

iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA

3D Terrain Generation using Neural Networks

Rodrigo de Matos Pires Tavares de Almeida

Master in Computer Engineering

Supervisor:

PhD Pedro Figueiredo Santana, Assistant Professor,
Iscte - Instituto Universitário de Lisboa.

October 2020

“There is no greater sorrow than to recall our times of joy in wretchedness.”

— *Dante Alighieri, Inferno*

Acknowledgements

Quero agradecer ao IT por ter emprestado o espaço para o desenvolvimento desta tese e de igual forma agradecer ao Professor Pedro Figueiredo Santana por se ter disponibilizado para orientar esta dissertação.

Um muito especial agradecimento aos meus Pais por me terem acompanhado nesta longa trajetória acadêmica, que durou muito mais do que seria desejado. Sem o seu infinito apoio nada disto teria sido possível, não só nesta etapa, mas na minha vida toda. Estarei eternamente agradecido por me terem dado esta oportunidade.

À minha namorada, que aguentou todos os meus momentos baixos causados por esta tese, que, de igual forma, se manteve ao meu lado e me deu alento para progredir, mesmo quando eu menos merecia. O meu eterno amor.

Aos meus amigos que me acompanharam de perto, e de longe, durante esta etapa. Que me deram força e sorrisos ao longo deste tempo todo. Aos meus colegas Bernardo Ribeiro, João Bernardo, João Pereira, Kevin Ramos e Robert DeHaven, que graças ao seu companheirismo fizeram com que esta experiência fosse muito mais divertida. Um especial agradecimento ao João Pereira que graciosamente se pôs ao dispor para me ajudar e opinar sobre o desenvolvimento desta tese. Um grande abraço.

A todas estas pessoas, um sentido obrigado.

Resumo

Com o aumento do poder de computação, juntamente com os avanços neste campo na forma de GANs e cGANs, as Redes Neurais tornaram-se numa proposta atrativa para a geração de conteúdos. Graças a estes avanços, abriram-se oportunidades para os algoritmos de Geração de Conteúdos Procedimentais (PCG) explorarem o poder generativo das Redes Neurais para a criação de ferramentas que permitam aos programadores remover parte da carga criativa e de desenvolvimento imposta em toda a indústria dos jogos, seja por parte dos investidores que procuram um retorno do seu investimento ou por parte dos consumidores que querem mais e melhor conteúdo, o mais rápido possível. Esta dissertação pretende desenvolver uma ferramenta de iniciativa mista PCG, alavancando cGANs, para criar terrenos 3D cocriados, permitindo aos utilizadores influenciarem diretamente o conteúdo gerado sem necessidade de terem formação formal sobre a criação de terrenos 3D ou interações complexas com a ferramenta para influenciar a produção generativa, opondo-se assim a algoritmos generativos comumente utilizados, que apenas permitem a geração de conteúdo aleatório ou que são desnecessariamente complexos. Um conjunto de testes feitos a 113 pessoas online e a 30 pessoas presencialmente, revelaram que é de facto possível desenvolver uma ferramenta que permita aos utilizadores, de qualquer nível de conhecimento sobre criação de terrenos, e com uma formação mínima na ferramenta, criar um terreno 3D mais realista do que os terrenos gerados a partir da solução de estado da arte, como o Perlin Noise, e de uma forma fácil.

Keywords: Redes Neurais, Geração Procedimental de Conteúdos, PCG, GAN, cGAN, Terrenos 3D, Perlin Noise.

Abstract

With the increase in computation power, coupled with the advancements in the field in the form of GANs and cGANs, Neural Networks have become an attractive proposition for content generation. This opened opportunities for Procedural Content Generation algorithms (PCG) to tap Neural Networks generative power to create tools that allow developers to remove part of creative and developmental burden imposed throughout the gaming industry, be it from investors looking for a return on their investment and from consumers that want more and better content, fast. This dissertation sets out to develop a PCG mixed-initiative tool, leveraging cGANs, to create authored 3D terrains, allowing users to directly influence the resulting generated content without the need for formal training on terrain generation or complex interactions with the tool to influence the generative output, as opposed to state of the art generative algorithms that only allow for random content generation or are needlessly complex. Testing done to 113 people online, as well as in-person testing done to 30 people, revealed that it is indeed possible to develop a tool that allows users from any level of terrain creation knowledge, and minimal tool training, to easily create a 3D terrain that is more realistic looking than those generated by state-of-the-art solutions such as Perlin Noise.

Keywords: Neural Networks, Procedural Content Generation, PCG, GAN, cGAN, 3D Terrain, Perlin Noise.

Table of Contents

Acknowledgements	iii
Resumo	v
Abstract	vii
Table of Contents	ix
List of Figures	xi
Glossary of acronyms	xiii
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Research Questions	8
1.4 Objectives	8
1.5 Document Structure and Organization.....	8
Chapter 2 Literature Review	9
2.1. Artificial Neural Networks (ANNs).....	9
2.1.1 Deep Neural Networks (DNNs).....	10
2.1.2 DNN Architectures.....	11
2.1.2.1 Convolutional Neural Network (CNNs).....	11
2.1.2.2 Recurrent Neural Networks (RNNs)	13
2.1.2.3 Generative Adversarial Networks (GANs)	14
2.1.2.4 Types of GAN	15
2.1.2.5 Image-to-Image Translation by Isola et al. (pix2pix)	17
2.2 Procedural Content Generation Definition	19
2.2.1 Brief History of PCG in Commercial Games.....	20
2.2.2 PCG Tools.....	22
2.2.3 PCG Taxonomies	24
2.2.4 PCG Methods and Approaches.....	27
2.2.4.1 Traditional PCG methods.....	28
2.2.4.2 Search-Based PCG methods	29
2.2.4.3 Machine Learning methods.....	29
2.6 Previous and Related Work in Procedural Terrain Generation.....	30
2.3.1 Recursive Subdivision	30
2.3.2 Noise	31
2.3.3 Evolutionary Algorithms	32
2.3.4 Procedural Brushes.....	33

2.3.5	Software Agents	33
2.3.6	Terrain Generation using Generative Adversarial Networks	34
2.3.6.1	Using real world data without authoring	34
2.3.6.2	Using real world data with authoring.....	37
Chapter 3 Tool's Design and Implementation.....		39
3.1	Tool Component Overview.....	39
3.1.1	Neural Network System Requirements.....	41
3.1.2	Tool Flow	42
3.2	Single Input Preparation.....	43
3.3	Neural Network Sensitivity Analysis.....	44
3.4	Software Structure and Terrain Generation Logic	50
3.5	Interface version 1.....	53
3.6	Interface version 2.....	57
3.7	Image Log and Copy/Paste	59
3.8	Perlin Noise generator.....	60
Chapter 4 Experimental Results.....		63
4.1	Terrain Generation Comparison Test	63
4.2	Usability Test	64
4.3	Terrain Generation Comparison Test Results.....	68
4.4	Usability Test Results.....	70
4.5	Second Usability Test Results	82
Chapter 5 Conclusions.....		89
Bibliographical references.....		93
Annex.....		104
Annex A NN Behavioral Test Pairs.....		104
Annex B Terrain Comparison Test Pairs.....		144
Annex C Usability Test Guide.....		154
Annex D Second Usability Test Guide.....		160

List of Figures

Figure 1.1 - Projected market growth by segment	1
Figure 1.2 - Regional breakdown of global game revenue (Newzoo, 2019).	2
Figure 1.3 - Age Breakdown of video game player in the USA in 2019.....	2
Figure 2.1 - McCulloch neuron.	9
Figure 2.2 - ANN example	10
Figure 2.3 - Example of CNN.....	11
Figure 2.4 - Max pooling example	12
Figure 2.5 - RNNs information flow (Olah, 2019).....	13
Figure 2.6 - Comparison between regular RNN and Gated Recurrent Units.....	13
Figure 2.7 - Vanilla GAN architecture	14
Figure 2.8 - DCGAN Generator architecture.	15
Figure 2.9 - cGAN architecture example	17
Figure 2.10 - Left - Auto-Encoder.....	18
Figure 2.11 - PatchGAN Discriminator	19
Figure 2.12 - EA logic flow.....	32
Figure 2.13 - Beckham et al. resulting terrain (Beckham & Pal, 2017).	35
Figure 2.14 - Example of final render shown to test participants	36
Figure 2.15 - Spick et al. resulting heightmaps and 3D renders of three different generation of training	37
Figure 2.16 - Guérin et al. different inputs and their corresponding outputs.....	37
Figure 3.1 - Tool component diagram.....	40
Figure 3.2 - Application Pipeline.....	42
Figure 3.3 - User Input/NN output pair.....	43
Figure 3.4 - Initial test inputs and their corresponding output.....	45
Figure 3.5 - More test inputs and their resulting heightmaps.	47
Figure 3.6 - Pattern testing with different opacities.	48
Figure 3.7 - More pattern testing with different opacities.....	48
Figure 3.8 - Color variation testing. Original pattern on top left corner.....	49
Figure 3.9 - Dark green and red weight difference.	50
Figure 3.10 - Class Inheritance graph.	51
Figure 3.11 - GenerateMap function call graph.	52
Figure 3.12 - First iteration of the interface, with a red numeric overlay.....	54

Figure 3.13 - Interface detail: action buttons.	55
Figure 3.14 - Interface detail: drawing tools.	56
Figure 3.15 - Version two of the Interface.	58
Figure 3.16 - At the top of the interface, Log menu, with generated terrain preview	59
Figure 3.17 - Copy/Paste loop	60
Figure 3.18 - Perlin Noise editor fields.	61
Figure 3.19 - Perlin Noise comparison.	61
Figure 3.20 - Perlin noise terrain example.	62
Figure 4.1 - Example of terrain pair used in the Comparison Test.	63
Figure 4.2 - Participant choice distribution between NN and PN by pair	68
Figure 4.3 - Box plot distribution of NN pick percentage	69
Figure 4.4 - Results for Phase 1 Group 1 question 1a) and Phase 1 Group 1 1b).	70
Figure 4.5 - Results for Phase 1 Group 1 question 1c) and Phase 1 Group 1 question 1d).....	71
Figure 4.6 - Results for Phase 1 Group 2 question 2a) and Phase 1 Group 2 question 2b).	72
Figure 4.7 - Results for Phase 1 Group 2 question 2c), Phase 1 Group 2 question 2d) and Phase 1 Group 2 question 2e).	73
Figure 4.9 - Results for Phase 1 Group 4 questions 4a), 4b) and 4c).	76
Figure 4.10 - Results for Phase 4 question 1 and Phase 4 question 2.....	77
Figure 4.11 - Results for Phase 4 question 3 and Phase 4 question 4.....	78
Figure 4.12 - Results for Phase 4 question 5 and Phase 4 question.	79
Figure 4.13 - Results for Phase 4 Question 7.	80
Figure 4.14 - Results for question 1a) and 1b), second usability test.	83
Figure 4.15 - Results for question 1c) and 1d), second usability test.	83
Figure 4.16 - Results for question 2a) and 2b), second usability test.	84
Figure 4.17 - Results for question 2c) and 3a), second usability test.	85
Figure 4.18 - Results for question 4a), 4b) and 4c) of second usability test.....	86

Glossary of acronyms

NN Neural Network

ML Machine Learning

PCG Procedural Content Generation

ANN Artificial Neural Network

GAN Generative Adversarial Network

cGAN Conditional Generative Adversarial Network

DCGAN Deep Convolutional Generative Adversarial Network

CNN Convolutional Neural Network

RNN Recurrent Neural Network

DNN Deep Neural Network

PN Perlin Noise

EA Evolutionary Algorithm

MOEA Multi-objective evolutionary algorithm

UI User Interface

SD Standard Deviation

Introduction

1.1 Motivation

The video game industry is a growing one, boasting more than 2.5 billion players worldwide (Newzoo, 2020). Trend analyses show a constant growth over the previous several years and projected to grow by a considerable margin in the next few years. It is safe to say that this industry is in very good shape.

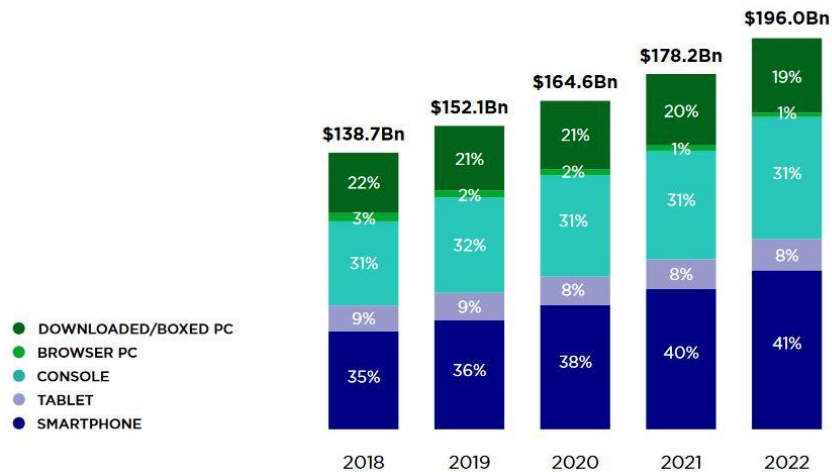


Figure 1.1 - Projected market growth by segment (Newzoo 2019).

In 2019, the video game market had an estimated value of \$152.1 billion and by the end of this year (2020) it is expected to increase by 8.2 % to \$164.6 billion, with all segments showing growth (Mobile leading, with consoles trailing behind and PC in third) with the exception of browser games due to the migration of the now dying genre to Mobile platforms and to boxed/downloadable PC games, as seen in Figure 1.1. To put this in perspective, the video game market is almost three times the size of the movie industry and music industry, combined (IFPI, 2019) (OppenheimerFunds, 2018). The entirety of the video game industry shows growth and is in a healthy state with compound annual growth rate of global games market, between 2018 and 2022, projected to be +9.0 % , with all major world regions remaining stable on their percentage of the revenues, as seen in Figure 1.2. Furthermore, if we analyze USA’s Entertainment Software Association (ESA) statistics¹ regarding video game players demography, we are faced with values that completely dispel the notion that gaming is for young male adults (Yannakakis & Togelius, 2011).

¹ Using USA numbers, firstly because it’s the second biggest market in the world, closely following China ("Top Countries & Markets by Game Revenues | Newzoo", 2020), secondly because it’s more heavily surveyed thus having more statistical information freely available and finally because it’s the country with the biggest number of developer and publishing studios (www.gamedevmap.com, 2020).



Figure 1.2 - Regional breakdown of global game revenue (Newzoo, 2019).

Of all American adults, 65 % play video games with the average male gamer age being 32 and 34 for woman. Men represent 54% of the market and woman 46% (Theesa.com, 2019). Moreover, to further dispel this notion, the percentage of people playing video games below the age of 18 is the same as the percentage of people with 50, or more, playing video games, as seen in Figure 1.3 (Gough, 2020).

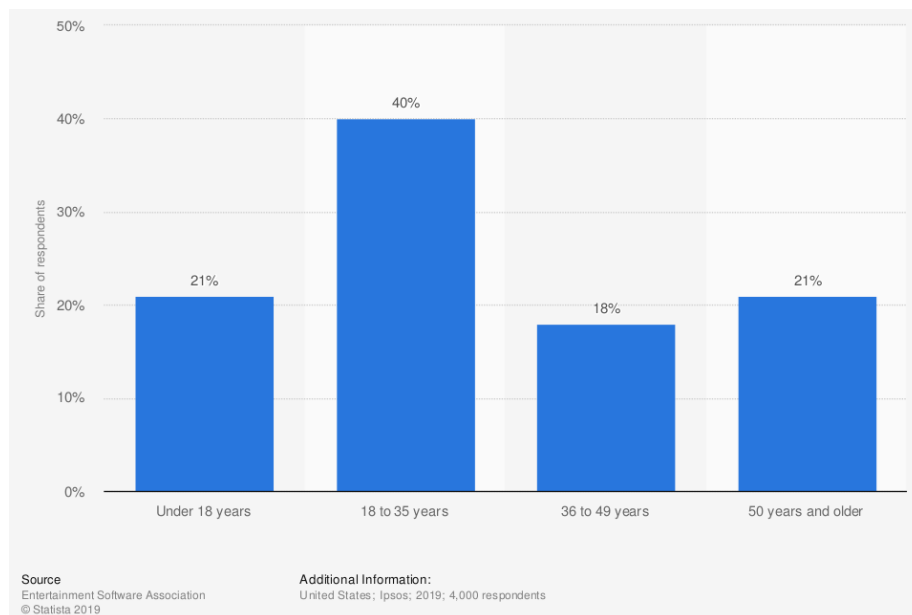


Figure 1.3 - Age Breakdown of video game player in the USA in 2019.

Gone are the times where gamers were mostly young white males with technological interests. Today everybody plays video games meaning that the range of skills, preferences, preferred emotional elicitations is as wide as it can be (Yannakakis & Togelius, 2011). This means that developers now have a gigantic scope of possible game content to create, depending on the target demographic profile

leading to a fast growing necessity in creating more tailored video games and faster (Yannakakis & Togelius, 2011).

Just like any other technological industry, gaming evolved along with hardware and so did the demands of the players. Better hardware meant more, better in quality and more regularly released, game content. Back in the 1990s, games like *Wolfenstein 3D* and *Doom* were created with one full-time content developer in a developing team made of five to six people with budgets well below the million-dollar mark (Kushner, 2004). By early 2000s, developing teams were already in the hundreds (Krueger et al. 2005) with multi-million budgets. One example is *World of Warcraft (WoW)*, a game with a scope never seen before (initial release date was 2004), had an estimated budget between \$20 million and \$150 million (Johnson, 2006) and *WoW* is updated with more game content regularly for the past 16 years. Just 4 years later, *GTA 4* is released and in an interview with *TheTimes*, *GTA 4* producer Leslie Benzies, confirmed that the Budget was approximately 100 million dollars and had more than 1000 people working on it (Androvich, 2008).

ESA reports that quality of graphics and story/premise are two of the main reasons that drive a consumer to buy a game. (Theesa.com, 2019). Unfortunately for developers, even though video game technology has advanced significantly, allowing for the creation of games with increasing realism and complexity, game content creation is still largely manual, meaning that 2 of the main reasons consumers buy games represent the majority of time spent developing a game (Yannakakis & Togelius, 2011) and a considerable chunk of the budget, nearing the 30%-40% mark (Hendriks, Meijer, Van Der Velden, & Iosup, 2013b). This creates a bottleneck in the budget, time-to-market, but more importantly, it creates a risk aversion in investors that find taking multi-million-dollar risks to try out new concepts, business wise, a very bad idea (Yannakakis & Togelius, 2011) (Hendriks et al., 2013b).

Ultimately, these monetary and time constraints leave game content creators in a creative bankruptcy. Required to meet deadlines, adapt to any scope or direction change but at the same time keep the quality fidelity that a multi-million-dollar project demands. This forces game content creators to rehash content to be able to meet the demands of publishers and investors to remain at the peak of profitability, resulting in imitation and stagnation of the industry (Semuels, 2019)(White, 2009) (Katzenbach, Herweg, & Van Roessel, 2016).

It is unmistakable that any technology that lessens the resources that go into content creation (especially graphical assets) and shortens the time necessary to tailor craft content to specific groups of players would be more than welcomed by all the participants involved, from developers to consumers. Following this logic, having a tool that allows to procedurally generate a game map, which

is a foundational element in any 3D game (Togelius, Preuss, & Yannakakis, 2010) (Forbus, Mahoney, & Dill, 2002) (it sets the tone, entices players with its mysteries and drives gameplay forward (Wulff-Jensen, Rant, Møller, & Billeskov, 2018)), would be something worth having, especially if the artist/designer can still manipulate several parameters of the creation procedure. To ease the creative and development burden brought in by rising consumer demands, several games in the past have used Procedural Content Generation (PCG) techniques. PCG consists in generating content algorithmically, which otherwise would be created by a human.

Games such as *Rogue* (1980), *Elite* (1985) and *The Sentinel* (1985) all used PCG techniques mainly to reduce the size of the game files and not necessarily related to PCG. More recently, we have examples of commercial games that used PCG such as the *Diablo Series* (1997-2012), *Dwarf Fortress*, *Civilization IV* map generation (Togelius, Preuss, & Yannakakis, 2010), “.*kkrieger*” (Farbrausch Prod, 2006), *Spore*, *Spelunky* (Hendrikx et al., 2013b), *Tiny Wings* (N. Shaker, Togelius, & Nelson, 2016) and the *Borderlands* series (2009-2019). All these games use one or more PCG techniques and not necessarily PCG map generation, given the indication that PCG is commercially viable and well received by consumers.

Truly compelling games that exemplify the successful implementation of these technology are games such as *Minecraft* and *No Man’s Sky*. Both are commercial successes, both use PCG to create their game worlds and both have almost incomprehensible sizes: 1024×10^{12} blocks (Mojang, A B. 2013. *Minecraft*. Stockholm, Sweden.) and 18×10^{18} plus planets (Cook, 2016), respectively. This size and scope for any game is beyond our wildest dreams if we stick strictly to manual content creation making these two games prime examples of how important map generation is and why we should be developing robust and fast map generation solutions: without an easily traversable and engaging map, the player will not be immersed enough to captivate user interest and attention, leaving the project in a position that might be financially non-viable (future downloadable content and/or monetization paths will not reach the user simply because it no longer is captivated by the title).

Another argument for automatic map generation is game replayability (Sampath, 2004). Players stuck with the same map for the entirety of the game can get bored, master the map too quickly, in the case of the map being either too easy or too small, or dislike the map altogether. With an automatic map generator, we can change map presentation to the user and enable a fresh experience whenever the developer sees fit, for example, when a new campaign is started, when the player backtracks, to apply terrain deformations to convene map destruction tied to the narrative. This possibility would also help in taking game content creators away from potentially having to crunch ludicrous hours (Gilbert, 2019) and avoiding profit-driven creative bankruptcy.

Current algorithms to automatically generate 3D terrain and in-house developed methods for the same objective are either too time consuming, expensive, too flawed for today's standards or a combination of these three. Such examples include Diamond Square (Fournier, Fussell, & Carpenter, 1982), Perlin Noise (Perlin, 1985) (Perlin, 2002) and Midpoint Displacement (Fournier et al., 1982) (Koh & Hearn, 1992). Midpoint Displacement is a simple and fast implementation algorithm, but its generative simplicity is unsuitable for today's high-profile games and it suffers from recurring patterns. Perlin Noise, a noise based heightmap and texture generator, is inefficient with higher dimensional noise creating directional artifacts (Miller, 1986) and, according to (Miller, 1986) "generate defects due to what is known as the 'creasing problem' which is the occurrence of creases or slope discontinuities along boundaries". Finally, in-house developed algorithms like the ones proposed by Shaker et al. (N. Shaker et al., 2016) can require a lot of handcrafting and human resources to reach a desired quality threshold (Togelius, Preuss, & Yannakakis, 2010).

To add to the above uses, gamified technology and complete games are frequently used in training, simulation scenarios, decision support in various sectors in our society and in education. Rescue operatives need to train their skills in a consequence free environment, the military needs simulations to train their soldiers and pilots, industries ranging from logistics to customer service use simulations to train their workers (Yannakakis & Togelius, 2011). Nowadays, in Portugal, it is required to drive in the simulator for a pre-determined time before driving a real vehicle. Students would benefit from exposure to varied scenarios in which they can apply acquired knowledge in new and novel situations (Freiknecht & Effelsberg, 2017). The more realistic these scenarios are, allied with the potential voluminous output of new scenarios, the more effective, accurate and diverse is the training.

