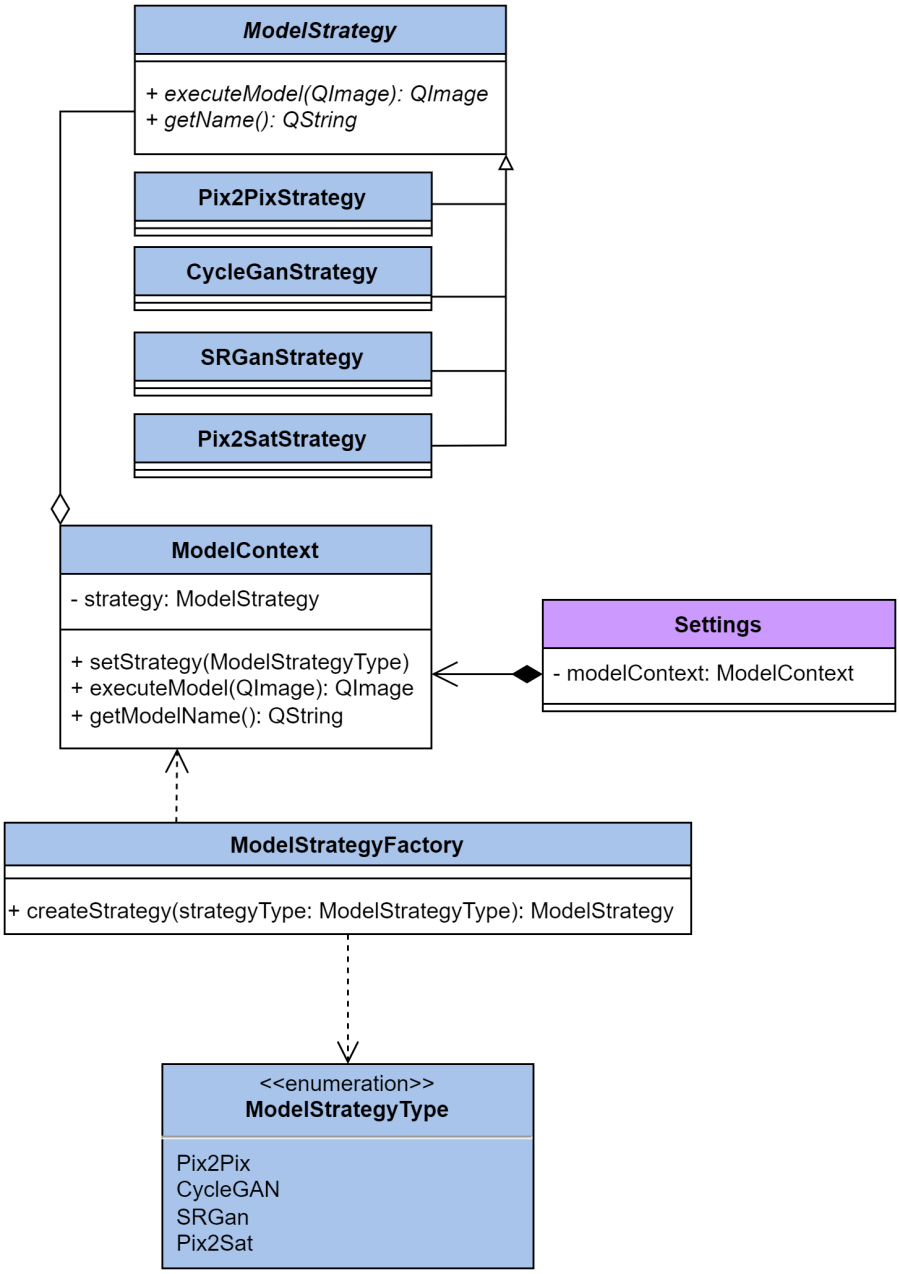


Figure 6.9: Model Section from the Class Diagram

this section, the ModelContext acts as the Context, the ModelStrategy abstract class acts as the Strategy, and the implementations for the different models are the ConcerteStrategies. As stated before, the ModelContext is stored in the Settings singleton class. When the strategy is changed from the SettingDialog, the `setStrategy` method from the ModelContext class is called. In order to adhere to the Single Responsibility Principle, the creation of the strategies was delegated to a separate class, the ModelStrategyFactory, therefore the Strategy Pattern was extended with a simple implementation of the Factory Pattern. The static method `createStrategy` of the Factory class is used by the `setStrategy`, employing the use of the `ModelStrategyType` enumeration in order to create the corresponding strategy. Each Strategy only implements the `executeModel` method of the abstract class, but also the `getName` method, used for displaying the models' name in the SettingsDialog Widget.