Given the growth of the gaming industry, the importance of 3D terrain in games and the rising quality demands, both from producers and consumers regarding content quality and quantity, which place extra pressure on developers, there is a growing need for tools that remove part of the burden from the creative and development process. This dissertation sets out to contribute to the solution of these problems by tapping the PCG body of work for the development of a tool that allows its users to create 3D terrains, that is both easy to use and real-time but still produce realistic looking results.

1.2 Context

This dissertation contributes towards the used of Machine Learning (ML) algorithms, more specifically, Deep Neural Networks (DNNs) as the main generative engine in a PCG algorithm to mitigate development burden and its associated rising costs. It is framed in three main areas: Machine Learning (ML), Computer Vision and Procedural Content Generation (PCG). The first related field, ML, is an

artificial intelligence branch, more specifically, a data analysis method that has its objective set in automating analytical model building. These systems are grounded on the idea that such systems can learn from data and be able to identify patterns and perform decisions with minimal human supervision. The second related field is Computer Vision, which is tasked with automatic extraction of high-dimensional data, analyzing and understanding the information contained within a single or sequence of images to produce numerical or symbolical information with the objective of creating theoretical and/or algorithmic data.

DNNs and Computer Vision algorithms are not something new and novel. In recent years, supervised DNNs have been experiencing success on the field of computer vision, partly because computation performance has gone up steadily since its inception, accordingly to Gordon E. Moore prevision (Schaller, 1997). One of those successes was the winner of the *Imagenet Classification Challenge* in 2012 that accomplished an improvement of 10% over the winner of the previous year (Krizhevsky, Sutskever, & Hinton, 2012) becoming state-of-the-art.

Since then, several improvements and breakthroughs have been made, further advancing not only the classification percentage, but also DNNs versatility and a wider range of use cases. Several developments were made by different research teams, Goodfellow et al. gave us Generative Adversarial Networks (GAN) (I. J. Goodfellow et al., 2014), making this work one of the most influential in the generative image modeling field. Afterwards, several modifications and improvements were made to this design. For instance, Radford et al. (Radford, Metz, & Chintala, 2016) presented deep convolutional generative adversarial networks with a more stable set of constraints to allow for a more stable GAN training. Arjovsky et al. (Arjovsky, Chintala, & Bottou, 2017) developed the Wasserstein GAN, which allows for better parameterization while training (Mirza & Osindero, 2014) that developed the Conditional GAN (cGAN) that allows the addition of a conditional label that enables the network to be directed in the generation process. This final DNN is the network used in this dissertation as the generative engine tasked with generating 3D terrain. More specifically the network developed by (Isola, Zhu, Zhou, & Efros, 2016) and translated to Tensorflow, an open-source software library used numerical computation and large scale machine learning, by Guérin et al. (Guérin et al., 2017).

Lastly, there is PCG, which can be broadly defined as the algorithmic creation of content and assets for traditional and digital media such as games, movies and simulations requiring none to minimal user input, especially when compared with the hand-made content creation process (Freiknecht & Effelsberg, 2017). This includes any content that would be usually created by hand, such as maps, textures, sound effects, music, weapons, rules, and game mechanics. This dissertation focuses primarily on functional content, meaning content that impacts the user's interactive experience, rather

than cosmetic content, mainly because generative graphic and sound has been garnering plenty of attention in the past decades, while functional content generation is still considered a niche subject, especially when in regards of implementation in large commercial projects (Freiknecht & Effelsberg, 2017).

PCG has been around for a few decades already (Peachey, 1985). Commercially viable PCG implementations date back to 1980, with *Rogue*, a dungeon crawler game in which levels are procedurally generated at the beginning of each playthrough that was so critically acclaimed it spawned, what is nowadays known as, Rogue-like games. Another noteworthy implementation is *Elite*, a space trading video game released in 1984, whose PCG implementation does not differ that much from *No Man's Sky*: each galaxy, its planets and their contents were generated from a single seed number that ran through the algorithm, thus circumventing memory limitation and allowing for a far more expansive game if the only option was to store all these assets in memory (Freiknecht & Effelsberg, 2017)(Khaled, Nelson, & Barr, 2013)(N. Shaker et al., 2016)(Barriga, 2019).

In each of these embryonic PCG implementations the goal was not to make it a game element but to beat a resource bottleneck (low available memory that prevented developers to add the necessary content for replayability) (Freiknecht & Effelsberg, 2017). Although there are several, more contemporary, examples of PCG implemented as a deliberate game mechanic, such as *Diablo* with its generated dungeons, *The Elder Scrolls V: Skyrim* generated quests and *Borderlands* generated guns (Freiknecht & Effelsberg, 2017)(Hendrikx et al., 2013b), there seems to be a growing need for more robust tools that allow developers to beat the creative bottleneck and development burden discussed in Section 1.1.

Due to the performance increase in ML and NNs, the potential shown in image classification by these technologies and the popularization of GANs and others types of NNs, there is an increased interest in crossing PCG and ML so we can better leverage the power of both to create new content automatically, thus creating a new terminology for these kinds of algorithms: Procedural Content Generation via Machine Learning (PCGML) (Torrado et al., 2019). The tool proposed by this dissertation is inserted in this classification, as it aims to help streamline functional content generation, by showing that functional content generation has a place on the development process.

1.2 Research Questions

As mentioned in Section 1.1, PCG algorithms are still not recurrently seen as solutions for time and economic constraints in development, despite showing promising prospects of helping development flow in a more streamlined fashion. Having this in mind, this dissertation set out to answer the following questions:

Q1 - Can we leverage machine learning to create a tool that allows users of varied fields of experience to create a realistic looking 3D terrain without requiring any formal training?

Q2 - Are the maps generate by the developed tool regarded as more realistic looking when compared to randomly generated terrain generated without user input?

1.4 Objectives

The main objective of this dissertation is to leverage Neural Networks (NN) algorithms, jointly with Procedural Content Generation (PCG) to create a generative tool that allows its users to create authored 3D terrains without any type of previous knowledge on 3D terrain creation. A NN will be used as the generative engine behind a PCG algorithm. This gives its users the opportunity to create more detailed 3D terrains when compared with industry established generative algorithms such as Perlin Noise, as well as giving the user the ability to directly influence the end result, which most broadly used generative algorithms do not allow. This notion will be achieved by comparing 3D terrains generated by the developed tool with terrains generated via a more regular algorithm, in this case, Perlin Noise. The tool will enable users from any experience level to create an authored terrain with more ease than handcrafting it, reducing the burden of development, and streamlining 3D terrain creation.

1.5 Document Structure and Organization

The remainder of this dissertation is divided in 6 chapters. Chapter 2 will address the existing body of work related with this dissertation, starting with Neural Networks, PCG and finishing with bodies of work more closely related with this dissertation. Chapter 3 will present the work that was done to be able to achieve the proposed objectives. Chapter 4 will present results regarding testing done to the work developed, as well as discussing the obtained results. Finally, in Chapter 6, the conclusions regarding this dissertation will be discussed.

Literature Review

Before proceeding, a quick overview of how DNNs work is necessary to comprehend how this promising technology accomplishes its results. We start with a simplified explanation of how DNNs operate and after that we will explore specific bodies of work more closely related to this dissertation's theme, their achievements, and contributions. Afterwards, an overview of PCGs history, concepts, tools, and taxonomies is given to better understand the research and impact this area has on the gaming industry. Finally, an overview of some PCG's legacy algorithms is presented, finishing on work that is directly related to this dissertation.

2.1. Artificial Neural Networks (ANNs)

ANNs are an information processing model and its development began in 1943, when neurophysiologist Warren McCulloch and mathematician Walter Pitts projected a binary threshold unit to translate a biological neuron into a computational model (McCulloch & Pitts, 1943) (Sarle, 1994). As we can see in Figure 2.1, this model is roughly similar to its biological counterpart, the brain (A. K. Jain, Mao, Road, & Jose, 1996).

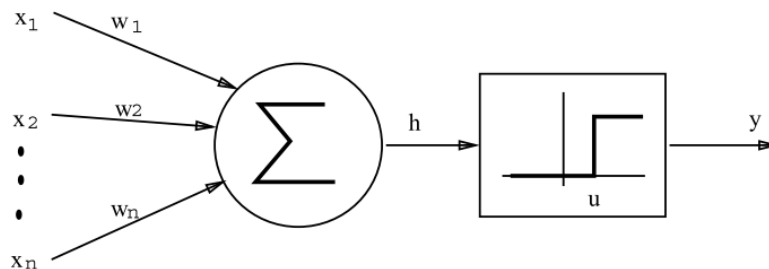


Figure 2.1 - McCulloch neuron.

This neuron receives $x_i, i \in \{1, 2, \dots, n\}$, inputs (initial inputs or inputs from a previous layer), multiplies them by the weights $w_i, i \in \{1, 2, \dots, n\}$ (that represent the connection strength), sums the results and passes them through an activation function to calculate the output (Sarle, 1994). Several of these neurons can be chained, stacked, and linked together to create a structure capable of solving problems and tasks without having to be explicitly programmed to do them, and excel at solving non-linear problems. When neurons are chained, they become an ANN which receives inputs in the input layer, generally real numbers (Nielsen, 2018). An example of an ANN can be seen in Figure 2.2.

These inputs are passed to each neuron that can receive several at the same time and computations are made accordingly to each neuron programming and its output serves as input to the next layer. Note that each connection has a weight that is altered depending on the number of times the connection is used: heavy use means higher weight and vice-versa, just like real neurons work. This is repeated until we reach an output and final layer (Nielsen, 2018).

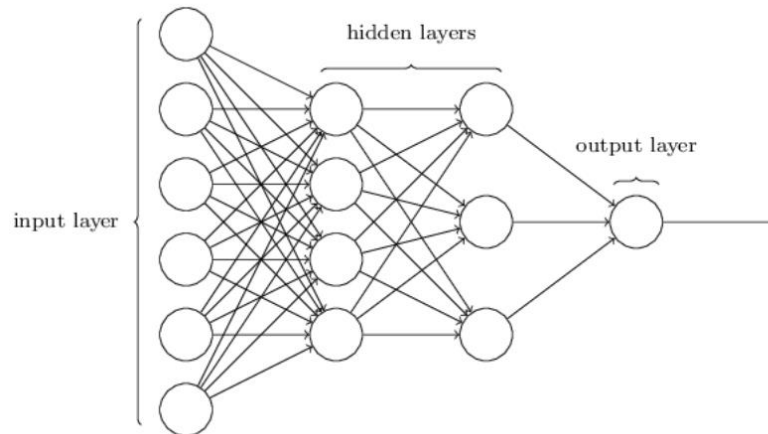


Figure 2.2 - ANN example, Taken from (Nielsen, 2018).

Several advancements were made in the field, such as the Perceptron (Rosenblatt, 1958), the representation of the threshold proposed by (McCulloch & Pitts, 1943) were converted in bias terms in (Widrow, 1960) and the rediscovery of backpropagation (Paul Werbos, 1974). Werbos et al. work played a key factor in the resurgence of interest in this area seen in the 80s because it enabled the possibility to efficiently train multi-layered networks. More complex layout pushed this area into blooming, with different neurons, different activation functions and new ways to train ANN, paving the way for an exponential growth in complexity and, consequentially, more possible applications.

2.1.1 Deep Neural Networks (DNNs)

Deep Learning (DL) was first used by (Dechter, 1986) in machine learning and applied to ANNs by (Aizenberg, 2000) to describe ANNs with several hidden layers. This type of network configuration is, operation wise, manifolds heavier than previous layouts, to such a degree that only recently they began to be attractive in real world solutions due to the steady increase of graphical processing units in GPUs, permitting that heavy parallel operations be carried out. Thanks to more computation, these networks can represent far more complex concepts and relationships between features (Nielsen, 2018).

The network is aiming at learning an optimal mapping function that best describes the probability distribution present in the given dataset. This is achieved by trying to minimize a previously defined loss function that represents the distance between the prediction made by the network and the actual label. There are several loss functions, each better suited for the type of objective (Nielsen, 2018).

With each iteration, the parameters of the network are corrected depending on the values of the loss function, towards the global minimum. All these loss values are then compounded on a cost function, also referred to as regularization that represent the global error of the network. If this value is acceptable and we have learned an optimal mapping function, the training data is no longer necessary, and we are ready to input never seen before values and compute predictions based on this unknown distribution probability (Nielsen, 2018).

2.1.2 DNN Architectures

This section will shed some light on the different kinds of neural networks available nowadays, focusing on the most notable examples.

2.1.2.1 Convolutional Neural Network (CNNs)

The most notable examples of CNNs are Krizhevsky et al. that broke the record for ImageNet Large Scale Visual Recognition Challenge by a margin of error of less than 10% comparing to the runner-up (Krizhevsky et al., 2012) and Lecun et al., responsible for the network capable of recognizing handwritten digits (Yann Lecun, L'Éon Bottou, Yoshua Bengio, 1998), creator of the MNIST dataset and whose solution is still used nowadays.

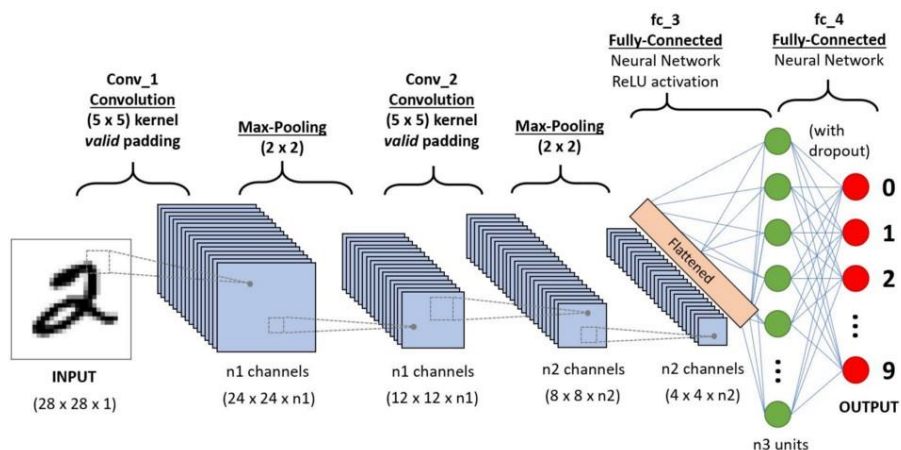


Figure 2.3 - Example of CNN (Saha, 2019).

As by the given examples, this network excels at image classification and it is composed of three main parts: the convolution layers, the pooling layers and fully connected layers, as can be observed

in Figure 2.3. The role of all these layers is to reduce the input image into a more easily processable form without losing critical features that might influence the final prediction (Saha, 2019) (Stanford, 2019).

First, the network has a convolution layer and, as the name points out, it performs convolutions over the input volume with a specified kernel size. What this does is perform a matrix multiplication between the kernel and the volume inputted, transforming the area of the kernel in a single input on the convoluted layer. For example, a 5x5x1 matrix convoluted with a 3x3x1 kernel would output a 3x3x1 feature. With this operation, high level structures, such as edges and orientation, are extracted (Stanford, 2019). After the convolution layer is the Pooling Layer. It decreases the spatial size of the convoluted layer further. This helps decreasing the computations needed to process the data and it extracts fixed dominant features (constant features that are immutable faced with rotations and positional variations). This can be achieved by using pooling techniques, such as Max Pooling, Average Pooling and Global Pooling (Stanford, 2019).

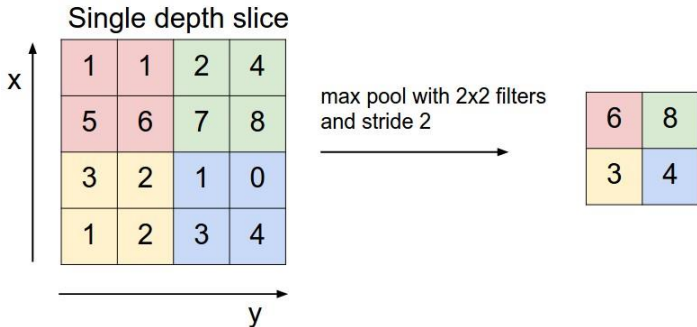


Figure 2.4 - Max pooling example (Stanford, 2019).

In the first case, a kernel size is defined and traverses the full length and width of the previously convoluted layer picking the maximum value present on that area. An example of this method can be seen on Figure 2.4, where a 2x2 kernel finds the maximum value on its 2x2, every time it moves. Average Pooling is similar, but instead of choosing the maximum value, an average between all the values is computed. Lastly, Global Pooling, that allows for entire feature map to be reduced to a single value, which helps reduce overfitting and can replace the Fully Connected layer (Lin, Chen, & Yan, 2014), which is the last layer. This layer is fully connected to the outputs of the previous layer, just like in any regular NN, and it serves as the de facto classifier of the information previously computed (Saha, 2019) (Stanford, 2019).

2.1.2.2 Recurrent Neural Networks (RNNs)

The most notable example of RNNs are Long short-term memory (LSTM) networks developed by (Hochreiter & Schmidhuber, 1997) but for simplicity sake this subchapter will address how RNNs operate and not LSTMs.

One of the differences between RNNs and other ANNs is how data is inputted. On regular ANNs, data is fed to the input layers at the same time, travels from left to right and something is outputted by the ANN. In RNNs data input is sequential, meaning that the first neuron's output is taken in consideration when the second neuron is receiving its input (Olah, 2019). This means that the previous output will influence the output of the next neuron. Additionally, each neuron has a hidden state that is updated with information that the neuron should remember, which increases the complexity of the network..

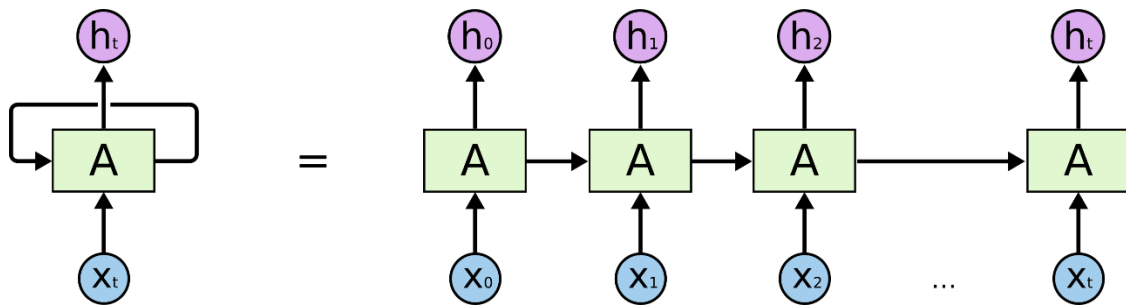


Figure 2.5 - RNNs information flow (Olah, 2019).

As we can see in Figure 2.6 this is achieved by using Gated Recurrent Units. Inside these gates, the output of the previous neuron activation value is multiplied by the sigmoid of the current data being fed to the neuron. If the sigmoid value is 0 then the previous state is remembered and if it is 1 the remembered value is updated to the current one (Olah, 2019).

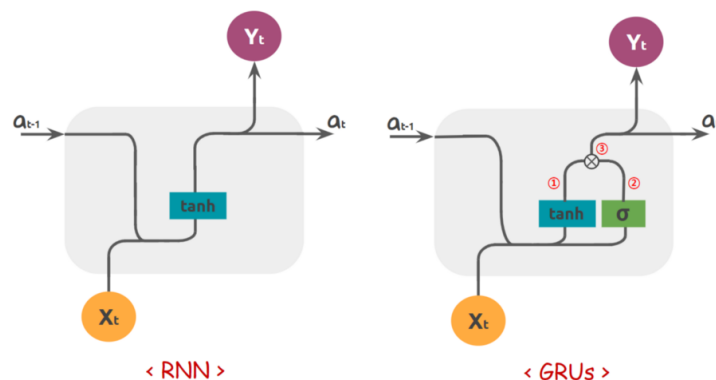


Figure 2.6 - Comparison between regular RNN and Gated Recurrent Units (Olah, 2019).

2.1.2.3 Generative Adversarial Networks (GANs)

The third DNN architecture and the most relevant one for this dissertation are GANs, because it is the architecture used in this dissertation. This type of DNN is proficient in image generation. Given a training set, this type of DNN can learn the underlying distribution and be able to generate images similar to the ones it was trained on (Salimans et al., 2016). Until Goodfellow et al. (I. J. Goodfellow et al., 2014), deep generative models did not had much impact on the ML scene, especially when compared to the discriminative models, due to the difficulty of estimating hard to control probabilistic calculations that arise in the strategies usually used in the generative context, like in maximum likelihood estimation (I. J. Goodfellow et al., 2014). Another problem that arises when generating content is how do you distinguish between the quality of two generated images. A common method is measuring the distance between the output of the ANN and the nearest dataset neighbor, e.g., BLEU score (Papineni et al. , 2002), used commonly in language translation. These methods become inefficient when the amount of data goes from a simple sentence to an image where each pixel can have several different colors, simply because of the sheer number of possible configurations even the smallest image can have. The solution is to pit two neural networks against one another: one, the Generator (G), generates the content and the other, the Discriminator (D), evaluates whether the output is generated by G or if comes from the training dataset, classifying as fake any input it assumes fake, low quality, or real (from the dataset), if it has enough quality to fool D. The general GAN architectural outline can be seen in Figure 2.7.

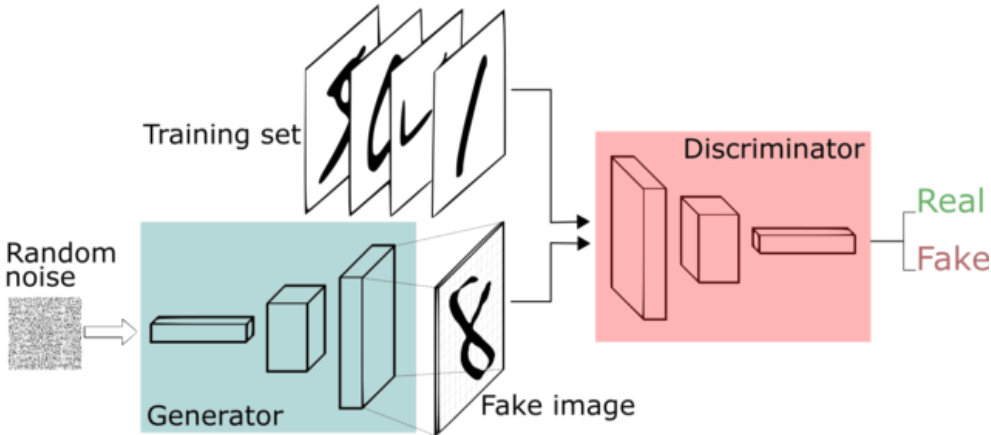


Figure 2.7 - Vanilla GAN architecture (Carey, 2019).

This model essentially works like a “Catch Me If You Can” game, a police *versus* counterfeiter scenario if you will. G generates an image, that is passes to D that tries to discern if the image comes from the training set or if it is a generated image, a fake. With each iteration, D gets better at spotting fakes but at the same time G becomes better at generating seemingly indistinguishable images when compared to the training set, until we reach a point (ideally) where D does not have more than a 50 % chance of spotting a fake and our G is ready to generate images without the need of adjustment.

Equation 2.1 represents the loss function of a vanilla GAN as seen in (I. J. Goodfellow et al., 2014). G is trying to minimize the right hand part of the equation and D is trying to maximize the left hand side of the equation which means that, fundamentally, these two networks are locked in a minimax two player game (I. J. Goodfellow et al., 2014).

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}}(\log D(x)) + \mathbb{E}_{z \sim p_{data}}(\log(1 - D(G(z)))) \tag{2.1}$$

Training is done as any other ANN, but with a key difference: G and D are updated asynchronously. Since both are trying to optimize in different directions you first need to train D with real images, then generate fakes with G so that D can classify them as real or fake and then finally, use Ds prediction to train G so it can learn how to fool D.

2.1.2.4 Types of GAN

After GANs where first proposed by Goodfellow et al. (I. J. Goodfellow et al., 2014), several advancements where made in the field, such as Batch Normalization (and all its variations) and the *All Convolutional Net* (Springenberg, Dosovitskiy, Brox, & Riedmiller, 2014), inspired (Radford et al., 2016) to create the Deep Convolutional Generative Adversarial Networks (DCGAN) architecture to achieve a more stable training and avoid the problems described in the body of work created until then.

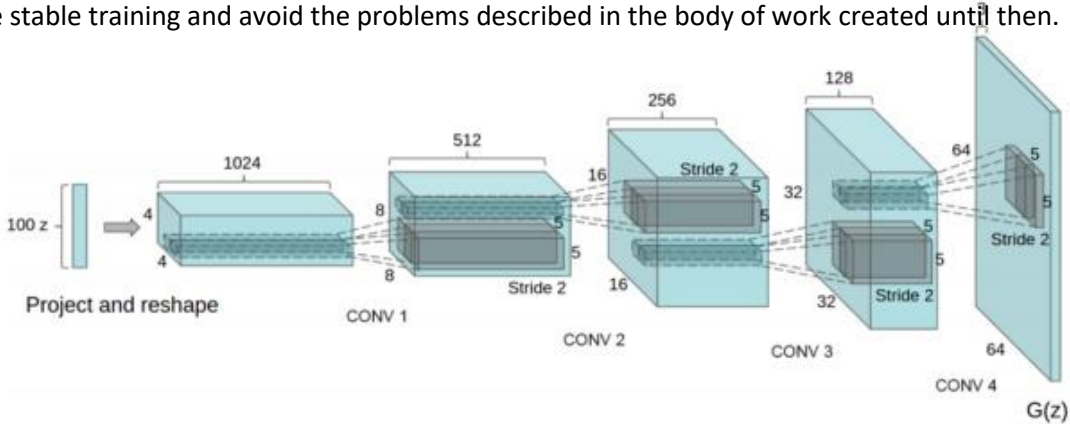


Figure 2.8 - DCGAN Generator architecture. Network input is a 100x100 noise vector (z) and its mapped to a G(z) output of 64x64x3 (Radford et al., 2016)

In regard to architecture, in DCGANs (see Figure 2.8), the Generator network (G) is a four-layer deconvolution network that has as input a vector z with one hundred random values that pass through the 4 hidden layers and end in a 64x64x3 pixel image (Radford et al., 2016). The Discriminator (D) is also a four-layer deep convolutional network that receives as input the output from G, pass it through the network, transforming it into a 4x4x512 feature map, which is then passed to a fully connect output neuron that will output either 1 for an image from the dataset or a 0 for a generated image.

In order to achieve a more stable architecture, Radford et al. replaced the pooling layers in D in lieu of strided convolutions and in G with fractional-strided convolutions, applied Batch Normalization to both G and D, removed fully connected layers in hidden layers. Radford et al., then trained this architecture with three different datasets, Large-scale Scene Understanding (LSUN), Imagenet-1k and the Faces dataset and demonstrated that a much more stable GAN architecture is possible (Radford et al., 2016).

The next relevant GAN architecture are Conditional Generative Adversarial Networks (cGAN), first introduced by Mirza et al. (Mirza & Osindero, 2014). cGAN is a simple yet powerful expansion of GAN training. Instead of feeding the GAN with x (in the case of D) and with z (in the case of G), the GAN is fed with its specific input *and* a conditional information y that can take the form of a class label or any other information that might condition the generative process.

When comparing Equation 2.2 and 2.1, it is possible to see that the loss function of the cGAN (2.2) is extremely akin to the loss function of the vanilla GAN, 2.1, with the only exception being label y been fed to both the G and D to steer the generative process in the desired direction. This difference can be seen in Figure 2.9. The architecture is similar to vanilla GANs with the key difference being the simultaneous input of an image from the dataset and a label for G and the output of G and a label for D.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}}(x) [\log D(x|y)] + \mathbb{E}_{z \sim p_{data}}(z) [\log(1 - D(G(z|y)))] \quad (2.2)$$

In their work, Mirka and Osindero (Mirza & Osindero, 2014) also showed for the possibility of multi-modal labels, further increasing the description power that labels have on generation. This opened the doors for further investigation which led to the creation of the much famous *Image-to-Image Translation with Conditional Adversarial Networks* by Isola et al. which is one of the key-works for this dissertation

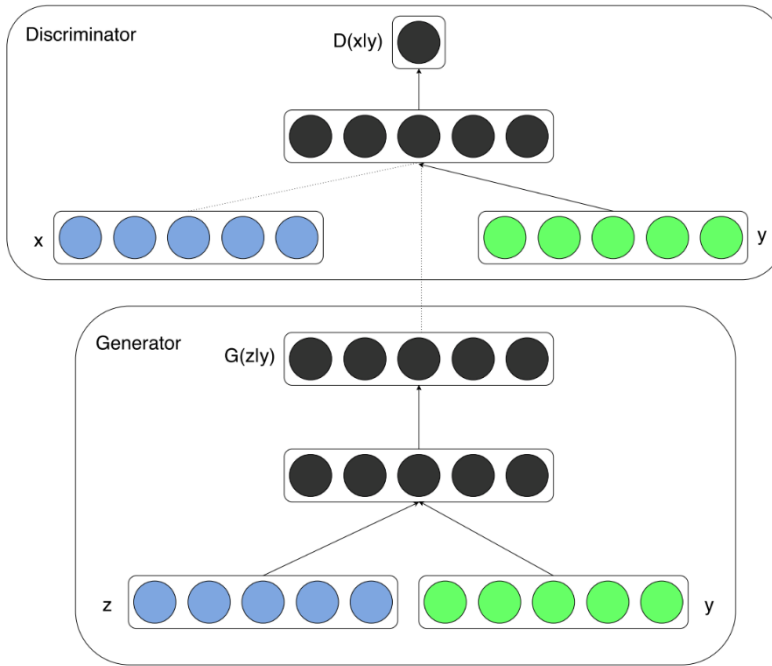


Figure 2.9 - cGAN architecture example (Mirza & Osindero, 2014).

2.1.2.5 Image-to-Image Translation by Isola et al. (pix2pix)

Isola et al. work is explored in this dissertation with more detail due to the fact that the GAN used in this work was originally developed by Isola et al. (Isola et al., 2016). Isola et al. set out to corroborate the concept that cGAN are an effective general purpose solution for image translation, which consists in translating one possible representation of an image into another, given sufficient training data (Isola et al., 2016), e.g., translate a picture in a painting of that same picture but with a specific painting style. Image translation hinges on the differentiating factor that the loss learned by a cGAN is a structured loss, a loss that takes in account the combined structure of the output, instead of treating each pixel of the output space as independent from each other. This confers cGAN the possibility of being applied to any structure (Isola et al., 2016).

The first differentiating factor of this GAN is the objective function, that instead of mixing the GAN objective with L2 regularization (the sum of the squared loss for each data point), Isola et al. mixes it with L1 regularization (sum of absolute differences between the target and predicted values), resulting in less blurry outputs.

$$G^* = \arg \min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G). \quad (2.3)$$

This resulting objective function can be seen in Equation 2.3, where $\mathcal{L}_{cGAN}(G, D)$ is the same as in Equation 2.2 and $\lambda \mathcal{L}_{L_1}(G)$ is the multiplication of a constant, λ , that can be manually set, multiplied by L1 regularization.

Another particularity of this GAN design is that the Generator (G) is not provided with noise as an input like previous GAN designs. The authors noticed that G simply learned to ignore the noise. Instead, noise is introduced in both G and the Discriminator (D) in several layers, both in training and testing in an attempt to increase output stochasticity, with weak stochasticity improvements (Isola et al., 2016).

Isola et al. also differ from vanilla GANs in the Generator and Discriminator structure which is based on the work of Radford et al. (Radford et al., 2016). The Generator architecture chosen was a *U-Net* based on (Ronneberger, Fischer, & Brox, 2015). This is a Fully Convolutional network, with a contracting side and an expansive side giving it its distinct “U” shape. Additionally, no noise is provided to the generator. This was substituted with dropout, which switches off hidden and visible units at random, in several layers of the generator, to prevent over-fitting.

The contracting side is a convolutional network with convolution layers, ReLu activation layers and pooling layers, that use max pooling. The contracting side learns feature information, then the expansive side combines these features with spatial information using up-convolutions and concatenations (Ronneberger et al., 2015) (Isola et al., 2016).

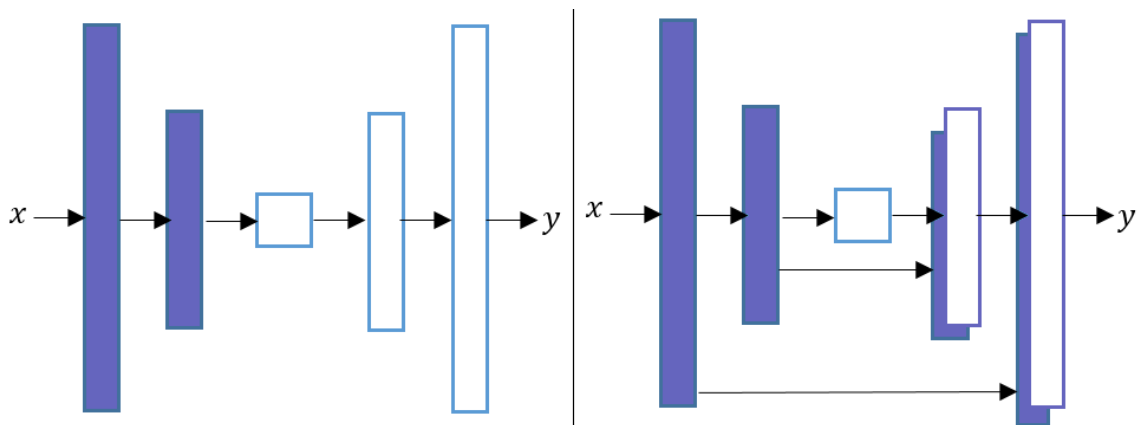


Figure 2.10 - Left - Auto-Encoder. Right - Pix2Pix U-Net (Nayak, 2019).

The rationality for this choice was that, in the context of image-to-image translation, the input and output are both high resolution grids that have the same underlying structure, making the Input/output structure be somewhat aligned. Previous works in this area used an Auto-Encoder solution, which exhibited the following problem: looking at the left network of Figure 2.10 we can see a bottleneck in the middle, because all information must invariably pass through the center

(Ronneberger et al., 2015) (Isola et al., 2016). To circumvent the problem of having a bottleneck in the middle and making use of the notion that on image translation there is a considerable volume of low level information that is shared between input and output, skips are added to the Unet, between layers so this information can jump between layers and free up space on the center for information that needs to invariably flow through the center. Layers where skipping is possible, perform a concatenation of channels from layer i with channels from layer $n-i$ (Isola et al., 2016).

As for the Discriminator of this GAN, a CNN (Isola, 2017) is used, named *PatchGAN*, whose architecture is seen in Figure 2.11. By using L1 regularization, the network is already being encouraged to accurately assess low frequency features so D needs to be restricted to only model high frequency structures (Isola et al., 2016). To achieve this, *PatchGAN* only penalizes high frequency structures at the level of individual patches. Instead of mapping from the full input to a single scalar output that dictates whether the image is fake or real, D traverses the input images and classifies each $N \times N$ patch of the image as real or not and then all results are averaged to produce the final output. This makes D smaller, faster, requires less parameters and can be applied to any size of image because its evaluating high frequency structures at a $N \times N$ patch scale rather than considering the whole image as the high level structure (Li & Wand, 2016) (Isola et al., 2016).

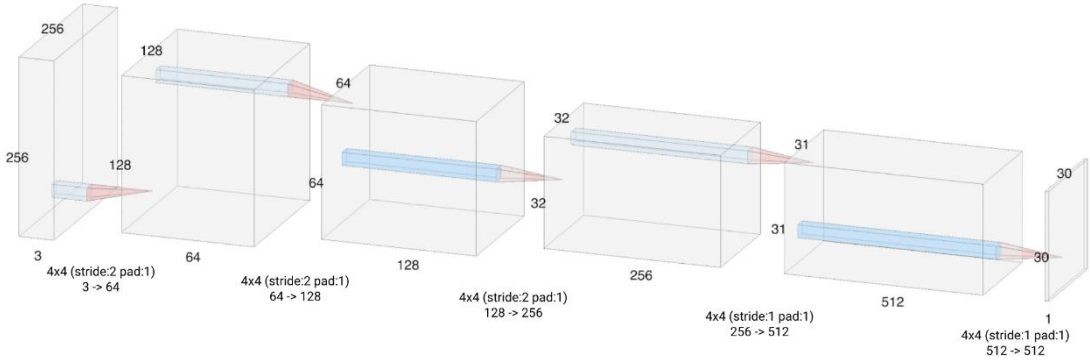


Figure 2.11 - PatchGAN Discriminator, and its patch-by-patch evaluation (Ku, 2019).

2.2 Procedural Content Generation Definition

The definition of Procedural Content Generation (PCG) in the context of computer games has been subject to change in scope rather than in its definition itself, mostly because of a lack of agreement between practitioners regarding what is considered content that should be generated by PCG algorithms (Togelius, Kastbjerg, Schedl, & Yannakakis, 2011).

Togelius et al. define PCG “as the algorithmical creation of game content with limited or indirect user input.” (Togelius, Kastbjerg, et al., 2011) and go a step further by contrasting what PCG is, what is not and what might be, defining that, for example, PCG is not something created by the user, even if it is via procedural methods (Togelius, Kastbjerg, et al., 2011).

Later, Hendriks et al. defined PCG as “the application of computers to generate game content, distinguish interesting instances among the ones generated, and select entertaining instances on behalf of the players” (Hendriks et al., 2013b), seeing PCG as technology suitable to replace hand-crafted content design, but making clear that a degree of parameterization is required so that content can be influenced by the designer’s needs (Hendriks et al., 2013b). Hendriks et al. are also responsible for coining the term Procedural Content Generation for Games (PCG-G), which further departs PCG-G from PCG methods employed in other fields, such as simulation and movie animation (Freiknecht & Effelsberg, 2017).

Lastly, Shaker et al., added to the definition provided by Togelius et al. a consideration that the content generated can be automatically generated or in conjunction with the designer. Furthermore, Shaker et al. specify their definition of content as including almost everything that is present in a game, excluding the game engine and non-player character behavior, which they do not consider to be content, although even this is a point of contention, if we take in consideration promising behavioral generation (Barreto, Cardoso, & Roque, 2014).

2.2.1 Brief History of PCG in Commercial Games

The earliest recorded commercial use case of PCG was *Beneath Apple Manor* by Don D. Worth, which dates to 1978. Developed for Apple II personal computer, it used procedural generation to randomize its dungeons, featuring 5 to 10 different rooms each, and all of this on only 16 KB of memory (Risi & Togelius, 2019) (Torrado et al., 2019) (Freiknecht & Effelsberg, 2017).

In 1980, *Rogue* was released to much critical acclaim, so much so that it spawned its own subgenre, Roguelike games. Similar to *Beneath Apple Manor*, *Rogue*’s dungeons were procedurally generated but with an added feature, permanent death. *Rogue* spawned several free-to-play variants such as *NetHack*, *Moria*, *Ancient Domains of Mystery (ADOM)*, *Angband* and others (Hendriks et al., 2013b) (N. Shaker et al., 2016). Commercially speaking, *Rogue* inspired the critically acclaimed dungeon crawler series, *Diablo*, with its third installment, *Diablo III*, released in 2012. Another paradigm setting game was 1984s *Elite*, a space exploration and trading game with fully procedurally generated galactic composition, economy and galaxy composition: eight galaxies in total, each with

256 unique planets (N. Shaker et al., 2016), everything fitting in 64 kilobytes of memory (Risi & Togelius, 2019). Its developers were able to achieve this by storing the seeds of each system that, upon entry, were then fed to a Pseudo-Random Number Generator (PRNG) (Hendrikx et al., 2013b) (Risi & Togelius, 2019). Just like *Rogue, Elite* had a strong impact that led to not only sequels, 1993s *Frontier: Elite II* and 1995s *Frontier: First Encounter*, but also spiritual successors such as *Elite: Dangerous*, a crowdfunded game that released in 2014, and inspiration for other games, such as *No Man's Sky* (Risi & Togelius, 2019) and *EVE Online* (Hendrikx et al., 2013b).

All these games have one thing in common: none of them implemented PCG solely for its intrinsic value. Aside from replayability, all of them implemented PCG to resolve memory issues due to the sheer scope of the games (Freiknecht & Effelsberg, 2017) (Risi & Togelius, 2019). This tendency, nowadays, has been mostly shifted. With growing hardware capabilities and the increase in budget, PCG moved from a technology to overcome hardware restrictions to a means to add replay value without bank breaking additional costs and to a technology that reduces development time (Barreto et al., 2014).

As these technologies and methods matured, their application and usage went from a way to circumvent hardware bottlenecks to a purposeful implementation as a game element. Early high-profile examples include *X-Com: UFO Defense*, released in 1994, *Diablo*, *The Elder Scrolls II: Daggerfall*, both released in 1996 and *Command and Conquer: Red Alert 2*, released in 2000. All these game series are still in existence and sequels are still being released (Freiknecht & Effelsberg, 2017).

More recent examples include the previously mentioned *EVE Online*, a massively online game from 2003 that generates planets visuals, universe and scenarios akin to *Elite*, resorting to modeling and simulation of complex systems for universe generation and PRNG to place hand crafted elements (Hendrikx et al., 2013b). *Minecraft*, released in 2009, uses PCG techniques to generate its world (N. Shaker et al., 2016), more specifically, scaled 3D Perlin noise with linear interpolation (Persson, 2011) to create structures and a Whittaker Diagram (Whittaker, 1962) to populate the world with diverse biomes ("Biome", n.d.).

Another recent game series that generates game space procedurally, is *Civilization*, which uses similar techniques to *EVE Online* to generate resources and place them on the generated map (Hendrikx et al., 2013b). In 2011, *Elder Scrolls V: Skyrim* was released and contained a scenario generation system called Radiant ("Skyrim:Radiant - The Unofficial Elder Scrolls Pages (UESP)", n.d.), which created quest based of templates that are completed according to existing game conditions (Hendrikx et al., 2013b). Even more recent, and more heavily centered on PCG, is *No Man's Sky*. Released in 2016, it features a procedurally generated deterministic universe, with over 18 quintillion planets, each with generated ecosystems with distinctive forms of flora and fauna, of which some is sentient and can be engaged by player for various purposes, including combat, trade, and linguistic training (Cook, 2016).

2.2.2 PCG Tools

Just like there are PCG solutions for games there are also some tools for asset creation that are worth mentioning. The first mention is SpeedTree ("SpeedTree - 3D Vegetation Modeling and Middleware", n.d. First released in 2002, it is the most widely used PCG middleware in game development (Togelius et al., 2013) and its purpose is to create organic vegetation, from bushes to trees. It achieves this by treating the tree as an interactive generative material, allowing the designer to deform the tree as one sees fit without it ever disrupting the tree's integrity (N. Shaker et al., 2016). SpeedTree also made the jump from a game only middleware, to become available for cinema and television productions since 2009 ("SpeedTree - 3D Vegetation Modeling and Middleware", n.d.). On the same generative subject, there is also Xfrog (Xfrog, n.d.), also with a generative approach to create plants. First released in 1996 and, nowadays, available for Maya and CINEMA 4D.

Texture generation tools can also be augmented by procedural methods. One early example is *DarkTree*, "an advanced procedural shader authoring tool" ("Darkling Simulation's", n.d.). This tool allows the user to connect several small procedural algorithms to create a new final texture (Roden & Parberry, 2004). *Blender* also offers the possibility of generating textures ("Procedural Textures — Blender Manual", 2017). Another texture generator is the *Substance Suite* of products ("Substance | The leading software solution for 3D digital materials", n.d.), with *Painter* being geared towards real-time texturing and *Designer* focused on texture creation. Both have tools at the user's disposal to paint and create textures using procedural methods ("The Substance Art of Rogelio Olguin", 2016) ("Substance Designer", 2017).

L3DT is a terrain generator that also generates textures ("L3DT", 2017). It uses Perlin noise or Fractal algorithms for the map creation and allows parameter manipulation ("L3DT documentation", 2017). It is also capable of placing water on the map, calculating surface normal, light maps, and others. CityEngine aims at generating several different buildings automatically through procedural modeling from a limited rule set or user created content ("Advanced 3D City Design Software | Esri CityEngine", n.d.) (Hendriks et al., 2013b).

Moving to gameplay oriented tools, the first example is Sentient Sketchbook (Liapis, Yannakakis, & Togelius, 2013), a real-time game level generator that produces levels through designer sketches (Barreto et al., 2014). This generator presents to the user several alternatives possible, taking into account the current design, tests created levels automatically and displays navigable paths (N. Shaker et al., 2016).

Ropossum (M. Shaker, Shaker, & Togelius, 2013) is also another authoring tool that generates and tests levels for Cut the Rope, a physics-based puzzle game. This tool allows for automatic generation of solvable levels, the ability to merge user's designs with automatically generated ones, as well as checking for playability and optimizing designs accordingly (N. Shaker et al., 2016).

Tanagra (Smith, Whitehead, & Mateas, 2011) is a tool for level design that allows for the designer to rapidly view several possible levels that meet a set of specified constraints. A reactive level generator guarantees that all created levels are playable. Moreover, designers can refine created levels by moving level geometry as intended, based on their own intuition or based on tool's suggestions (Barreto et al., 2014) (Smith et al., 2011). Tanagra levels fit a theory of "rhythmic patterns in platformer games" (N. Shaker et al., 2016), which can also be modified by the designer, leading to a recalculation of the portions that are affected by the changes.

A final example is *SketchaWorld* (Smelik, Tutenel, De Kraker, & Bidarra, 2010), a PCG tool that allows the user to create a fully integrated virtual world, from topology of the landscape to positioning of vegetation, roads, building, and water courses. The user interactively sketches high level terrain features in a 2D layout map, which is a coloring grid that contains elevation and soil material information. Then, on a second stage, the user adds information about urbanization and river features (N. Shaker et al., 2016). The final representation is 3D and for this to be possible the tool possesses blending and local adaptation steps to ensure logic placement of features, such as removing a misplaced tree from a generated road (N. Shaker et al., 2016).

2.2.3 PCG Taxonomies

Procedurally generated game content has been classified and categorized with different approaches, mainly by Hendriks et al. (Hendriks et al., 2013b) and by Togelius et al. (Togelius, Yannakakis, Stanley, & Browne, 2011), and later expanded by Shaker et al. (N. Shaker et al., 2016). In the first work, the approach was to classify according to what content can be generated, while the latter categorizes techniques according to morphology and application (Barreto et al., 2014).

Hendriks et al. (Hendriks et al., 2013b) divided procedurally generated game content in six classes, each with a subdivision that represents the specific type of content that can be generated in each class. The first class, *Game Bits*, includes the most basic structure present in games. When considered individually, separated from their over-arching composition, game bits typically do not interact with the user (Hendriks et al., 2013b). There are six types of game bits: textures, sound, vegetation, buildings, behavior, and graphical effects, such as fire, water, stone and clouds (Hendriks et al., 2013b) (Barreto et al., 2014) (Barriga, 2019).