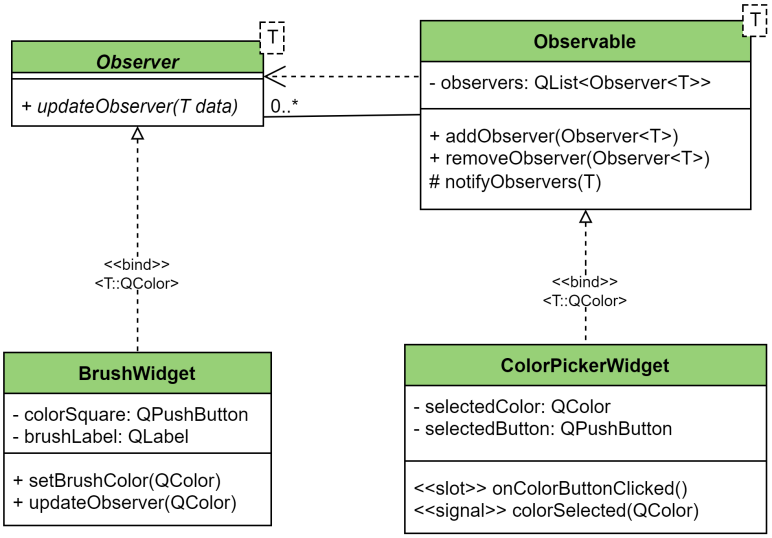


Figure 6.10: Observer Section from the Class Diagram

The Observer Section (6.10) is contained in the Core Section and it mainly concerns the BrushWidget and the ColorPickerWidget. The reason for implementing an Observer Pattern between the two relates to the fact that there are a lot of signal and slot connections that deal with important actions in the applications, and connecting these widgets using the same mechanism is unnecessary, given that this feature only exists to improve on the user experience, and does not facilitate any major functionality. Therefore we have the BrushWidget acting as the Observer and the ColorPickerWidget as the Observable. The abstract classes are templated so they could be used in the context of future developments that would require a different type of data to be exchanged between the two entities. In this case, the data sent from the ColorPickerWidget to the BrushWidget is a QColor value, used by the BrushWidget to update the color of its `colorSquare`.

6.3 UX Design

The idea of a smooth and intuitive user experience has been considered since the early stages of development. This focus is not merely about building attractive visual components but has a deeper implication - it's about creating a user-centric design that facilitates effortless interactions, promotes user satisfaction, and ultimately acts as motivation for future engagement.

Transitioning from the broad strokes of the user experience design, the focus shifts to the tangible elements of Pix2Sat. As seen in Figure 6.11, the interface layout closely resembles the classic design of any modern integrated development environment combined with elements from media-editing software such as Photoshop or Paint. The edges of the screen are dedicated to the functionality-oriented components, leaving the drawing area to be the most prominent component of the application window.