The second class, *Game Space*, refers to the virtual spaces in which the game will unfold. These spaces are filled with game bits and they play a big role in immersing the user and elicit personal interpretations regarding the game (Hendriks et al., 2013b). Game space is further divided in Indoor Maps, Outdoor Maps, and Bodies of Water (Hendriks et al., 2013b). This distinction is made because, aesthetically and thematically speaking, outdoor maps are more consistent, which is not observed with indoor maps, that can be heavily influenced by the cultural setting. Implementation wise, the algorithms used to generate an indoor location are completely different from outdoor maps (Hendriks et al., 2013b) (Barreto et al., 2014)(Barriga, 2019). If evaluated according to this categorization, this dissertation will produce an artifact that falls in this class, more specifically, an Outdoor Map.

The third class are *Game Systems*. This classification class refers to complex systems theory and modeling that simulate and generate complex environments that grant games more believability and appeal (Hendriks et al., 2013b) (Barreto et al., 2014). These game systems can be ecosystems, road networks, urban Environments, and entity behavior. The latter mainly refers to Non-Playable Characters (NPC) (Hendriks et al., 2013b).

The fourth class, *Game Scenarios*, pertains to story and content related to how the game unfolds, its logical progression (Hendriks et al., 2013b). Its subcategories are Puzzles, Storyboard, Story, and Levels (Hendriks et al., 2013).

The fifth class, *Game Design*, refers to what can be done, what the player is trying to achieve, that is, rules and goals. It also includes rules regarding game design, i.e., overarching rules regarding thematic and general setting and tone (Hendrikx et al., 2013b). This class is subdivided into *System Design* and *World Design*. Lastly, the sixth class, *Derived Content*, is defined as “content that is created as a side-product of the game world” (Hendrikx et al., 2013b). This means generating auxiliary content that can be described as “meta content”, which can be used outside or inside the game itself, helping the user to further immerse themselves in the game. The two types of derived content are “News and Broadcasts” and “Leaderboards”.

As mentioned, Togelius et al. (Togelius, Yannakakis, et al., 2011) classifies procedurally generated game content based on morphology and application, initially creating a five-group taxonomy, which Shaker et al. (N. Shaker et al., 2016) later expanded to a seven-group taxonomy. The first group is *Online versus Offline*. Game content can be generated online (in-game), while the game is being played, allowing extended replayability and the possibility of creating content that is adapted to the player or it can be generated offline (off-game), either during development or before the user starts a new session (N. Shaker et al., 2016) (Togelius, Yannakakis, et al., 2011).

The second group refers to *Necessary versus Optional* content. The main distinction being that necessary content is mandatory for the completion of the game and, thus, it needs to always be correct and functional. On the other hand, optional content, being content that the player does not need to obligatorily interact with to complete the game and has the choice to ignore, does not need to be always correct (Togelius, Yannakakis, et al., 2011). An example of necessary content is a *Diablo* dungeon, which needs to be traversable in its entirety to be beatable. An example of optional content is a *No Man’s Sky* planet generated with aggressive weather conditions and fauna, so much so that it renders any gameplay on that planet impossible. However, since there are many planets on the game, the player can simply ignore it and go to the next planet.

The third group pertains to *Degree and Dimensions of Control*, also known as *Random Seeds versus Parameter Vectors*. Another difference regarding procedural content generation according to Togelius et al. and Shaker et al. is to what degree the algorithm can be parameterized to allow varying degrees of control. Considering control as a spectrum, on one end we have Seed Generation, which allows for a specific generation to be regenerated, but it does not allow for its parameterization (“Seed (level generation)”, n.d.), as in *Minecraft*. On the other end of the spectrum, we have any generator that accepts as input a vector with several parameters to control the unfolding of the generative process (N. Shaker et al., 2016)(Togelius, Yannakakis, et al., 2011).

Fourthly, the *Generic versus Adaptive* group refers to content that is generated without taking into account user behavior or proclivities. This content generation method is diametrically opposed to adaptive generation, in which user interaction is analyzed and taken into account when generating new content (N. Shaker et al., 2016). The fifth group considers *Stochastic Versus Deterministic Generation*. As the names imply, in deterministic generation, given the same initialization point, it is possible to regenerate the same content, as opposed to stochastic generation, in which the exact same content is not possible to regenerate (N. Shaker et al., 2016). An example of deterministic generation would be *Minecraft* seed generation. In some cases a totally deterministic algorithm can be understood as a form of data compression, as in *.kkrieger* (Togelius, Yannakakis, et al., 2011) or in *No Man's Sky*.

The sixth group regards *Constructive versus Generate-and-Test*. Constructive PCG refers to content that is generated once, without any form of verification regarding input parameters or playability of what was generated. To mitigate creating content that is broken or unplayable, the algorithm only performs operations that are proven to only generate content that is playable and desirable (Togelius, Yannakakis, et al., 2011). For generate-and-test techniques, a loop between generating and testing is formed. Once a contender is generated, it is tested according to some previously determined criteria so as to assure that the produced content is playable (Togelius, Yannakakis, et al., 2011).

Finally, the last group compares *Automatic Generation versus Mixed Authorship*. As the name implies, automatic generation allows for no or very limited user input regarding generation parameters. However, a bigger involvement of the designer is often desirable. Mixed authorship emphasizes the participation of a human designer that cooperates with the algorithm in order to fine-tune and modify the generated content, allowing for a more personalized outcome (N. Shaker et al., 2016). Examples of such tools and algorithms are presented in Section 2.2.2. The tool developed in this dissertation falls on the mixed initiative categorization.

Even though these taxonomic classifications focus on different aspects of the generative algorithms, one according to what content can be generated and the other according to morphology and application, they are not mutually exclusive and in fact they complement each other (Barreto et al., 2014). This taxonomic crossover can be seen in Table 2.1, where this dissertation's proposed tool is classified according to the taxonomic classes proposed by Togelius et al. and Hendrikx et al. To start, this tool creates a terrain of the game space type, more specifically, an outdoor map. This terrain then

falls, according to Togelius et al. taxonomic classification, in several taxonomic groups. It falls in the offline category because it is created before the game starts. It is also a necessary asset because a terrain is central to any game, without it there would be nowhere for the player to go. Following that, it falls in the generic category because there are no adaptive capabilities in the tool. The tool does not take in consideration player behavior. It is also deterministic because the same input generates the same output every time. To finish, the proposed tool also belongs to the mixed authorship group, as the tool's user is able to heavily influence the generative process.

	Game Space
Online versus Offline	Offline
Necessary versus Optional	Necessary
Random Seeds versus Parameter Vectors	N/A
Generic versus Adaptive	Generic
Stochastic Versus Deterministic Generation	Deterministic
Constructive versus Generate-and-Test	N/A
Automatic Generation versus Mixed Authorship	Mixed Authorship

Table 2.1 - Taxonomic crossover table for dissertation tool classification.

2.2.4 PCG Methods and Approaches

Similar to the taxonomies present in Section 2.6, there are also taxonomies that group the methods used to generate content by algorithms or by approach. One such taxonomy was proposed by Hendriks et al., (Hendriks et al., 2013b) that separates generative techniques in six different groups with the goal of identifying fundamental methods that would be core to a generic content generator. They are as follows: Pseudo-Random Number Generators (PRNG), Generative Grammars, Image Filtering, Spatial Algorithms, Modeling and Simulation of Complex Systems and Artificial Intelligence (Hendriks et al., 2013b). Likewise, Gillian Smith (Smith, 2014) also categorized PCG techniques, focusing more on algorithms and approaches that allow for the generation of content with which the player will interact with as part of the gameplay, focusing more on what a level designer would create rather than a writer or a game designer. The defined categories are: Simulation Based, Constructionist, Grammars, Optimization, and Constraint Driven.

The following sections present the key PCG methods, organized according to the simplified categorization methods presented in (Risi & Togelius, 2019) and (Barriga, 2019), in which PCG methods

are divided in three major groups: Traditional methods, Search-Based methods, and ML methods.

2.2.4.1 Traditional PCG methods

Generally speaking, this group of methods is the most commonly found in games and are of constructive nature, meaning these methods execute on fixed time and perform a singular iteration without any kind of search regarding the validity of its output (Risi & Togelius, 2019). The first example of traditional methods are Pseudo-random number generators, which are characterized by being initialized by a seed, a pseudo-random value, and outputting a sequence of numbers that appear random to the observer.

One of the most widely used algorithmic classes and notable examples of PRNG, are noise-based functions, which are especially convenient due to their apparent randomness and lack of structure when detail without structure is needed (Lagae et al., 2010). Inside this family, one of the most prominent examples is Perlin Noise, developed by Ken Perlin in 1985 (Perlin, 1985). Both noise function and Perlin Noise will be further analyzed in Section 2.3.1 and Section 2.3.1.1, respectively.

Another noteworthy mention is the Midpoint Displacement algorithm by (Fournier et al., 1982). Simply put, the algorithm starts with a square array, whose four corners are attributed a random value. Then, a midpoint is set for each edge existent in the grid with an average value of the two containing edge points plus a random value. Afterwards, a midpoint is set between the four obtained midpoints, using the average of the previous four points calculated plus a randomized value (Fournier et al., 1982).

The second group of traditional methods are Generative Grammars, a class of algorithms that originated from Noam Chomsky's study of languages (Hendrikx, Meijer, Van Der Velden, & Iosup, 2013a). These methods are composed of sets of rules that operate on finite pools of words and are only able to generate grammatically correct sentences. Generative Grammars, for the purpose of procedural generation, differ from the regular linguist Grammars in that, instead of using words, they use terminal nodes and shapes that are encoded as letters or words, giving the designer a visual language to model the Grammars (Fêo, Santos, & Santana, 2017) (Van Der Linden, Lopes, & Bidarra, 2014). The most predominant examples of Generative Grammars are L-Systems and Graph Grammars (N. Shaker et al., 2016). The first method is used to generate branching system such as vegetation or cave systems (Risi & Togelius, 2019), whereas the latter is more appropriate to generate more complex and non-linear game spaces and missions (N. Shaker et al., 2016).

As last example, Simulation of Complex Systems, which entails imitating, in the case of Complex Systems, a set of large interconnected elements/individuals that interact with the environment and between themselves in a nonlinear fashion, making it difficult to ascertain properties of the whole system at a glance (Aziza, Borgi, Zgaya, & Guinhouya, 2016). Due to the impracticality of predicting the global behavior of this kind of systems with mathematical equations, simulations can be run to understand and predict the relations between the individual sums of the system. One predominant example is Cellular Automata, a discrete and abstract computational system, in which a group of cells on a grid, bound to the same set of rules, evolve in accord to the rules and based on states found on neighboring cells at discrete time steps (Berto & Tagliabue, 2017) (Hendrikx et al., 2013a).

2.2.4.2 Search-Based PCG methods

Search-Based methods are stochastic search/optimization algorithms that fall under the category of generate-and-test methods, typically comprised of three components: a search algorithm, a content representation and an evaluation function (N. Shaker et al., 2016). They are used for searching desired content given a content representation and instead of accepting or rejecting a candidate, a separate evaluation function scores candidates and, until an evaluation criteria is met, content will keep being generated (Risi & Togelius, 2019) (Barriga, 2019). Newly generated candidates are always conditioned by the previously generated content with the higher evaluation score, in an effort to generate content that will have an higher score than the previous candidate (Togelius, Yannakakis, et al., 2011). The search algorithm is responsible for the generation of the content and normally an evolutionary algorithm is used (Togelius, Yannakakis, et al., 2011), although other metaheuristics are possible (N. Shaker et al., 2016). The content representation is what is going to be generated, consequently defining and limiting what content can be generated (N. Shaker et al., 2016). Finally, we have one or more evaluation functions, each measuring different metrics, such playability and aesthetic appeal. These functions produce a score representing the quality of the generated content (N. Shaker et al., 2016).

2.2.4.3 Machine Learning methods

The resurgence of Neural Networks and the use of Machine Learning for model training based on big data datasets (Krizhevsky et al., 2012), has led to an enormous increase in use and capabilities of these methods, leading to an increase in its application in several areas such as music, speech recognition, and image processing (Summerville et al., 2018), and also for content generation (Summerville et al., 2018).

Driven by the increasing importance PCG is having in aiding developers creating new and increasingly higher quality content, either with human participation or not, new generation paradigms are being explored and one of them is Procedural Content Generation via Machine Learning (PCGML). The term was coined by Summerville et al. “the generation of game content using machine learning models trained on existing content.” (Summerville et al., 2018). In other words, PCGML is an approach for regulating content generation through preexisting examples, supplied by artists and designers, which are then used to train statistical models (Karth & Smith, 2019).

PCGML sets itself apart from search-based algorithms in the way content is generated. Search-based algorithms generate content by searching content space and afterwards subjecting the candidate content to an evaluation process. In PCGML solutions, the content is generated directly from the learned model, the output of a PCGML algorithm is the content itself (Summerville et al., 2018).

Examples of PCGML applications include platformer game level generation through Long Short-Term Memory recurrent neural networks (LSTMs) (Summerville & Mateas, 2016), Autoencoders (R. Jain, Isaksen, Holmgård, & Togelius, 2016) and Markov models (Snodgrass & Ontañón, 2017). The development of Generative Adversarial Networks (GAN) (I. Goodfellow, 2016) has also been employed in various previous work, in which the relation between two images was learned (Isola et al., 2016) and digital images were generated and completed (Mirza & Osindero, 2014).

2.6 Previous and Related Work in Procedural Terrain Generation

This section addresses and explains relevant previous work in terrain generation, as well as it addresses more closely related and relevant work for this dissertation.

2.3.1 Recursive Subdivision

Recursive Subdivision is a class of fractal algorithms (J. P. Lewis, 1987) in which a geometric object is recursively subdivided based on a previously defined norm until the target criteria is met (E. Catmull, 1978). This family of algorithms was previously known as “Fractal Subdivision” (J. P. Lewis, 1987)(Fournier et al., 1982).

The diamond-square algorithm was an improvement over the earlier Midpoint Displacement (see Section 2.2.4.1). Developed by (Miller, 1986), it sets out to solve the directional artifacts found in the original work of Fournier et al. that stem from using two points to calculate the edge midpoints and four to calculate the center point (Archer, 2011).

Four corners are initialized just like in Midpoint Displacement, then a midpoint is set for each square existent in the grid with an average value of the four corner points plus a random value, the diamond step. Then, for each diamond that is in the array, a midpoint of the diamond is set with a value equal to the average of the four points that make up the diamond plus a randomized value, the square step (Miller, 1986) (Archer, 2011).

Although this algorithm is attributed to Fournier et al. and the algorithmic class majorly attributed to Catmull et al., there is one historical figure that in an ad-hoc fashion derived the midpoint displacement method: Archimedes. He calculated the area between a parabola and a chord AB (Heinz-Barnsley, M. F. et al. , 1988).

2.3.2 Noise

Noise algorithms are repeatable pseudo-random functions of their input, which are n -dimensional. Being pseudo-random, given the same input, the generated noise level is always the same (Ebert et al., 2003). Ideally, procedural noise is continuous, multi-resolution, meaning being able to create samples at any resolution, non-periodic, parametrizable, thus allowing for the generation of several different noise patterns and randomly accessible, allowing for the evaluation of any given point in constant time and without being depended of previous evaluations (Lagae et al., 2010). When addressing non-periodicity, it is noted that noise is in fact periodic but on a scale so large it is not evident when applying it (Ebert et al., 2003).

When it comes to heightmap generation, noise functions are regarded as the workhorse of methods (Hyttinen, 2017). Several methods have been created, each addressing different challenges, which can be combined and differently parametrized allowing for a vast amount of distinct results.

One of the most notable examples of PRNG, and arguably one of the most famous and more widely used noise function, is Perlin Noise, a lattice based gradient noise algorithm that instead of generating random values for each corner of the lattice's cells, generates a gradient which is, in turn, interpolated. For this algorithm to return a real value, instead of gradient vectors, given a point, directions are calculated between each corner of the lattice's cell and the point being sampled. As a last step, a dot product is calculated between the gradient at the corner of the cell and a directional vector from the corner to the desired point (Perlin, 1985). This results in smoother changes in the noise field, when compared to non-gradient based solutions, making the end result look more organic and the whole method is more efficient in term of memory and computation due to the use of lattices (N. Shaker et

al., 2016). On top of this, it is possible to combine multiple layers of Perlin noise and manipulate layer features in order to obtain more intricate results (Hendriks et al., 2013a).

2.3.3 Evolutionary Algorithms

Evolutionary algorithms (EA) are a subset of Evolutionary Computation and bio-inspired computing methods and, thus, a branch of search based methods (Raffe, Zambetta, & Li, 2012)(Vikhar, 2017). This algorithmic group draws inspiration from nature and solves problems utilizing processes similarly to how living organisms behave and evolve. They are heavily inspired on Darwinian Evolution and, just like in Darwinian Evolution, fitter solutions tend to proliferate and pass their characteristics to the next solution generation, while unfit options tend to be discarded (Coello, 2005).

Roughly speaking, EAs have four overall steps, which can be seen in Figure 2.12, as well as the logic recursive flow of the algorithm. Initialization is where the initial population is generated, randomly or seeded to areas of more optimal solution presence's likelihood. Then, a selection is made, or in other words, a fitness function is evaluated, and the best solutions are selected to reproduce a new generation. Thirdly, crossover and mutation operators are applied to pick parent solutions for subsequent breeding of new solutions exhibiting shared characteristics from the parent solutions. Finally, when a pre-determined condition being met, the selection-reproduction loop is terminated (Coello, 2005) (Eiben & Smith, 2003).

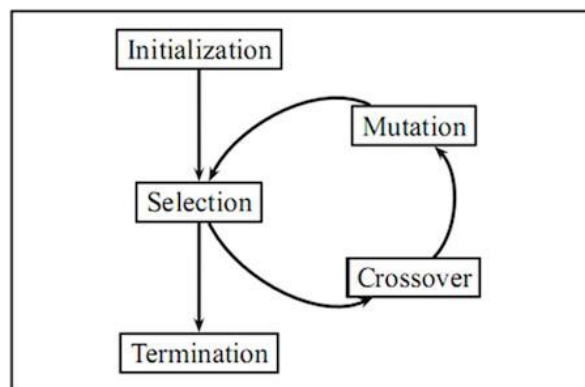


Figure 2.12 - EA logic flow (Soni, 2020).

As previously stated, EAs may employ a single fitness function responsible for the selection of solutions that will pass their characteristics to the next generation. But EAs can be extended to have multiple fitness functions, each with its own attributed weight, allowing for a set of optimal points to be found, called the Pareto frontier. These Pareto optimal solutions are the best possible solutions given the fitness functions and the characteristics of the solutions presented on the current generation. Meaning that outside the pareto front there are no other solutions that are equal or better

in any of the dimensions. These solutions can then be plotted to a graph for visual or automated evaluation of the solutions, allowing for undesired tradeoffs to be easily spotted and vice-versa (Togelius, Yannakakis, et al., 2011). This greatly enhances the authoring possibilities of content (Togelius, Preuss, Beume, et al., 2010) (Togelius, Preuss, & Yannakakis, 2010).

Togelius et al. proposes the use of a MOEA approach, more specifically the SMS-EMOA (Togelius, Yannakakis, et al., 2011), to create maps for strategy games. The overall goal is to create a playable map, which includes the placement of gameplay elements and objectives. These elements have a (x,y) direct representation on the map (Togelius, Preuss, Beume, et al., 2010). The terrain itself is a sub-component of the global goal and it is created from a flat heightmap where “representation, positions, Standard Deviation (SD) and heights of several two-dimensional Gaussians are evolved” (Togelius, Yannakakis, et al., 2011), and subsequently the terrain height is calculated based on these parameters (Togelius, Preuss, & Yannakakis, 2010).

2.3.4 Procedural Brushes

Carpentier et al. introduced the concept of Procedural Brushes, where the user sculpts directly on a 3D height field by using brushes that apply traditional operations in addition to GPU-based operations that generate various different types of noise (e.g., Perlin noise, directional noise, and erosion noise), allowing for local control for detailed sculpting and algorithmic generation of larger features (De Carpentier & Bidarra, 2009).

Procedural Brushes achieve their real time performance by employing procedural synthesis algorithms. Instead of simulating natural processes, they approximate the fractal-like semi-random patterns found in natural processes, allowing for a much faster generation since they do not need several iterations as it would be required in a simulation scenario (De Carpentier & Bidarra, 2009).

2.3.5 Software Agents

Software Agents are computer programs that work in a dynamic environment, cooperatively or competitively, towards a goal, on the behalf of another entity, without direct control and supervision. These entities display flexibility, and some creativity, when making decisions that are based on data obtained from other agents and the environment (Nareyek, 2001).

Doran et al. (Doran & Parberry, 2010) employed software agents that, given a featureless terrain, are able to work in tandem to create a visually appealing terrain. Five types of agents were included, each assigned to a different task. A *coastline agent* is the first agent to start working, and depending on the size of the terrain, can multiply itself to accelerate the task. As the name implies, it creates the

outline of the terrain. *Smoothing agents* are responsible for averaging out heights. *Beach agents* create flat areas near coastlines. *Mountain agents* create mountain chains and, finally, *river agents* erode the terrain to carve out rivers (Doran & Parberry, 2010) (N. Shaker et al., 2016). According to the authors, this approach is more controllable than fractal generation, and it is highly parameterizable: the *mountain agent* alone has twelve possible parameters (Doran & Parberry, 2010) (Frade, de Vega, & Cotta, 2012).

2.3.6 Terrain Generation using Generative Adversarial Networks

As stated in Section 1.2 and Section 2.1.5, GANs have been shown to be very proficient in image related workloads, including terrain generation. In this section we will approach other previous works in the terrain generation field, all of them using real world data as their primary source of data for training the neural networks, with the main difference between them being the amount of authoring possible.

2.3.6.1 Using real world data without authoring

In the quest to achieve greater and more realistic detail on map generation without the requiring manual programming, a part of the terrain generation research community focused on the application of GANs to the task at hand. These neural networks are often trained with datasets composed of real-world satellite imagery. This allows the network to learn real world terrain distributions, instead of trying to mimic real terrain, which is the case of other algorithmic-driven implementations.

Beckham and Pal (Beckham & Pal, 2017) proposed a two GANs pipeline for terrain generation: one DCGAN responsible for generating the heightmap itself and a pix2pix GAN responsible for creating the textures depending on the corresponding height of the terrain. Both these NN had no modifications done, so the descriptions on Section 2.1.5.2 and 2.1.5.4 apply. To train the networks, a sliding window is passed through two high resolution images of the Earth, taken from NASA's "Visible Earth" project. One of the images contain height information, whereas the other contains texture information. The DCGAN is then fed with the height data and the pix2pix GAN with the texture data. The discriminator receives a pair composed of the heightmap data and the texture data. The low resolution of the images utilized in the dataset can contain several biomes, confusing the GAN during training. To improve the texture generation, a reference texture was chosen, desert. Pairs are chosen from the texture collection based on a Euclidean distance between the base texture and all others on the collection. Textures with a low Euclidean distance are then chosen, limiting the textures to desert biomes. A resulting terrain can be seen in Figure 2.13.