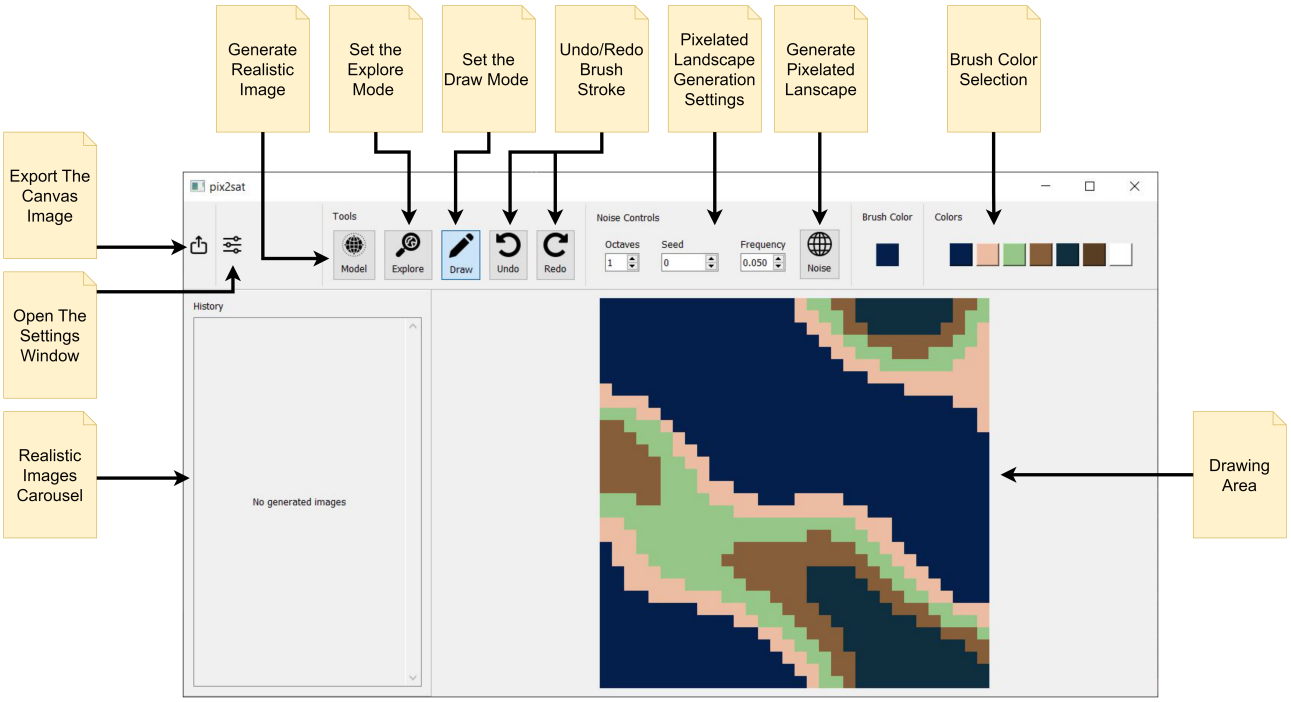


Figure 6.11: Interface Overview of Pix2Sat

Analyzing the application window from left to right, top to bottom, the first feature that comes into view is the "Export Canvas Image" button, which allows the user to save the current drawing as an image. When clicked, a native file manager will pop up, allowing the user to save the image as any one of a few supported image formats. Moving on, the "Settings" button opens a different window where the user has the option to modify values used for different functionalities of the application (see Figure 6.12). The "Noise Type" selectable value refers to the procedural noise function used to generate the pixelated terrain, while the "Model" selectable value refers to the AI model that is used when generating realistic landscape images. The values with colors associated with them represent the

upper bounds used when transforming numeric noise into colorful pixelated landscapes. These bounds control the prominence of each color in the final result.