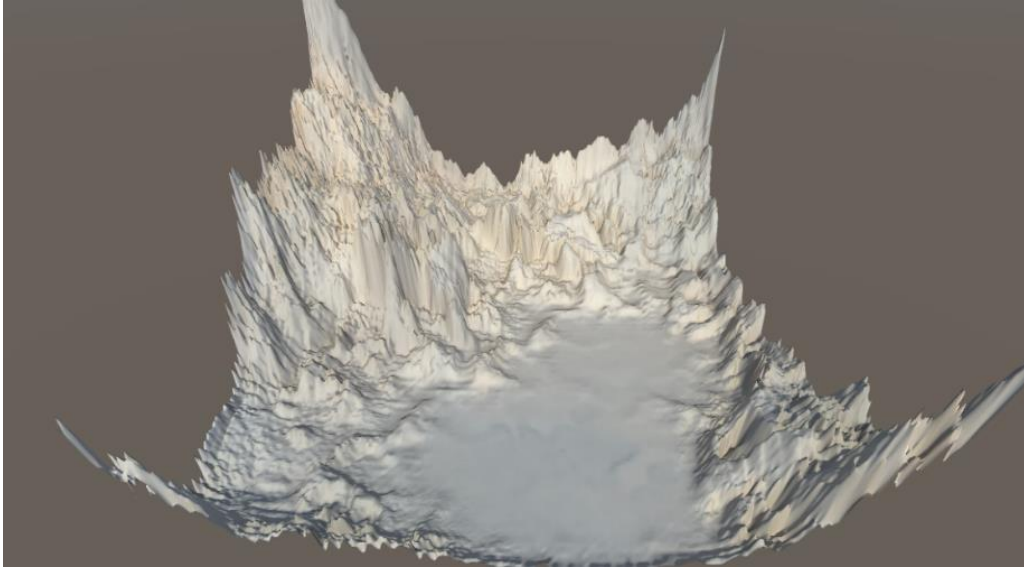


Figure 2.13 - Beckham et al. resulting terrain (Beckham & Pal, 2017).

Wulff-Jensen et al. (Wulff-Jensen et al., 2018) propose the use of DCGANs as a means to expand the algorithmic pool of terrain generation, citing the versatility of GANs regarding possible outputs depending on the training done. This opens the possibility to influence outputs while still retaining diversity and randomness, problems that are present in popular solutions such as Perlin Noise. No modifications to the DCGAN architecture were made, so the description in Section 2.1.5.2 applies.

The dataset used was retrieved from *viewfinderpanoramas* and consisted of images from the Alps. These images were then cut into 64x64 to fit in the input of the discriminator network (D). This resulted in 360,000 images. The resulting heightmaps were then rendered in Unity3D and textured using a built-in tool (Wulff-Jensen et al., 2018).

Several tests were made, including diversity tests between the generated images and Perlin Noise. These tests are used to measure the similarity between two images. These tests were done using Mean Square Error (MSE) and Structured Similarity Index (SSIM) to measure image pixel-based diversity. Usability tests were performed with only three participants. Diversity tests were positive, but usability tests were somewhat negative. According to the authors, the users were unable to separate the texturing and the lack of features, such as trees and grass, from the map topology, which was the focus of the test. This led to the conclusion that all these elements are intertwined, and further aesthetic addition are important to increase the perceived usability (Wulff-Jensen et al., 2018). One of the renders shown to test participants can be seen in Figure 2.14.

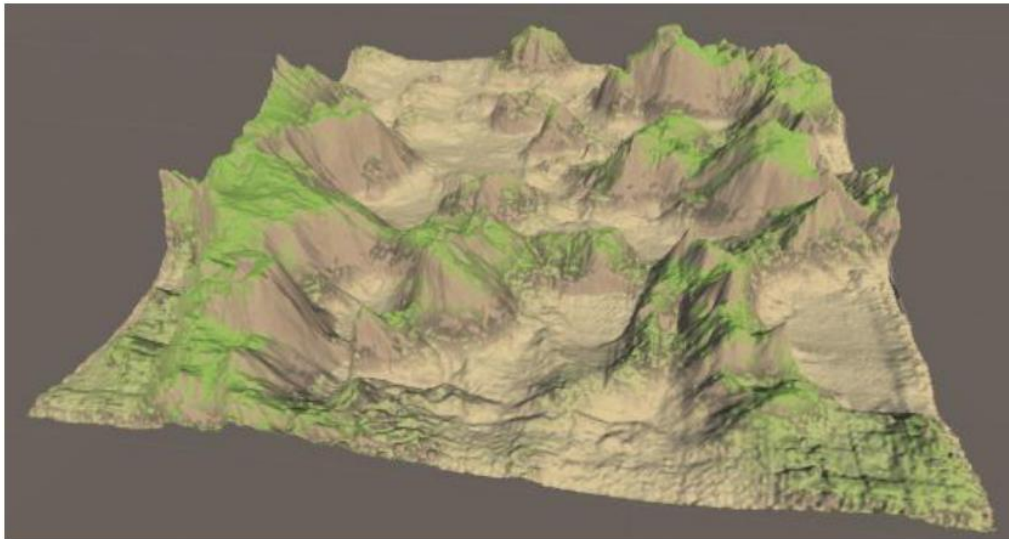


Figure 2.14 - Example of final render shown to test participants (Wulff-Jensen et al., 2018).

Spick et al. (Spick, Cowling, & Walker, 2019) proposed a new approach to generate heightmaps similar to the regions that were used to train the network, giving the user the ability to choose what kind of terrain they want based on the training. Each training session of the Generator used a 1400x1400 region, equivalent to a one hundred squared kilometers. This approach uses Spatial GANs (SGAN) (Jetchev, Bergmann, & Vollgraf, 2017). SGANs are similar to DCGANs, but they differ on a few key aspects. Firstly, fully connected layers are removed, allowing for a more robust learning of features and, secondly, the generator is inputted with a tensor of latent noise instead of the single noise vector used in DCGANs. The generator network's depth was set to six 6 after some experimentation, whereas the discriminator network's depth was set to 3. The resulting terrain and its corresponding generations can be seen in Figure 2.15.

The employed dataset was taken from *Shuttle Radar Topography Mission 30m (SRTM)*, an open-source elevation map dataset that provides thirty meters to one pixel of resolution of the world's topology. An automated script was developed to extract terrain patches, given a latitude and longitude. Result-wise, Spick et al. presents positive results, especially when comparing SGAN outputs to Perlin Noise, claiming that this solution boasts far more structured and controlled output and with possibility of control depending on the inputs fed to the network. An online test was done where participants were shown a pool of twenty samples, each of them with four possible regions from which the generation was derived from. Each participant was asked to identify which original training region more closely resembled the presented generated sample. Testers were able, with a high interval of confidence, to positively identify the areas from which the generation was derived (Spick et al., 2019).

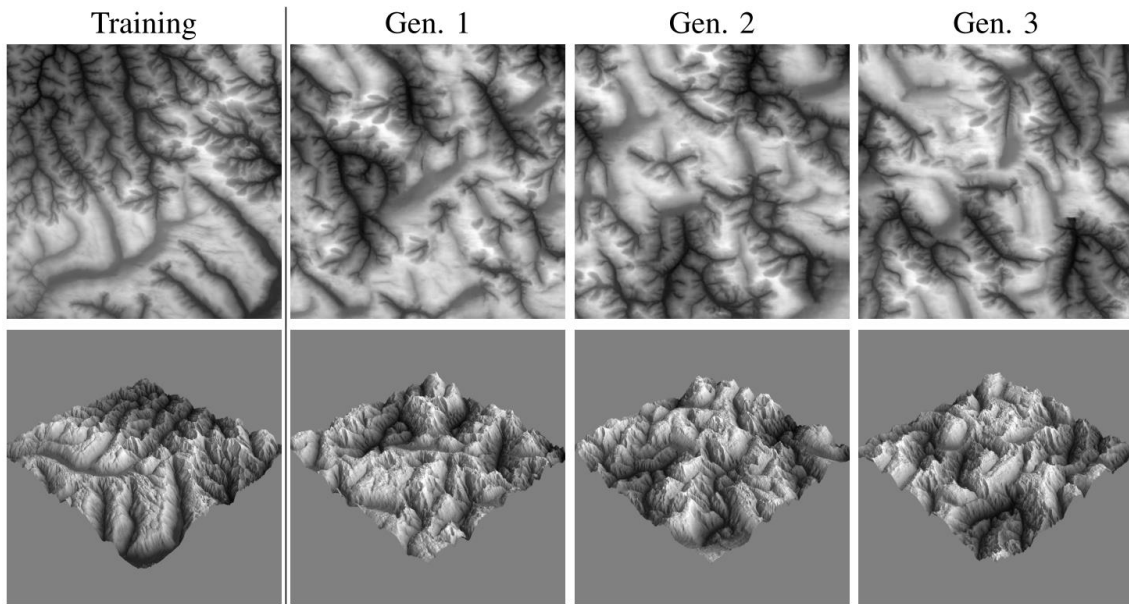


Figure 2.15 - Spick et al. resulting heightmaps and 3D renders of three different generation of training.

2.3.6.2 Using real world data with authoring

As previously mentioned, there is a need for controllable generative tools that fill the void left by automatic, low parameterization, and overly complex tools. Such complexity, at times, is more of a detraction than a benefit for designers. As such, a mixed initiative framework can mitigate some of these problems by combining the machine’s learned distribution and the human’s creative drive (Guzdial, Liao, & Riedl, 2018). Bearing this in mind, Guérin et al. (Guérin et al., 2017) set out to create an authoring tool, with simple users’ sketches used as inputs, highly controllable, in real time and with realistic outputs. Figure 2.16 depicts various possible authoring inputs and the resulting generated terrains. Users have three colors at their disposal, red to draw ridges, blue to draw “rivers”, meaning depressions, and green to dot the sketch with altitude cues, which are peak points of altitude on the map. Users have several ways of generating terrain (Levelset-to-terrain, erasing parts of the terrain to be regenerated and erosion), but for this dissertation, only sketch-to-terrain method is relevant. This method takes a users’ sketches and transforms them in terrains, as seen in Figure 2.16.

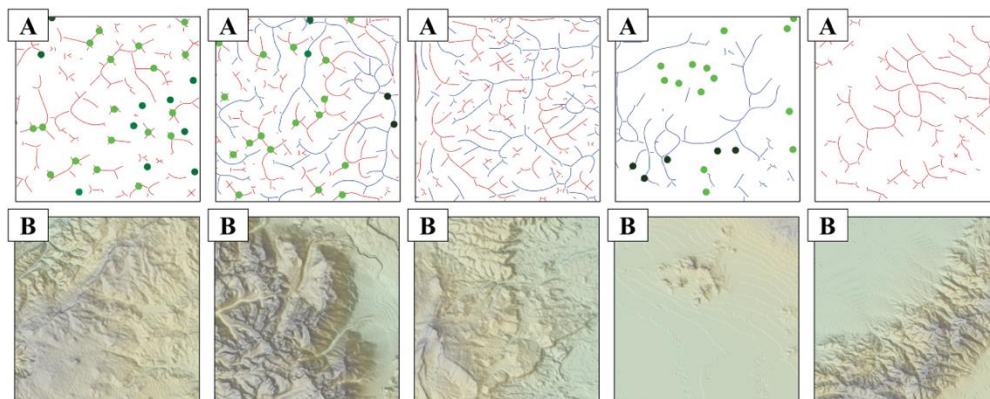


Figure 2.16 - Guérin et al. different inputs and their corresponding outputs (Guérin et al., 2017).

To train the cGAN, a pair is needed. In this case, one part of the pair is a generated sketch representing a real-world terrain and the other part of the pair is the real-world terrain. The real-world terrains are a collection of Digital Elevation Maps (DEM), taken from USGS Earth Explorer (EarthExplorer, n.d.), composed of thirty-five 3600x3600 patches of one square degree with a precision of one arc-second from NASA's SRTM. These DEMs are also used by a dedicated algorithm to automatically generate their sketch counterparts. The sketch is generated in three different phases. In the first phase, and using a real-world terrain taken from the collection, water flow is simulated utilizing a modified steepest descent D8 algorithm, followed by generating the river network using an algorithm inspired by Tarboton et al. (Guérin et al., 2017). Ridges are then detected by inverting the terrain and using the same method used for river detection, followed by detecting altitude cues, which are defined by points where the water flow accumulated (Guérin et al., 2017). This generates a sketch similar to the ones found on the top part of figure 2.16, that represents a real-world terrain, which will then create a pair for the dataset used to train the cGAN.

When it comes to results, the authors obtained positive feedback. The method produced more realistic and detailed results than its counterparts. Human testing (five subjects) revealed simple usability and satisfaction with the produced terrains. Participants were asked to draw three terrains described in text. Almost all participants being queried responding very positively, with one sole participant reporting they did not find easy to express their goal when drawing. These results and the work done by Guérin et al. are extremely exciting for the prospect of creating usable yet powerful mixed-authorship tools capable of creating believable terrains, one of the central assets of many games. Although, in their tool, Guérin et al. provides more than one way to generate a terrain, the pre-trained sketch-to-terrain NN provided on the authors personal page is powerful enough to be able to provide the users with the tools necessary to author their own terrains, as it is also the most intuitive of the ones present in Guérin et al. Adding functionality to ease the creation process, as well as extra color option to expand on the possible ways to add rich terrain variation to the users sketches should give users a good authoring experience.

The work of Guérin et al. used the previously explained cGAN (see Section 2.1.5.3), as the workhorse for the generative and discriminatory processes. *Unet* and *PatchGAN* architectures were used for the generator and discriminator networks, respectively (see Section 2.1.5.4). The networks training also follows the original work of Isola et al., with the addition of horizontal and vertical flipping to add more variation to the training data.

Tool's Design and Implementation

This chapter presents what was developed in order to satisfy the objectives outlined in Section 1.4 and the research questions enumerated in Section 1.3. The development has been divided in three major parts: the first being research and experiment, where a suitable NN capable of outputting the desired heightmaps needed to be found and tested to corroborate if indeed it had the desired outputs and the capabilities of being used in a mixed initiative setting, and furthermore, test it to verify whether there was a possibility of finding unused latent information to enrich the application. The second part was the development of the application itself, where a NN would need to be integrated with a graphical engine to achieve the necessary results and seemingly streamline the drawing process and the NN processing without having to step out of the application. And finally, developed testing capable of assessing the application for usability and the outputs appearance, when compared to other outputs from other methods, which will be further discussed in Chapter 4

3.1 Tool Component Overview

The developed 3D terrain tool is composed of two components. These components, and their interactions, can be seen in Figure 3.1. The first component is the tool's UI developed with Unity3D graphical engine, where the user interacts with the tool itself. It is in the tool's UI where the user sketches their input and visualizes their generated terrain. When the user concludes a sketch, it is sent to the NN, the second component of this tool, where it will be processed and transformed into a grayscale heightmap. A heightmap is a raster image where each pixel has a value that is interpreted as value for displacement on a 3D mesh, often visualized as luma of a grayscale. Black represents the minimum height possible and white the maximum height possible. The values of each individual pixel of the heightmap are then loaded into Unity3D, thanks to a developed method, which will then pass its values to another developed method tasked with displacing points in a flat mesh according to each pixel's grayscale values. This results in a 3D terrain that is rendered by Unity3D using the heightmap processed by the NN with the users sketch as input. The NN used in this dissertation is a pre-trained model of the cGAN used in Guérin et al. work (Guérin et al., 2017), capable of taking user's sketches and generating a heightmap based on the drawn user sketch. This allows users to author terrains with ease, thanks to the labelling capabilities of the cGAN that allow for users to influence the generative process with their sketch, which are inputted as labels to the cGAN that sees the sketches as labels to guide the terrain generation. More details on cGANs are present in subsection 2.1.2.4.

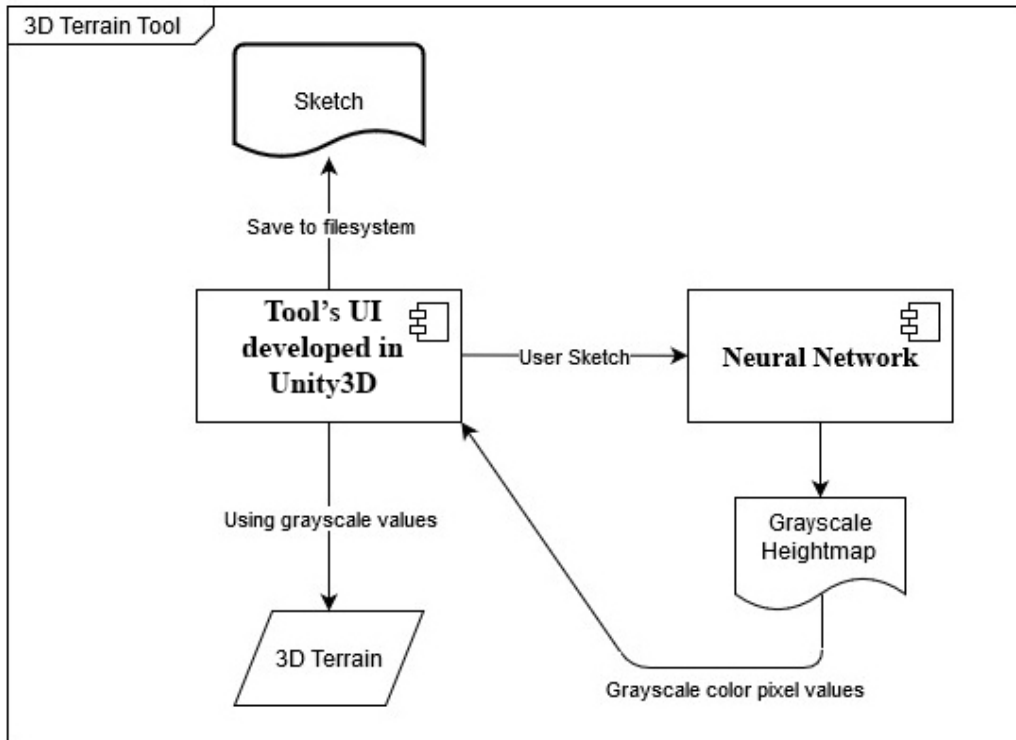


Figure 3.1 - Tool component diagram.

This tool was developed using Unity3D as the graphical engine, written in C# and in Ubuntu 18.04, as per recommendation of the NN authors. When searching engines suitable for this project, it was concluded that using an established enough platform and with a strong and mature community was best. This way early life cycle issues newer engines may have are avoided and having a strong and mature community from which to tap knowledge would be beneficial to solve any problems that eventually arise. This reasoning can also be applied to the selected language, C#. Widely used in graphical development, with considerable documentations and use cases, makes it a strong candidate.

From these prerequisites two possible candidates emerged: Godot Engine (Godot Engine, 2020) and Unity3D (Unity, 2020). Pros and cons to both engines can be seen in Table 3.1. It was decided that having a mature platform was more important for the development of this project as a stable and well-known engine is less prone to unforeseen errors and problems, as well as any roadblock can be more easily researched and a solution more probable to be learned. For these reasons Unity3D was chosen.

Unity3D	Godot Engine
C# support (among others)	C# support (among others)
At the time non-native Linux support	Native Linux support
Proprietary but with free personal use plan	Open-source MIT License
Mature community	Small but growing community
A plethora of learning material	Comparatively less learning material

Table 3.1 - Unity3D and Godot Engine pros and cons comparison

3.1.1 Neural Network System Requirements

In terms of hardware requirements for this tool to run and be able to make use of the NN, a *NVIDIA* GPU with *CUDA* cores is needed. Without a *NVIDIA* environment, the NN will not be able properly function and scripting alterations are needed for it to be able run from the CPU. While this will allow users without *NVIDIA* GPUs to be able to compute input images, sub-optimal performance will be the best a CPU will achieve when compared to the performance offered by a GPU. This is especially notable if any training is necessary, as GPUs are more parallelized than CPUs and their superior bandwidth capabilities enable GPUs to deal with vastly bigger amounts of data more efficiently. A GPU is strictly needed to train the network, but in the case of a pre-trained network, a CPU will work for production purposes, although a GPU is still recommended due to its architecture and the highly parallelized workload.

Software-wise, a *NVIDIA* driver compatible with the GPU being used is necessary. Afterwards, and since this development was made on a Linux platform, to make use of *CUDA* cores, we must install *CUDA* dependencies for Ubuntu 18.04 or no *CUDA* cores will be detected ("Installation Guide Linux : *CUDA* Toolkit Documentation", 2020). After this, *Docker* is needed to run the NN on a container where all the remaining dependencies are installed automatically or, if the preference is to run the NN outside a container, *Tensorflow* 1.4.1 must be installed, plus *NVIDIA CUDA*® Deep Neural Network library (*cuDNN*), a GPU-accelerated library of primitives for DNNs, is also necessary. Additionally, although not strictly necessary, it is recommended the installation of *nvidia-docker*, which allows the container to have access to the GPUs driver installed on the host computer, permitting access to the GPU from inside the container without being necessary to exactly match the driver installed on the host machine with the driver installed inside the container (Olson, Calmels, Abecassis & Rogers, 2016).

It is necessary to have two versions of Python installed if you are running the NN inside a docker container, Python 2, and Python 3. The latter, Python 3, is needed to run the Docker script provided by Guérin et al., responsible for setting up the container and specifying what is installed inside said container, and Python 2 is necessary to then run the NN script inside the newly created Docker container.

In this development, we opted to use *Docker* and run the docker file made available by developers of the NN used, as it streamlined the requirement installation phase and avoids the compatibility issues between required software that normally arise in environments such as these when trying to install all dependencies directly on the host machine.

3.1.2 Tool Flow

Figure 3.2 depicts the tool pipeline and its flow, from user input, to final NN output and respective rendered 3D terrain. The users start by drawing a low-level input, representative of their desired terrain, exemplified on Figure 3.3. Similarly to Guérin et al. (see Section 2.3.6.2), the user has the same three colors to choose from, plus another three color variations, one for each one of the original colors (further detailed in Section 3.2). When satisfied, the user presses the Play button (further detailed in Section 3.5), which triggers the beginning of the generative and automatic sequence responsible for the terrain generation. Then, the users' input is saved in the image log (further detailed in Section 3.6) and a NN script (further detailed in Section 3.3) is called and passed the user's sketch as input for the NN to generate a heightmap using the users sketch as conditioning for its generative process.

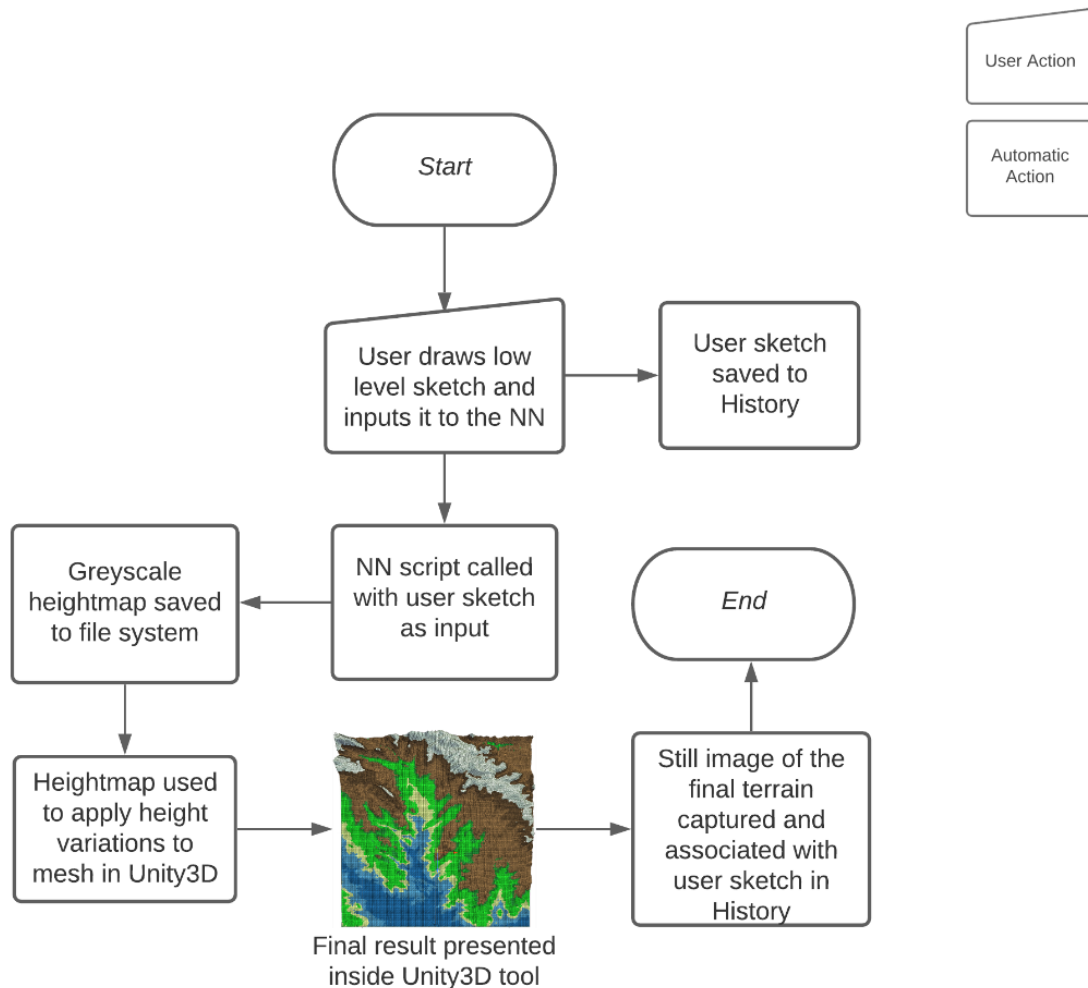


Figure 3.2 - Application Pipeline.

When the NN finishes its generative process, it outputs a grayscale heightmap. An example of such heightmap can be seen on the right side of Figure 3.3. Then, the pixel values of this heightmap are passed to the tool, which will use this output to displace a mesh to create the corresponding 3D terrain render. Immediately after the 3D terrain is rendered, a screenshot of the terrain is taken from a top-down angle and associated to the users' input to give them a preview of what their input produced when they browse through the Input Log (further detailed in Section 3.6).

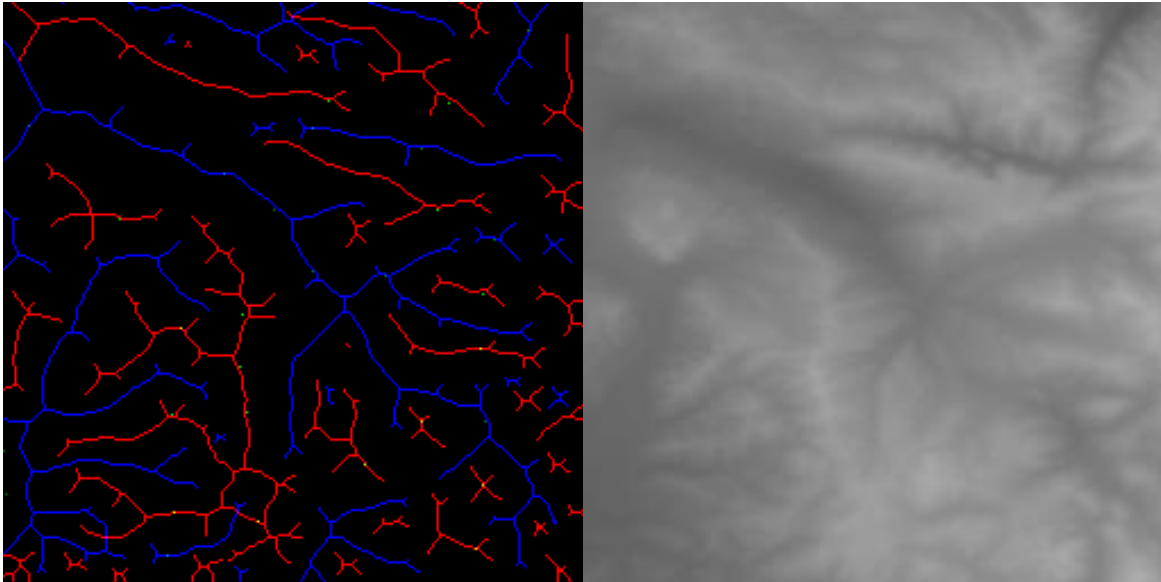


Figure 3.3 - User Input/NN output pair. Left: Potential user input. Right: Corresponding grayscale heightmap generated by the NN using the user input as conditioning.

3.2 Single Input Preparation

To be able to execute the NN, be it for the sensitivity tests and regular use by users, the NN had to be able to accept a single, unpaired image. As previously explained in Section 2, cGANs are trained utilizing a pair of images: one a generative target and the other a user input that should lead to the generation of the target. Having access to a pre-trained model, meant no training was necessary, but exporting the model was still necessary for it to be production ready: to freeze the network's parameters and remove any data relevant for training only.

Checkpoints are merely saved states of the NN, in which the weights, training configurations such as loss, epochs and optimizer state are stored. This is a good practice to avoid loss of training in case of failure. From checkpoints training can be resumed or inference itself can be done, although not ideal. Exporting removes unnecessary and irrelevant metadata to the inferencing process, reduces model size and increases inference speed, since all that is needed for inference is the model and the weights. This was done using an existing export mode in the original NN script.

From all the operation modes provided in the original NN script (e.g. train mode and test mode), none gave a way to use the model in a production setting, meaning all the modes were either for training or testing and were expecting a paired image (a generative target and an input). One of the solutions was generating an image pair with the user input and a blank target and run the NN in test mode but this workaround was deemed insufficient, as the objective was to change the script in order for it to accept a single image as input. A similar issue to ours had already appeared in the past and the solution was made available by (Barbosa Anda, 2019). The script was based on a previously available script that made use of exported models and allowed for single image inference. With this and after some minimal changes to the script, a NN script capable of accepting a single image as input was developed.

3.3 Neural Network Sensitivity Analysis

As previously mentioned, what allows us to generate heightmaps from the users input is a NN. The NN used in this dissertation is a *TensorFlow* translation, of the work accomplished by Isola et al., done by Guérin et al., and its explanation and details can be found in Section 2.1.5.4 and 2.3.6.2.

This dissertation used part of the work of Guérin et al. The author provides, under MIT license, not only the NN used on their work (Guérin, 2018), but also the training data and a pre-trained model (Guérin, 2018b). Given the hardware constraints (the only CUDA core enabled GPU accessible was a portable edition GPU with only 902Mhz frequency and two gigabytes of graphical memory) we opted to use the pre-trained model, which came with a validation dataset. Previously mentioned in Section 2.3.6.2, the dataset is composed of Digital Elevation Maps (DEM), taken from USGS Earth Explorer (EarthExplorer, n.d.), composed of thirty-five 3600x3600 patches of one square degree with a precision of one arc-second from NASA's SRTM. These DEMs are then used by a dedicated algorithm to automatically generate their sketch counterparts, which are paired with the real terrain counterpart to create training pair for the cGAN. The three original colors were maintained with their original RGB values: Red, that produces elevations, was kept at RGB (255,0,0), Blue, responsible for depression on the terrain, was kept at RGB (0,0,255) and Green, *to be drawn solely as dots* that translate in to altitude cues, was kept at RGB (0,255,0). After setting up the NN, the validation dataset was ran to make sure everything was running as intended.

After a couple positive validation runs, it was verified that the validation dataset did not test edge cases (e.g., inputs with sparse information or with too much information) this did not cover the full length of behaviors one might expect from the NN. It was decided to run some behavioral tests to better grasp how the network would react to less conventional inputs (an example of a conventional

input sketch can be seen on the left side of Figure 3.3), such as inputs with minimal lines, and check if those inputs translated to outputs that could be somewhat predicted by the user. Given that a pre-trained model is being used and being treated as a black-box component, it is necessary to better understand how the NN would react, not only to more complex inputs but also to sparse inputs. This is important because potential users could have a diminished experience if they sketched an input that would not be correctly processed by the NN due to not being trained for that particular input. It is also important to understand how the NN reacts so it can be better understood what kinds of additional features (beyond just sketching with the pre-trained colors) have the potential to be added to the tool.

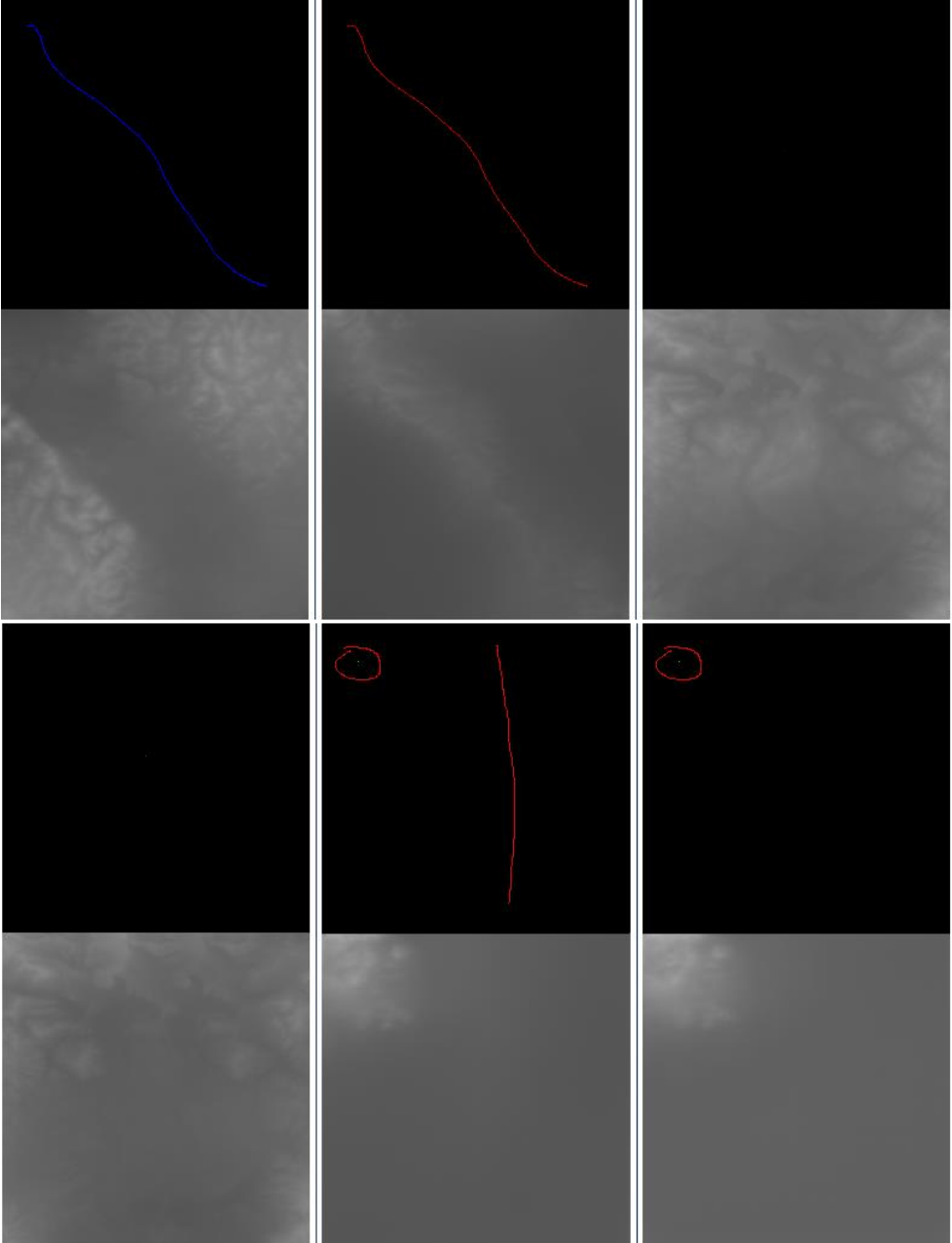


Figure 3.4 - Initial test inputs and their corresponding output. Top right input is a singular blue dot, and bottom left a singular red dot. It is possible to see in these two test examples that when faced with sparse input data, the NN fills in the voids with a recurring pattern.

Figure 3.4 shows some of the tests that were done. Starting from the two left most examples, singular line inputs are well handled by the NN, with the blue line displaying a lot more freedom to fill areas without input information as it sees fit. This is happening due to the geographical areas used for the dataset having more areas that contain elevation surrounding areas with a deeper depression than the other way around. Another detail unfolds when we analyze the inputs that are composed of only a single dot. When faced with so little sketch input information, the NN populates the heightmap with a regular pattern. Preferably, the NN would produce a completely random heightmap but when faced with a blank input, the NN defaults to an artifact resulting of the cGAN training (Guérin et al., 2017). On the bottom two right most pairs, something interesting happened: the NN ignored the red line in the middle and gave all its height weight to the red circle with the green dot in the center. Not only that but the overall median color of the empty portion is darker, hence deeper, on the output with the red line than in the one without. This led us to believe that, when it comes to height, the NN will always try to make green the highest part of the map. This was further substantiated when testing with Dark Green, as seen in Figure 3.5.

We then continued to feed the NN some more intended and unintended inputs. An unintended input is any input that causes the NN to output an heightmap that has other colours other than a grayscale. These inputs can be green lines, high color saturation shapes and thick sketch lines. Examples of such inputs can be seen in Figure 3.5 and 3.6, as well as in Annex A. In Figure 3.5, the top two inputs are a simple pixel color conversion, from blue to red, and the results are as expected: areas without sketches are the opposite of the surrounding inputs: if the surrounding is Red, there is a tendency to deepen the areas without information and if the surroundings are blue, the NN tends to raise the areas without information.

When experimenting with formats supposedly not supported by the NN, in this case green lines, at first the outputs were what we expected: corrupted, unsupported, showing colors others than on a grayscale. But if the opacity of the brush being used to draw the lines was reduced, meaning there was a color bleeding from the background into the color being used, the output became usable. By the documentation this should not be possible as Green is supposed to be used strictly as dots. This led us the believe there was some latent representation information that could potentially be leveraged to expand the application.

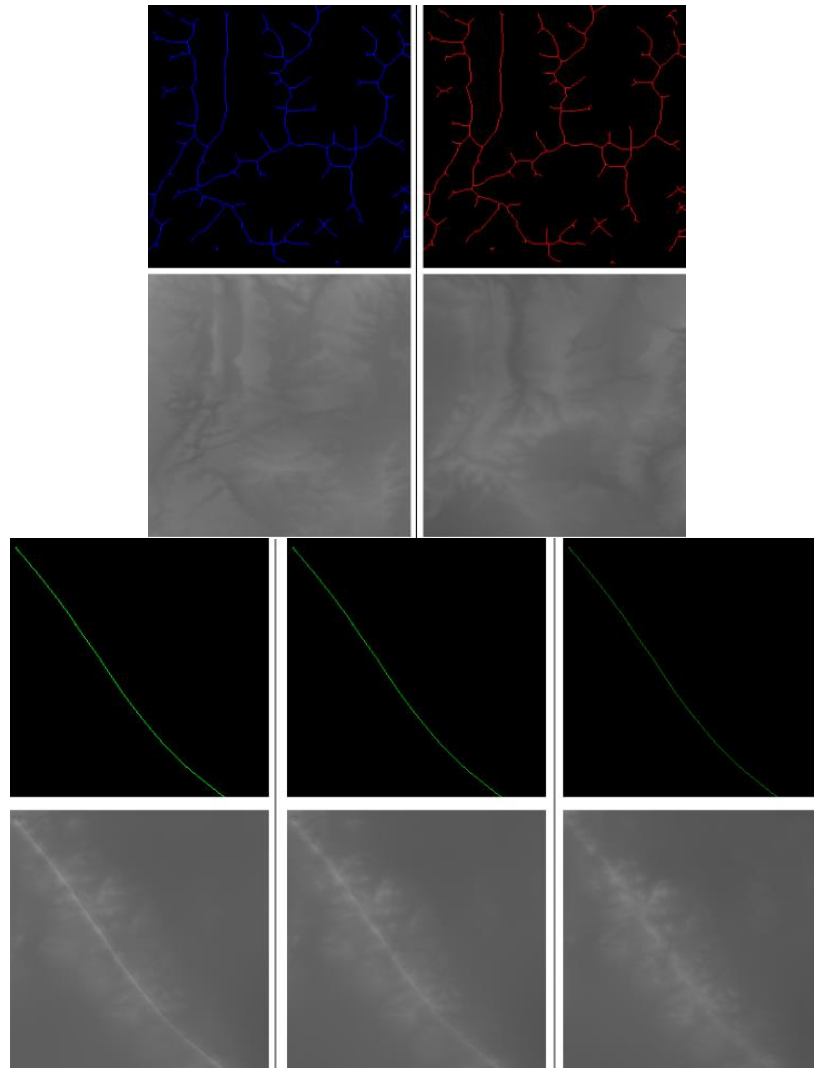


Figure 3.5 - More test inputs and their resulting heightmaps. On the top we have a color conversion between blue and red. Same input drawing but different color. Corresponding outputs display an expected result. On the bottom right we have a corrupted output due to green being used as a line instead of a dot. Subsequent inputs have a lower opacity than the initial input. The last input has a 50% opacity, and the output presented no corruption.

It was initially decided to explore the potential implementation of brush patterns. Several patterns and combinations were attempted, from solid patterns, to patterns with changing opacity, as seen in Figure 3.6, to having brush thickness control, seen in Figure 3.7. We conclude that opaque blue brush (terrain depressions) patterns tend to corrupt the outputs more than opaque red brush (terrain elevations) patterns, and that blue shapes add no extra intricacies to the shape of the terrain, mostly just limiting themselves to adding a shapeless depression in the terrain, as exemplified in Figure 3.6. Adding to that, we also conclude that red patterns (terrain elevations) did not have as much predictability as lines, resulting in terrains not reflective of user expectation, contributing to a potential feeling of lack of control by the user. Furthermore, giving the user opacity control leads to a lot of inputs being unfit for the NN due to high color saturation, resulting in a diminished experience and a

lot of red tapping to make sure no non supported inputs were drawn, e.g. input that lead to an overflow of the NN neurons and result in outputs that cannot be properly transformed in heightmaps.

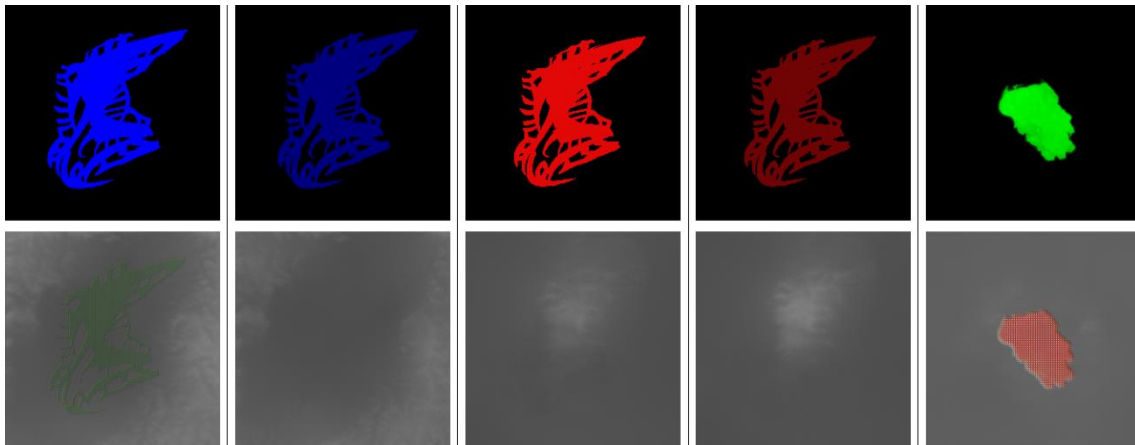


Figure 3.6 - Pattern testing with different opacities. Top row are inputs and bottom row are corresponding heightmaps. On the two left most outputs, we can see green in the shape of the input. This means the NN was unable to process the input due to the high color saturation on the input.

Moreover, we also note that increasing line stroke width is an hit and miss situation, either, adding little variation to the output or being dependent on opacity control in order to produce a viable, uncorrupted output, as it be seen by the two input/output pairs on the right side of Figure 3.7. We made the decision of not implementing this feature to keep the outputs from becoming corrupted and to avoid having a feature such as line stroke width but needing to limit it to a point that it might as well not be available.

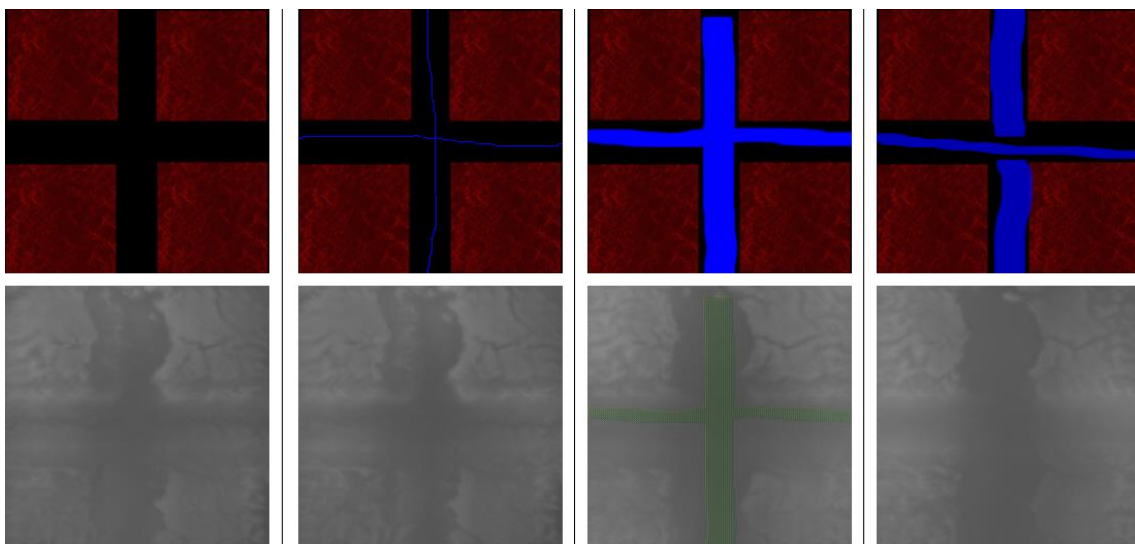


Figure 3.7 - More pattern testing with different opacities. Top row are the inputs and bottom row their corresponding outputs. Third output from the left is corrupted as it presents colors other than the ones present in a grayscale.

Although brush patterns and line stroke width control did not produce the expected results, color variation is still a possibility due to the results seen in Figure 3.5: a green line produced a viable output when it was not supposed to (the NN was never trained from origin to recognize green lines as an input, only green dots), as long as the color saturation was reduced, effectively transforming it in a different green than the original.

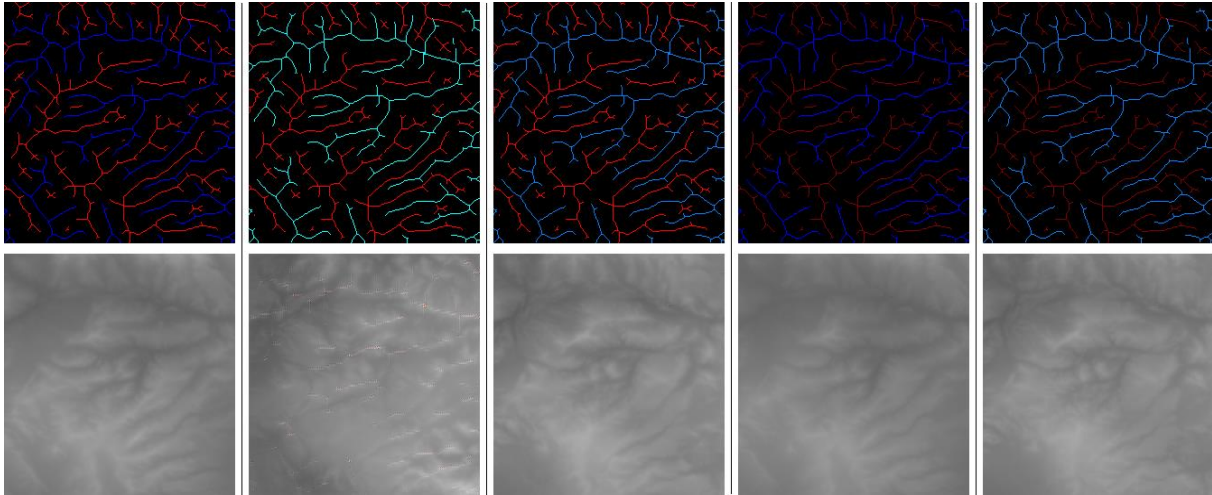


Figure 3.8 - Color variation testing. Original pattern on top left corner. Subsequent Inputs have the same pattern but with different colors. Second output from the left presents some corruption due to the blue being used having a too strong green component.