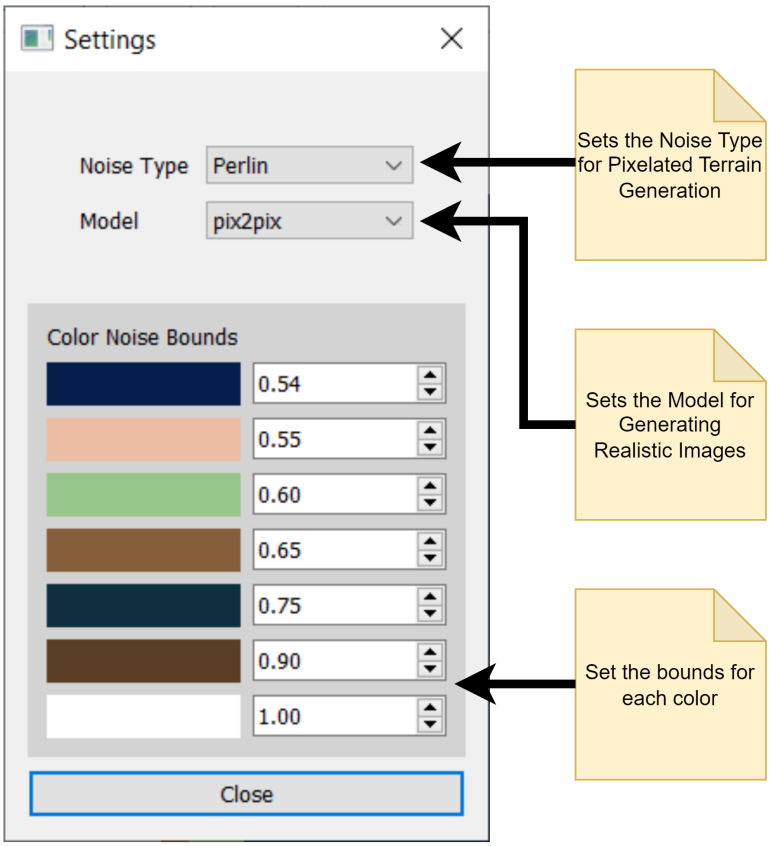


Figure 6.12: Settings Window of Pix2Sat

Distinctly grouped, the tool buttons - Model, Explore, Draw, Undo, and Redo - are found in their dedicated section. Pushing the Model runs the generative model on the pixelated image from the drawing area and when done, the output will appear in the "Realistic Images Carousel" list. The "Explore" button puts the drawing area into the Explore Mode. This change is indicated by the fact the mouse cursor changes from the cross cursor, which indicates the drawing mode, to the open-hand cursor. The user can then left-click and drag on the canvas and explore the pixelated landscape in order to find the most suitable location. To enhance the user's experience when using this feature, the cursor changes from an open hand to a closed hand during the dragging process, visually mimicking the action. In opposition to the "Explore" button is the "Draw" button, which sets the drawing area to its default mode. The "Undo" button reverses the last performed brush stroke on the canvas, while the "Redo" button reapplies the brush stroke that was previously undone. In the Noise Controls section, the user can control the frequency, octaves, and seed values in order to fine-tune the pixelated output. The default values of these fields are chosen such that the user wouldn't need to interact with them

in order to obtain satisfying results. In the same section, the user can trigger the pixelated terrain generation process by pressing the "Noise" button. The brush color can be chosen from the Colors section, the current color is prominently displayed in the Brush Color indicator section.

Moving on from the overall interface, a look into a general workflow a user could potentially go through would reveal the true nature of the application (see Figure 6.13). Initially, the first step is to choose a base for the landscape. The Noise parameters can be tweaked in order to change different features for the landscape generation. This can be used in combination with the Explore functionality in order to pinpoint the perfect base. Next, the user can add their own modifications to the base, using the available brush colors. When the pixelated landscape is to the users liking, simply pressing the "Model" button would prompt the realistic image obtained from the user's pixel art to appear in the History section. Optionally, all the images from the History section can be saved by right-clicking on a specific image and selecting the "Save Image" option.

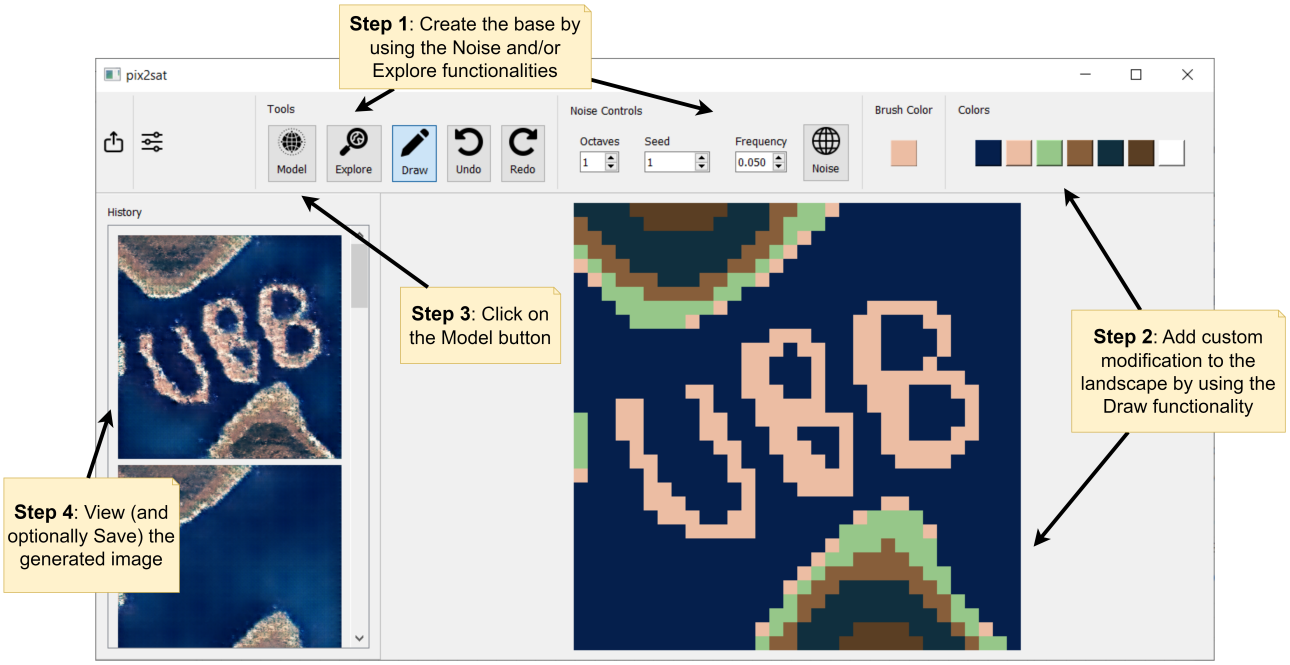


Figure 6.13: A general workflow in Pix2Sat

Chapter 7

Conclusion and future work

The proposed method has proven to be a valid solution for translating pixel art terrain images into realistic satellite-quality images. Moreover, experimentation has led to further improvements in the initial solution. Although the use of ResNet as a qualitative Generator was invalidated, a higher degree of detail has been achieved by employing a 34×34 PatchGAN as the used Discriminator, and the introduction of the Total Variation Loss greatly reduced the number of land sliver artifacts from the output. In the case of the latter, it was pointed out that it does not entirely solve the problem if the presence of the aforementioned artifacts is indeed considered a flaw. A possible method to be implemented in future work that would further reduce the frequency of the artifacts could be the introduction of a post-processing step in the proposed pipeline. As a general description, the post-processing step could include training another model to identify the position of the artifacts in the output image and remove them.

Considering the fact that the proposed approach of generating pixel art with procedural noise and further using a Generator model to obtain high-quality terrain images is a novel approach in the field of Terrain Generation, there is always room for improvement. Diving into specifics, the color palette was chosen arbitrarily, therefore future work could investigate methods to obtain the color palette automatically from a given dataset. Also on the topic of improvements related to colors, research into a more suitable color distance would greatly increase the quality of the dataset.

Hardware limitations were one of the driving factors in the development of the methodology since GANs usually require a lot of computing power, hence further experimentation, given enough resources, could yield even more impressive results.

Regarding the numerical evaluation of the experimental models, the metrics used, although mostly accurate, are not designed for the translation at hand. The development of a metric specifically designed for this task would help evaluate and understand the performance and quality of future models.

Overall, the method detailed throughout this thesis has successfully filled the gap identified in the current state of AI-powered Terrain Generation. Although a simple method indeed, it is able to produce results comparable to state-of-the-art methods. The simplicity of the method is, in my opinion, reminiscent of the simplicity that Generative Adversarial Nets initially brought to the field of Generative AI. Despite this trait, it demonstrated the ability to handle a wide range of terrain textures. Integrating the method into a software application has extended its usability, making it accessible for various purposes. It is also duly acknowledged that there is always room for improvement. As stated in this chapter, future work could tackle a diverse set of possible improvements that would refine the performance of the method. This being said, this thesis has shown that simplicity can indeed craft its own masterpiece amidst the dance of algorithms and data.

Bibliography

[1] L Abady, M Barni, A Garzelli, and B Tondi. Gan generation of synthetic multispectral satellite images. In *Image and Signal Processing for Remote Sensing XXVI*, volume 11533, pages 122–133. SPIE, 2020.

[2] Travis Archer. Procedurally generating terrain. In *44th annual midwest instruction and computing symposium, Duluth*, pages 378–393, 2011.

[3] Daniel A Ashlock, Stephen P Gent, and Kenneth Mark Bryden. Evolution of l-systems for compact virtual landscape generation. In *2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2760–2767. IEEE, 2005.

[4] Michael F Barnsley, Robert L Devaney, Benoit B Mandelbrot, Heinz-Otto Peitgen, Dietmar Saupe, Richard F Voss, Yuval Fisher, and Michael McGuire. *The science of fractal images*, volume 1. Springer, 1988.