After some experimentation with colors other than variations of the originals, some of which can be seen in Figure 3.8, we conclude that there is indeed space in the generative process to add new color variations to the application, bringing the total to six colors, but only if the colors are variations of the original three colors. For completely new colors, a new training dataset would have to be put together and new colors would have to be tied to a new terrain structure other than those already paired. Additionally, to the original red, blue and green colors with which the NN was trained, three new color variations were added: Dark Red, RGB (127,0,0), which produces more constant height elevations and generally less pronounced heights, Medium Blue, RGB (0,127,255), produces less pronounced depressions. Dark Green, RGB (0,94,0), that produce elevations like red but, due to the weight the NN attributed to green, are always the tallest elevations, even when red is present in the input sketch. Some examples of these tests can be seen in Figure 3.9 and in the Appendix A.

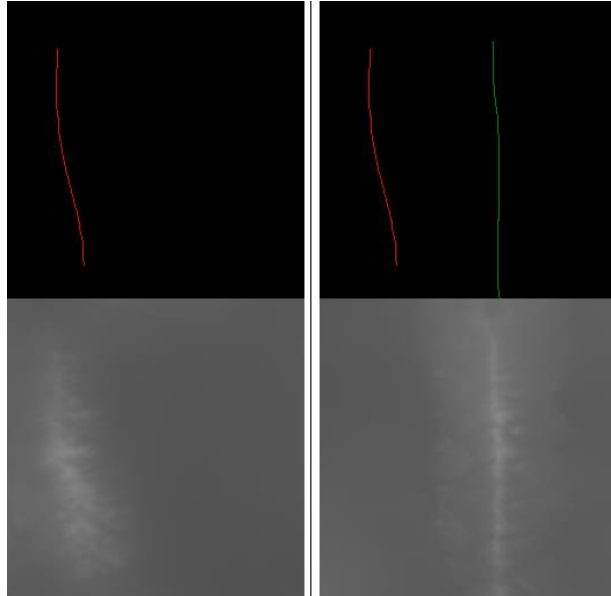


Figure 3.9 - Dark green and red weight difference. Note how the red line is ignored on the right output when a solid dark green line is present in the sketch. The NN always tries to make green inputs the highest.

3.4 Software Structure and Terrain Generation Logic

Figure 3.10 presents the class Inheritance diagram of the tool. As we can see, most classes inherited from *MonoBehaviour*. This is the base class of *Unity3D* engine, and it needs to be inherited if any method present in the class is attached to an object inside of *Unity3D* or if any sort of communication between engine and code is necessary, such as listening to an input event, accessing the state of any given object in the scene, or updating states from frame to frame. The remaining classes are either static, meaning they are sealed and cannot be instantiated, inherited, inherit, or are utility/helper classes which are not directly called by the engine, only by other classes. The only exception to this is the *MapGeneratorEditor* class, which extends the *Unity3D Editor Class*, that allows developers to create custom editor changes.

A summary of each class role is necessary to better understand Figure 3.10. Starting from the top left, the *CameraMovement* class is responsible for camera movements as well as color changes on the camera command labels found on the tool's UI, which will be presented in Section 3.5. The *Drawable* class is responsible for everything brush related, from tracking mouse movement to painting the appropriate pixels. It also controls the copy/paste function as well as saving the user's sketch to the filesystem and naming it. The class also is responsible for starting a console process and passing the NN script run command to the console, with the previously saved sketch as the input. Related to this class, there is *DrawingSettings*, a class with helper methods that are used to set drawing settings, such as brush color and other brush settings not accessible by the user, such as width and transparency. *ImageLoader* is the class that loads the stored sketches into the image log. *MapDisplay*

is the class that renders the 3D terrain on the screen, as well as the colored texture. In the *MapGenerator* class, information regarding texture and heightmap values are centralized and passed to the appropriate methods for mesh displacements and texture application, as well as height multiplier, what shading to use, and other attributes related to mesh creation. The *Noise* class creates a two-dimensional array with Perlin Noise values that can be used to generate 3D terrains and it is also responsible for getting the NN generated heightmap values from the filesystem and placing them in a two-dimensional array, just like the noise values. The *PanelScript* class is responsible for managing the panels in which a tutorial is presented to the user. The *PrefabButton* class is tasked with placing on the canvas the clicked sketch from the image log as well as drawing on a preview window the associated 3D terrain when sketches on the image log are hovered with the mouse. The *SaveMeshInEditor* simply listens to a key press and loads the existing 3D terrain to an object. The *ScreenshotHandler* class is tasked with taking a screenshot of the generated 3D terrain, naming it accordingly and, storing it in the filesystem to be later displayed as a preview of the sketches stored in the image log.

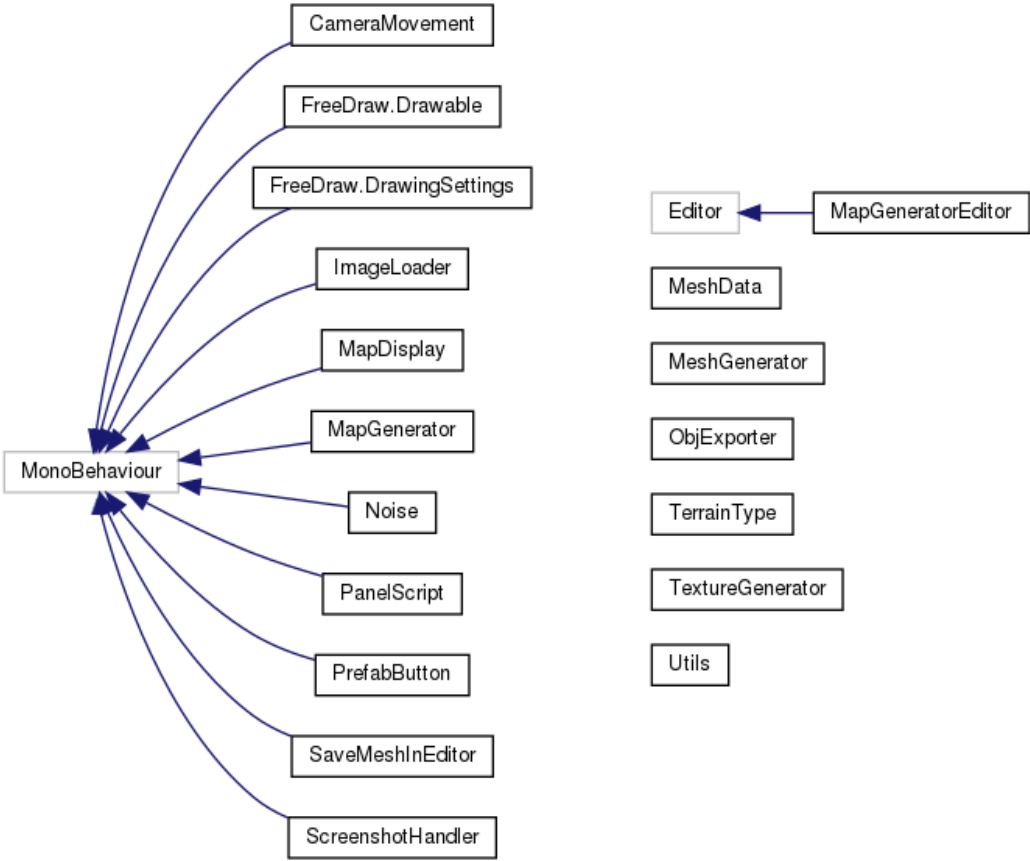


Figure 3.10 - Class Inheritance graph.

As mentioned, not all classes inherit *Monobehaviour*. For instance, the *MapGeneratorEditor* class does not. The sole purpose of this class is to add a button to the editor that generates the 3D terrain. This allows the generation of terrains without the need of launching the tool, hence why it inherits the

Editor class from Unity3D. *MeshData* class is the class that contains data pertinent to the terrain mesh. In this class, triangles that compose the mesh polygons are calculated, their normal mapping calculated, to allow for a coherent lighting of the terrain, as well as UV mapping (a 3D modelling process of projecting a 2D texture to a 3D model's surface) that allow for the defined color mapping to be applied correctly. Following this, there is the *MeshGenerator* class, which is the class that actually displaces the mesh vertices to the values found in the NN generated heightmap. It also sets the terrain's UVs so that the texture can be correctly applied. The *ObjExporter* class is used solely by the *SaveMeshInEditor* class. Its function is to take the terrain that was store in an object and transform it into an actual *.obj* file stored in the filesystem and name it. The *TerrainType* class is in fact a struct that was serialized, meaning its defined fields can be changed inside Unity3D, and its contained fields define the name, the color and the height interval in which they should be applied on the 3D terrain. The *TextureGenerator* class builds the texture, with the appropriate colors, to be placed on the 3D terrain. Finally, the *Utils* class is responsible for the creation of a visual delineator of the area being selected when the user is performing a selection of the sketch to copy.

Figure 3.11 presents the *GenerateMap* function call graph. This is the function that is called whenever its necessary to generate a new terrain, be it because the user drawn another input and a completely new terrain mesh is necessary or because the user moved the height slider that increases or decreases the terrain's height, further detailed in Section 3.5. This method is called after the NNs output image is saved on the file system. Given its importance, a brief explanation of the call flow and what is happening in each call is pertinent.

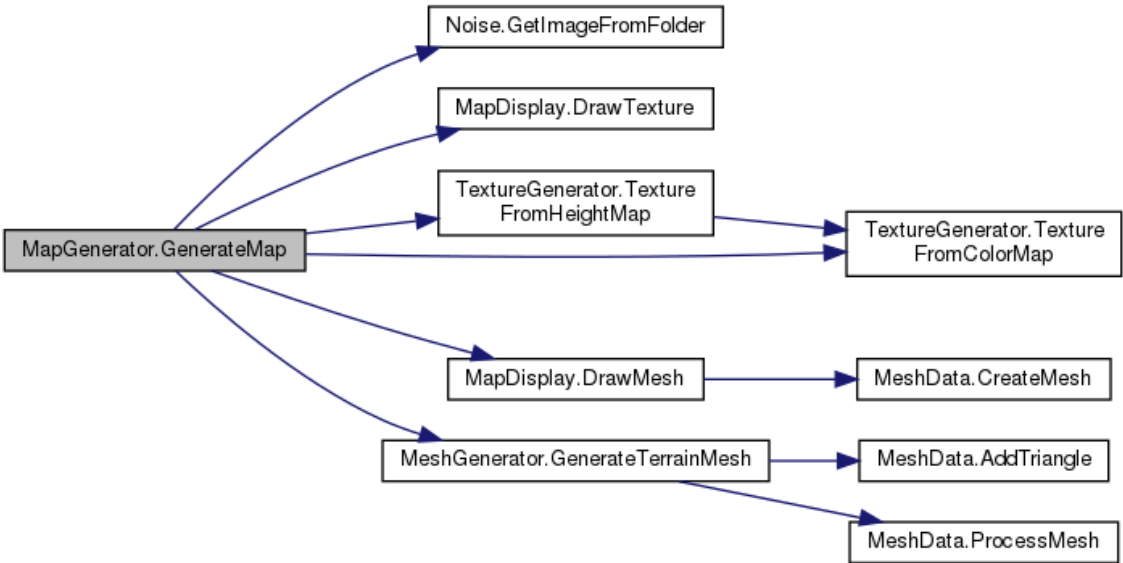


Figure 3.11 - GenerateMap function call graph. The methods *DrawTexture* and *TextureFromHeightMap* are debug methods used to make sure the proper noise map and texture were being correctly assembled.

To start, the output needs to be taken from the NN into the engine, more specifically, we want to have, inside the engine, the color information of each pixel of the output image. For this the tool calls the *GetImageFromFolder* method, whose job is to search for the output image on the filesystem, run through it and store in a two-dimensional array the linear interpolated color value of each pixel, which ranges from zero to one thanks to the interpolation. This is done so we can have a translation between color and height and additionally it allows for height multipliers, allowing for some customization.

With the height values now available to the tool, the next thing done by *GenerateMap* is a simple color assignment, achieved by comparing the height values with a defined height interval for each color. For example, every pixel in the mesh with a value equal to or lower than zero point one has its color set to dark blue. These color regions are stored inside an array of the type *TerrainType*, a custom struct, whose values are pre-defined and outside user's reach. With these two elements ready, the tool can start building the terrain mesh with the height values and colors obtained previously but we still need to build and display the mesh. For drawing the terrain mesh, the tool calls *DrawMesh*, the method responsible for drawing the mesh on screen and passes as arguments two methods. The first method passed as argument, *GenerateTerrainMesh*, oversees the displacement of the vertices with the height values previously stored and associates each three vertices in triangles, so we can have an actual mesh. For this, two helper methods are called, *AddTriangle*, which groups the modified vertices in triangles and the *ProcessMesh*, that marks the triangles to be shaded with flat shading. A flat shading was picked as to not interfere with the perception of the terrain: the objective is to produce morphologically real looking 3D terrains and not realistic looking textures. As second argument, we pass *TextureFromColorMap*, whose purpose is to transform the colors obtained for each pixel previously and transform them in a texture, ready to be placed on top of the mesh, with each pixel in the texture coinciding with a vertex of the mesh. After all this is done, *CreateMesh* is called and the terrain mesh is ready and finally passed onto the engine by *DrawMesh*.

3.5 Interface version 1

Figure 3.12 shows the first iteration of the applications' interface out of two existing ones. On the left side of the screen, we have our terrain visualizer. This is the area where the user can preview the generated mesh, in any angle they see fit: they can zoom in, pan, rotate, and reset its position to the original top/down view. The possible movements and their respective key combinations are always displayed on the top left corner of the screen, labeled with the number 1. This text changes to yellow if that combination is being pressed, to signify with some visual feedback that a successful combination of keys is being pressed.

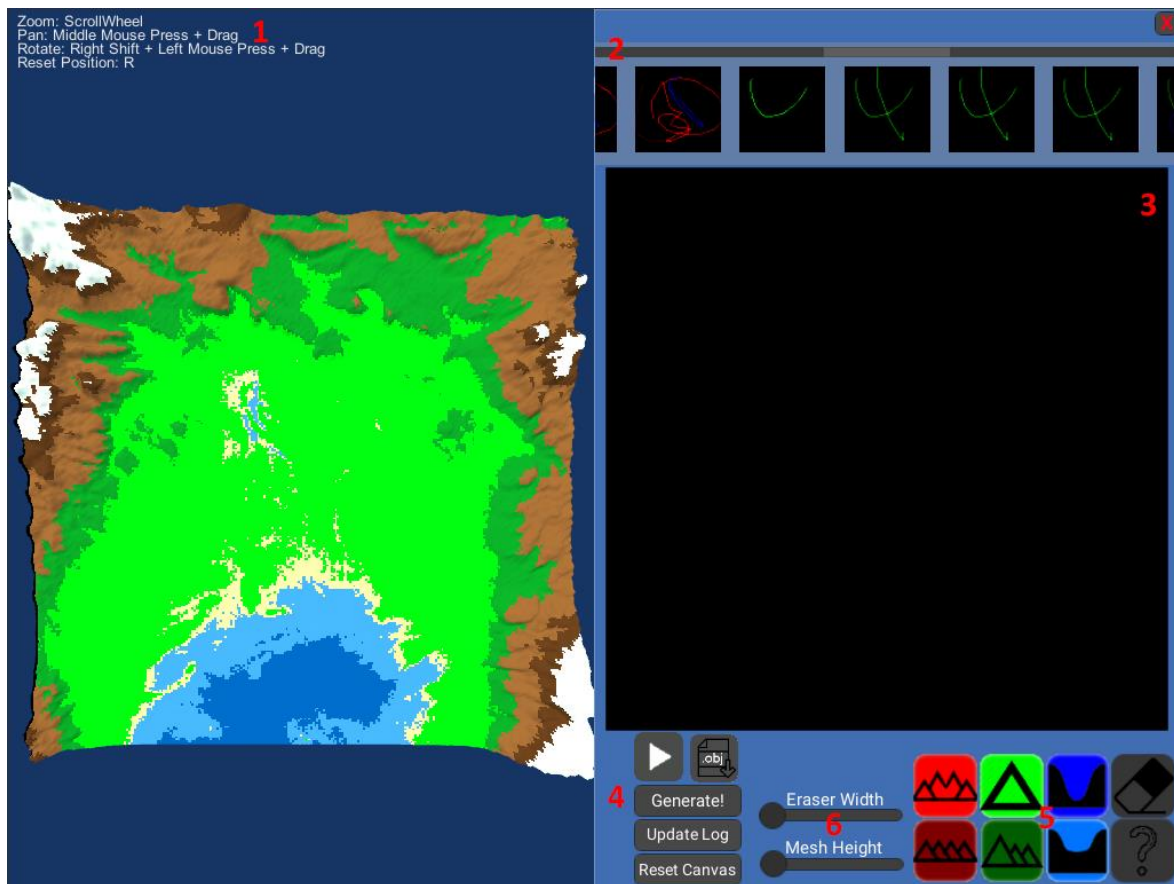


Figure 3.12 - First iteration of the interface, with a red numeric overlay. 1 – Camera movement control combinations label. 2 – Image log. 3 – Drawing Canvas. 4 – Action buttons. 5 – Drawing tools, eraser, and tutorial button. 6 – Terrain mesh height slider and eraser thickness control.

We opted for having this information always visible for clarity sake. Some people might have some previous 3D visualization experience and are familiar with the workings and combinations of key presses these types of interfaces have. Other users might have absolutely no previous experience at all and might feel either lost or frustrated if they have to randomly guess key combinations or go through the tutorial again to be able to pre-visualize the terrain they just created.

Going to the right side of Figure 3.12, the interface displays the drawing canvas and all the tools necessary to interact with the canvas. Starting from the top, with number 2, we have our Log menu, where all inputs and their subsequent previewable results are stored. Below that, with number 3, the interface’s black canvas, where the user will be drawing their sketch inputs. The canvas is a Sprite with the same size as the input for the NN (256 x 256), with a black texture applied to it. The canvas is black because input images for the NN have a black background and this way we save CPU clocks in converting the background to black from any other chosen color when inputting to the NN. We opted for a Sprite instead of a Unity3D Texture because Sprites, in Unity3D, are innately 2D and have no structures that prepare them to be applied to a 3D object, such as mipmaps and normal mapping. Since

our canvas is meant to be strictly 2D and have no perspective, this seemed like the appropriate choice to save CPU resources.

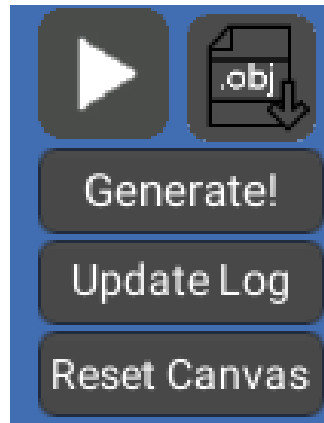


Figure 3.13 - Interface detail: action buttons.

Figure 3.13 presents a detail of the action buttons, which are labeled on Figure 3.12 with the number 4. Starting from the top left, the Play button is responsible for initiating the generative process' chain of events. Upon pressing the button, the contents of the canvas are scanned and encoded to an image file. This file is then saved in a specific folder and with the correct size accepted by the NN. After this is completed, the NN is initiated with the saved image file as the input.

After the NN processing is complete, which is indicated with the console closing, the user clicks on the "Generate!" button to update the mesh on the Terrain Visualizer on the left side and display the resulting terrain, which uses the newly outputted heightmap. The "Generate!" button, when clicked, triggers the *GenerateMap()* method, whose flow was explained in Section 3.4. Following that, it is necessary to update the Log, and for that the user needs to click the "Update Log" button. This simply checks to see if there is any alteration in the number of files in the folder the Log images are store. If there is, a new object is created inside the Log menu and the image is added as its texture so the user can identify what specific image is being represented in the Log.

Below the "Update Log" button, the "Reset Canvas" button, which, as the name implies, resets the drawing canvas to its original empty state. This allows the user to fully clear the canvas in one go, instead of having to painstakingly erase everything with the eraser.

Finally, the ".obj" button. If pressed, the current terrain on display is stored to the file system as a .obj format file that can be later opened with any application that supports that file format and further worked on and modified. This export feature is important because it allows users to not be limited to this single application. Exporting allows users to export the terrain to another application and, for

example, add textures or any other 3D object they wish. This allows users to integrate any terrain generated here in any project that needs a terrain. Exporting the terrain to *.obj* is also possible by pressing the F12 key on the keyboard. Table 3.2 summarizes what each action button does and their corresponding image.



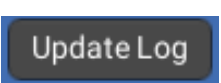
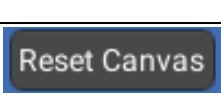

Button image	Button function
	Play button, responsible for initiating the NN with user's sketch as input.
	Generate button, responsible for updating the terrain after the NN finishes processing the user's sketch.
	Update Image Log button, responsible for updating the image log with the new user's sketch and resulting terrain preview.
	Reset canvas button, cleans the entire canvas of any lines.
	Export button, exports the terrain to <i>.obj</i> file.

Table 3.2 - Action button table.

In the bottom-right side are the drawing tools, labeled with the number 5 in Figure 3.12 and a detailed view can be seen in Figure 3.14. In the top row of Figure 3.14, we have three buttons representing the original colors the NN was trained on, plus an eraser that can be used to erase anything drawn on the canvas, mainly used to erase errors or lines that are no longer useful, opposing the "Reset Canvas" button. In the bottom row of Figure 3.14 are the three new colors added by us and a button with a question mark, which opens an in-depth tutorial of how the application works, ranging from a simple explanation on how the NN works, what kind of inputs are expected to what each color does and how to control the map visualizer. Any doubts the user has regarding the application are addressed in the tutorial.



Figure 3.14 - Interface detail: drawing tools.

Each color button was assigned a minimalistic design of what they produce in the context of the application, as explained in Section 3.3: Both reds draw elevation but dark red outputs more altitude constant mountain ranges when compared to red. Green outputs a singular elevation point, as opposed to a whole mountain range, so it has one single triangle to try to convey that. Dark green

produces a more varied altitude range and its always the biggest elevation on the terrain, so we gave it a big triangle on the front and smaller ones on the back. Finally, Blue, responsible for depressions on the map, has a minimalistic looking depression on its button, and Light Blue, as a slightly less pronounced depression to indicate that it outputs more shallow depressions.

Finally, there are two sliders between the action buttons and the drawing tools. The bottom one, as the name implies, further modulates the mesh height by multiplying a value with the Z axis of each vertex on the mesh, giving the user a more direct control of the mesh height. Lastly, the second slider, “Eraser Width” controls the size of the eraser, to facilitate erasing bigger parts of the drawing faster.