[5] Christopher Beckham and Christopher Pal. A step towards procedural terrain generation with gans. *arXiv preprint arXiv:1707.03383*, 2017.

[6] John Conway et al. The game of life. *Scientific American*, 223(4):4, 1970.

[7] Ugur Demir and Gozde Unal. Patch-based image inpainting with generative adversarial networks. *arXiv preprint arXiv:1803.07422*, 2018.

[8] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[9] Miguel Frade, F Fernandez de Vega, and Carlos Cotta. Evolution of artificial terrains for video games based on obstacles edge length. In *IEEE congress on evolutionary computation*, pages 1–8. IEEE, 2010.

[10] Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta. Evolution of artificial terrains for video games based on accessibility. In *Applications of Evolutionary Computation: EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I*, pages 90–99. Springer, 2010.

[11] Miguel Frade, Francisco Fernandéz de Vega, and Carlos Cotta. Breeding terrains with genetic terrain programming: the evolution of terrain generators. *International Journal of Computer Games Technology*, 2009, 2009.

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[13] Noel Gorelick, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 2017.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[15] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[16] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[17] Jordan Peck. Fastnoiselite. https://github.com/Auburns/FastNoiseLite.

[18] Ares Lagae, Sylvain Lefebvre, Robert L Cook, Tony DeRose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. State of the art in procedural noise functions. *Eurographics (State of the Art Reports)*, pages 1–19, 2010.

[19] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic

single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[20] Gavin SP Miller. The definition and rendering of terrain maps. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 39–48, 1986.

[21] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[22] J von Neumann. Theory of self-reproducing automata. *Mathematics of Computation*, 21:745, 1966.

[23] Teong Joo Ong, Ryan Saunders, John Keyser, and John J Leggett. Terrain generation using genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1463–1470, 2005.

[24] Ian Parberry. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques*, 3(1), 2014.

[25] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[26] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[27] Procedural Worlds. Gaia. Computer software, 2015.

[28] QuadSpinner. Gaea. Computer software, 2020.

[29] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[30] William L Raffe, Fabio Zambetta, and Xiaodong Li. Evolving patch-based terrains for use in video games. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 363–370, 2011.

[31] William L Raffe, Fabio Zambetta, and Xiaodong Li. A survey of procedural terrain generation techniques using evolutionary algorithms. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[32] Richard Garriott's Origin Systems. Akalabeth: World of doom. Computer game, 1980. Released for Apple II.

[33] Nelson Rodrigues, Miguel Frade, and F Fernández de Vega. Development of chapas an open source video game with genetic terrain programming. In *VII Congreso Espanol sobre Metaheurısticas, Algoritmos Evolutivos y Bioinspirados (MAEB), Valencia, Spain*, pages 1–8, 2010.

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015.

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[36] Ryan L Saunders. Realistic terrain synthesis using genetic algorithms. *Texas A&M University*, 2006.

[37] Mojang Studios. Minecraft. Video game, 2011.

[38] The Third Floor. World creator. Computer software, 2015.

[39] Yann Thorimbert and Bastien Chopard. Polynomial methods for fast procedural terrain generation. *arXiv preprint arXiv:1610.03525*, 2016.

[40] Julian Togelius, Mike Preuss, and Georgios N Yannakakis. Towards multiobjective procedural map generation. In *Proceedings of the 2010 workshop on procedural content generation in games*, pages 1–8, 2010.

[41] Unity Technologies. Unity terrain tools. Computer software, 2009.

[42] Georgios Voulgaris, Ioannis Mademlis, and Ioannis Pitas. Procedural terrain generation using generative adversarial networks. In *2021 29th European Signal Processing Conference (EUSIPCO)*, pages 686–690. IEEE, 2021.

[43] Paul Walsh and Prasad Gade. Terrain generation using an interactive genetic algorithm. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE, 2010.

[44] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[45] Hua Fei Yin and Chang Wen Zheng. A practical terrain generation method using sketch map and simple parameters. *IEICE TRANSACTIONS on Information and Systems*, 96(8):1836–1844, 2013.

[46] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[47] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

[48] Peter Ziegler and Sebastian von Mammen. Generating real-time strategy heightmaps using cellular automata. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pages 1–4, 2020.