Button image	Button function
	Outputs variable elevations. Meant to be drawn as a line
	Outputs elevation cues, elevates an area around where it was drawn. Meant to be drawn as a single dot.
	Outputs depressions on the terrain. Meant to be drawn as a line.
	Outputs elevations that are more constant trough their length. Meant to be drawn as a line.
	Outputs more shallow depressions on the terrain when compared to dark blue. Meant to be drawn as a line.
	Outputs elevations that have more height variation than red. Elevations done with dark green will always be taller than those drawn with red. Meant to be drawn as a line.
	Eraser with fixed size. Used to erase sketches partially.
	Opens the tutorial.

Table 3.3 - Drawing tools summary.

3.6 Interface version 2

Although this version was only developed after testing with users, in part prompted by some results, which will be further discussed in Chapter 4, we are showing it now. As we can see in Figure 3.15, the flow for getting a terrain on the terrain visualizer is simpler. The two intermediate buttons to update the mesh and the Log were removed. These two buttons just contributed to add unnecessary complexity and created some confusion. The user only has to press the Play button, and everything will be automatically done, from displaying the generated terrain to auto updating the Log, no extra

clicks necessary, just the one. This simplification translates in better results as will be seen in the next chapter, were results for testing will be presented.

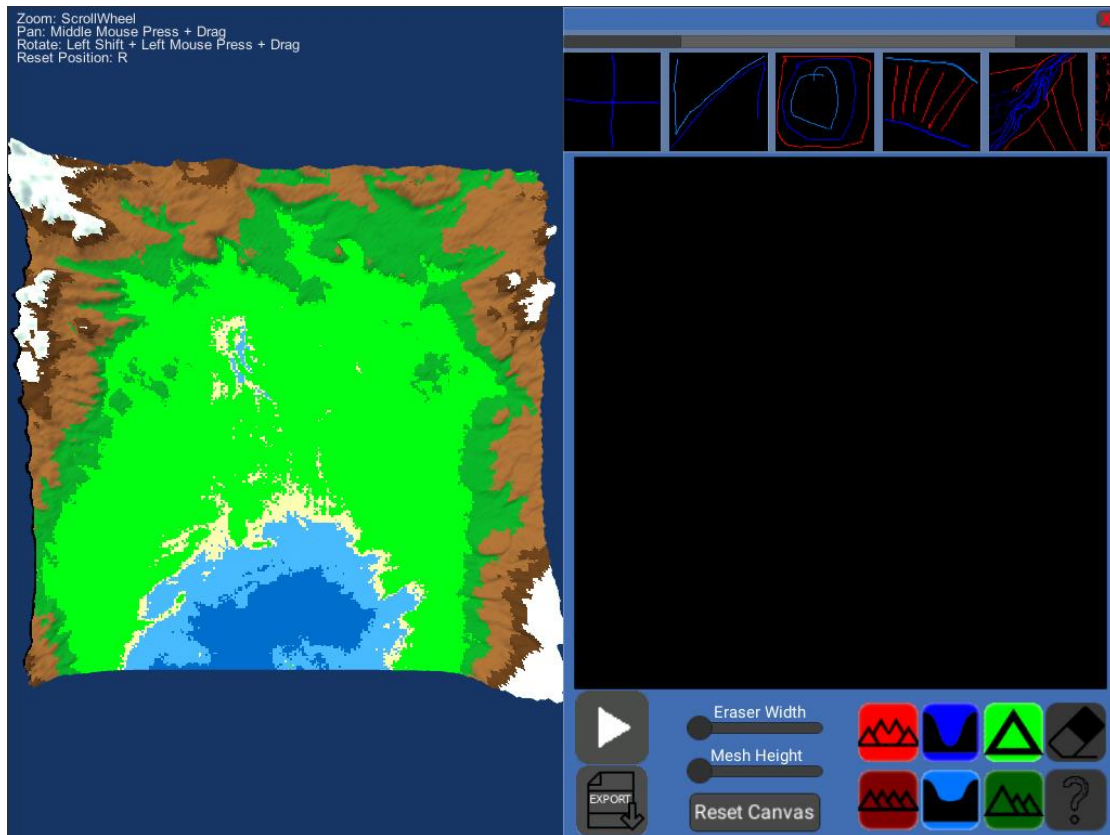


Figure 3.15 - Version two of the Interface.

The main reason for these modifications were the results obtained in the first Usability test results (detailed in Section 4.4). It became apparent that the UI was not refined for the final user, instead closely following the logic and the task sequence present in the subjacent code classes. This resulted in a more complex and cluttered UI than desirable, in particular for first-time users. When implementing the new changes, it was necessary to figure out a work-around for an unforeseen problem: the event system handler regarding processes not working on Linux the same way it does on Windows. Since .NET is developed for a Windows environment and not Linux, and even though it works on Linux, some internal Classes do not react the same way to Linux process event triggers as they do in Windows. Finding a way to fire all the other methods responsible for updating the interface contents once the NN finished running allowed us to effectively act upon the results obtained from the tests done with the first interface version and improve upon our design.

Other changes include the positioning of the color buttons and the “Reset Canvas” button and an increase in size of the “Export” button. Thanks to the gained space from removing the unnecessary clutter, we were able to make the button bigger and much more obvious regarding function. Further explanations will be given once we go through the test results in the next chapter.

3.7 Image Log and Copy/Paste

In order to allow users to have more creative options and more leeway when creating terrain, it was decided to add more options to the workflow of the application alongside the ones previously presented.

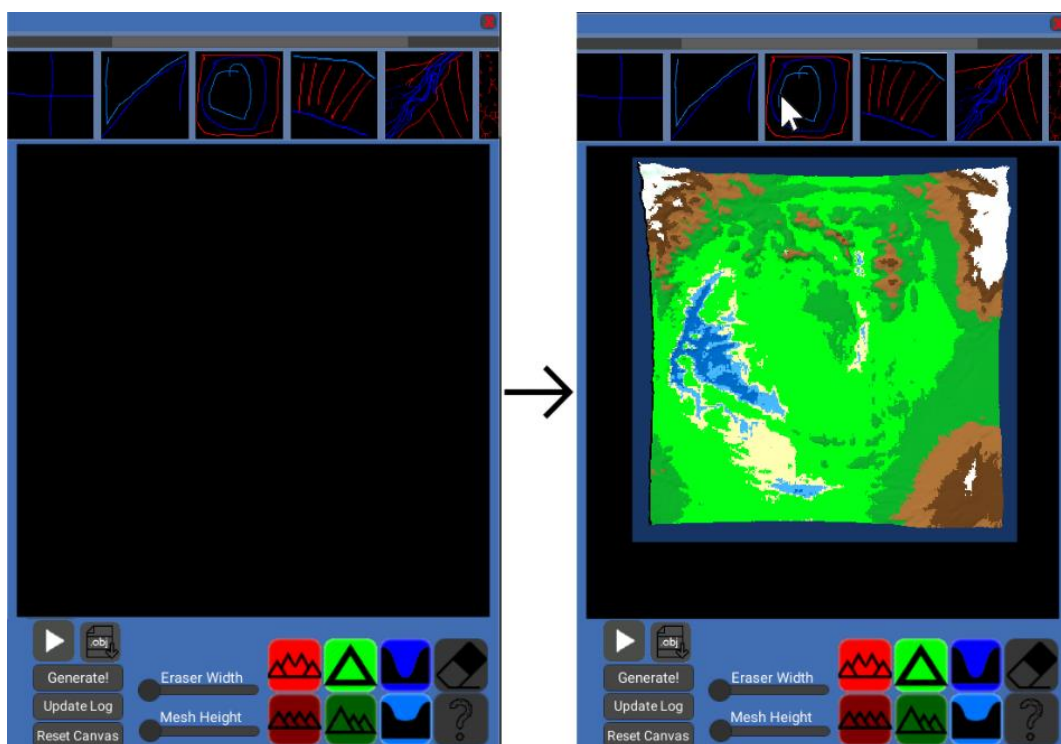


Figure 3.16 - At the top of the interface, Log menu, with generated terrain preview

To enable a more creative design process, something that kept track of what the user did while using this application was necessary. An image Log was the final choice, shown in Figure 3.16, instead of the more traditional History, like for example what Photoshop has. The intricacies of the drawing made with this application are not that detailed to the point that such a fine control is justifiable and in case of a NN crash, the user drawing is salvaged and work can be resumed easily.

In the course of using the application, every drawing that is sent to the NN to be processed is stored in the file system of the machine the tool is running on and accessible through the Log menu, alongside a preview of the generated terrain, which is shown when the user hovers their mouse over

each specific drawing in the Log. This allows for a more complete preview as the user can see the drawing and its resulting terrain. Additionally, if the user clicks on one of the stored inputs, that input is reloaded to the drawing area and can be either regenerated or it can be altered and then generated with the new addition, facilitating iterative work on chosen terrains and reutilization without having to redraw everything.

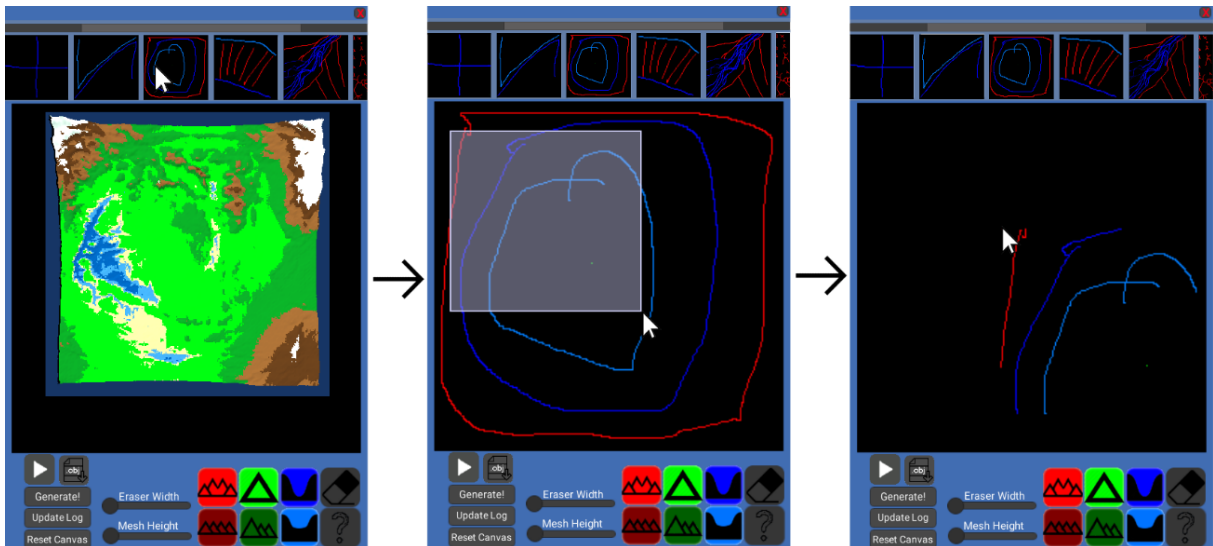


Figure 3.17 - Copy/Paste loop

Another feature that allows for more leeway when sketching is the ability to copy parts of the stored input onto the current drawing. Instead of regenerating a previous drawing, the user can opt to copy an area of the chosen drawing by right click and drag to select and pasting that section of the drawing in their current drawing area. This facilitates the extraction and incorporation of design features of previous input in the current drawing, as can be seen in Figure 3.17, further facilitating iterative work. All these features closely resemble the ones found in traditional paint style editors, features which most users will be somewhat familiar with. This is expected to promote and facilitate reusability of drawing and drawing iteration without alienating less terrain editor tool savvy users.

3.8 Perlin Noise generator

Although not accessible to the end user, the developed tool can generate Perlin Noise heightmaps. This is possible through the Unity3D editor, as seen in Figure 3.18, and it is used as a comparative method. This enables us to compare the solutions generated by the NN with a method that is more widely used and maintain terrain presentation style. Having to style variation helps by reducing subjective variability because the coloring style or texturing style were different in both methods. A quick overview of the noise parameters is necessary to better understand the variables in play that allow us to elevate the quality and controllability of the generated noise.

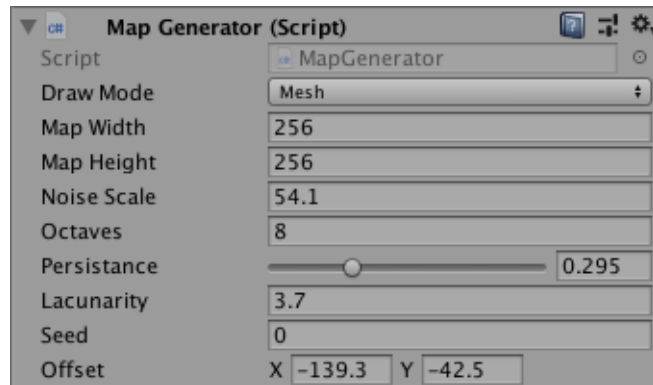


Figure 3.18 - Perlin Noise editor fields.

The first parameter field is Noise Scale, that serves to change the scale of the noise, at what distance we are seeing the noise. Octaves control the number of layers that are generated and then added to themselves, so we can have more detail in the noise sample. Persistence controls how much each octave contributes to the final noise output, it controls the amplitude reduction of each octave. The bigger the persistence the more the lower layers of octaves influences the top ones. Lacunarity is a frequency multiplier, meaning it influences the distance between octaves, resulting in a control over the details that are added to each octave. The Seed value is used so that we can regenerate any given noise map. And finally, offset is used to simply tile the noise manually.

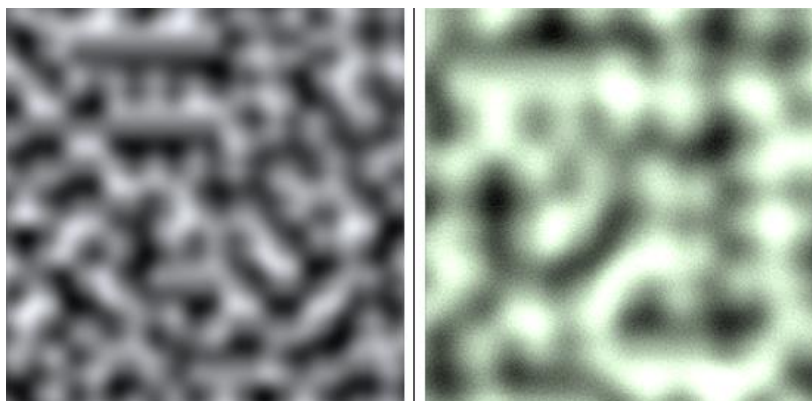


Figure 3.19 - Perlin Noise comparison. Right image is a sample of noise generated by *Mathf.PerlinNoise*. Left noise sample is from the developed application.

Unity3D has a basic Perlin noise generator, but the resulting noise is far too simple to be considered a quality candidate to generate heightmaps, due to not having any controls over the noise aside from scale and coordinates. All these fields and variables are all transformative, they use the base value calculated by *Mathf.PerlinNoise* ("Unity - Scripting API: Mathf.PerlinNoise", 2020) and the added octaves and the control over the influence, distance and persistence of the noise layers attributes allows us to have a much bigger control over the generated noise and a better quality when compared to the vanilla generator included in Unity3D. This difference in detail can be seen in Figure 3.19, where the right side presents a lot more variation in the noise structures, with more clearly

defined lighter areas and smoother transitions to the darker areas. This added noise detailed translates in more detailed 3D terrains when compared to vanilla Perlin Noise, and example of which can be seen in Figure 3.20.

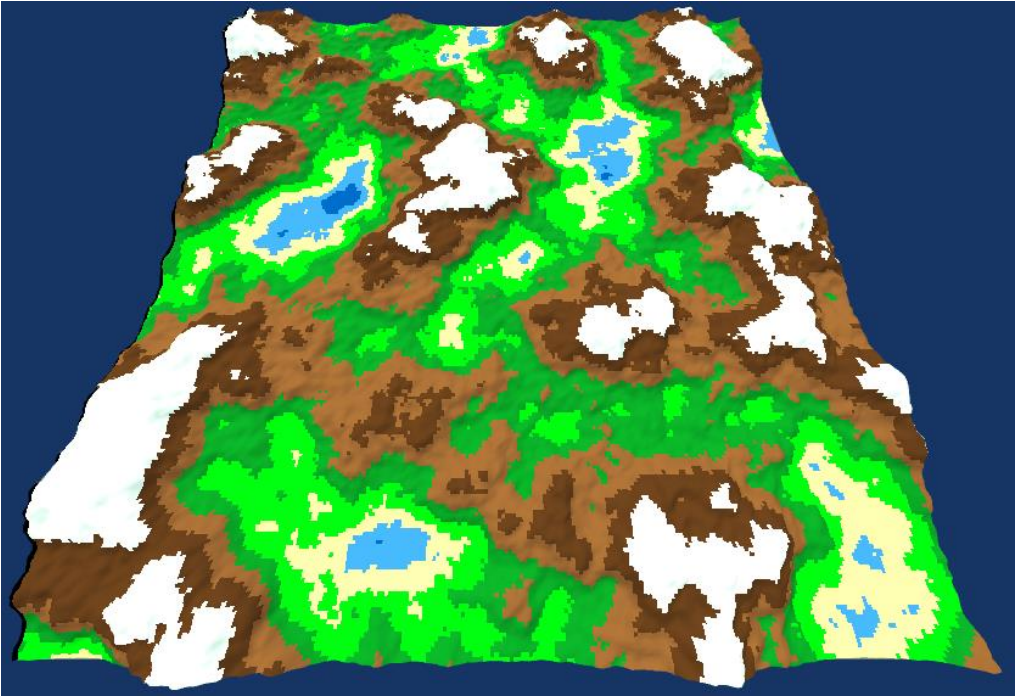


Figure 3.20 - Perlin noise terrain example. 3D terrain generated using Perlin Noise instead of an heightmap generated by the NN.

Experimental Results

In this Chapter, the testing methodology and the obtained results will be detailed. In total, three tests were carried out across two phases. The first test conducted was an online survey, with 113 participants, was used to compare terrains produced by the developed tool, using the NN as the generator, with terrains generated with a traditional alternative, i.e., Perlin Noise. Following the online test, a test with 30 participants was conducted to assess the tool's usability. These users utilized the developed tool in a guided test to ascertain its usability and intuitiveness.

After altering the interface according to the results obtained from the first testing phase, a second testing phase was initiated. The goal was to assess the impact of the introduced changes. For this phase, a smaller participant pool, with 5 individuals, was considered and the usability test was conducted again to evaluate the changes done to the UI.

4.1 Terrain Generation Comparison Test

This first test aimed to understand the perception user might have regarding the realism of the terrains produced by the NN. The goal was to assess whether users of the developed tool would find the resulting terrains to be pleasant to look at, and more importantly, if they were, morphologically speaking, more realistic when compared to another widely used method, namely, Perlin Noise. This test was carried out using thirty randomized image pairs, each pair composed of a terrain generated with the NN and another one generated using Perlin Noise. An example of one of these pairs can be depicted in Figure 4.1.

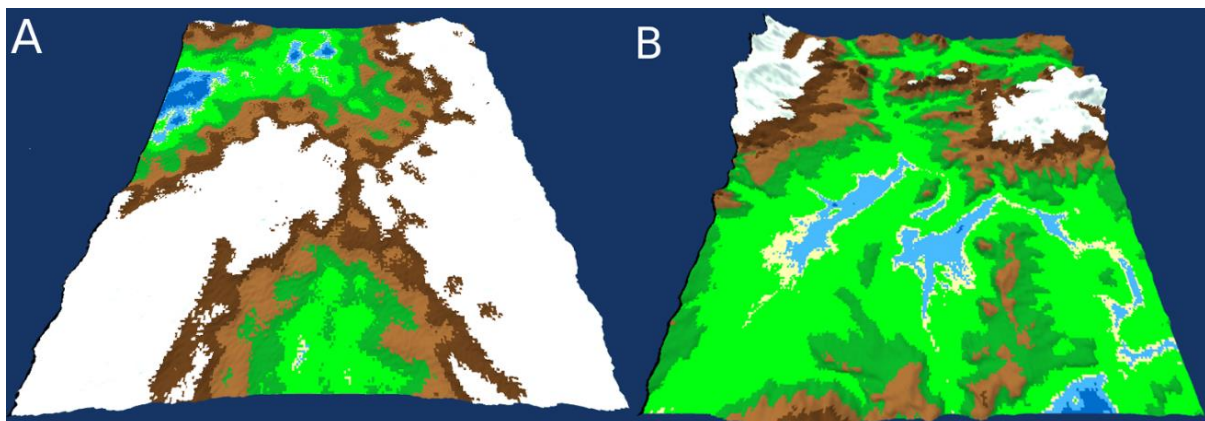


Figure 4.1 - Example of terrain pair used in the Comparison Test, where A is a PN terrain and B is a NN terrain.

Terrains generated with Perlin Noise were parameterized with a wide variety of values to obtain a broad representation of all types of terrain this method can output. For the terrains representing the

NN, we chose thirty random inputs from the testing dataset and used them as inputs for the NN and rendered the resulting heightmaps using the herein presented. Terrains were then randomly joined to form a pair and to avoid choice patterns and inserted in the online questionnaire at random. This was done to avoid any choice pattern (e.g., all NN terrains being the A option).

The online questionnaire was hosted in Google Forms and the link to access it was disseminated in two online communities dedicated to ML: *r/neuralnetworks* and *r/learnmachinelearning*, both Reddit communities. The rationale for this decision stemmed in part from related work in this field, which also used these and similar platform to reach a testing audience with success and because there was a bigger chance for better engagement and hence, more responses, with an audience that is already interested in these topics (Spick et al., 2019). Users were given no information beyond that the application was a terrain generator and a comparison between two methods was being evaluated for perceived realism. They were also made aware that there were no correct or incorrect answers. Please refer to Appendix B for a detailed presentation of the thirty generated terrain pairs.

4.2 Usability Test

The second test had as objective gauging if users managed to understand the function associated to each element of the interface, without any explanation or previous knowledge regarding the application, and, assessing if users were able to use the application in a productive fashion, meaning, if they were able to use it in order to express their design wishes. This test was worded and organized in a way that mimics a real session with the application. Instead of being asked direct questions, the user was placed in a situation mimicking a real-life session and asked what buttons they would click, or which actions would they perform to solve the current situation they are being placed in. Each subsequent question follows a flow that mimics a real-life session. Every user action was timed and registered. This test was based on the testing done by Guérin et al. to users, as well as on the work of Dix et al. whose works focus on how to deliver efficient experimental methods involving user participation (Dix, Finlay, Abowd, & Beale, 2004). General guidelines on how to apply the tests were based on the work of Lewis et al. (C. Lewis & Rieman, 1993).

The test was divided in five phases. In Phase zero, the participants were briefed with a very basic notion of what to expect. We started by explaining what the application was used for and how an input sketch, drawn by the user, could look like, how a generated terrain might look like and where the user will be allowed to draw and where the terrain will show up. No information regarding any button or feature was given in this phase.

Phase one served the purpose of determining whether users were able to identify the function of each button. When starting the first group of questions, users were prefaced with what was to be tested, in this case, interface intuitiveness, and that they should imagine they were in the middle of a real drawing session and that questions were to be posed as such. Users were made aware that there were no wrong answers and that they should not feel pressured to answer correctly or quickly. The goal was to ascertain if buttons presenting atomic actions were clear enough to be identified without explanation, as well as slightly more complex actions but still integral to the navigation and usability of the application. As such, participants only had one opportunity to answer the posed question. This phase was composed of thirteen questions in total. Phase one was divided in four groups. Group one tested if users were able to recognize what type of structure each drawing tool button (seen in Figure 3.13) allowed to generate. This group was composed of the following four questions:

- Question 1a): *“Imagine that you want to draw a mountain range, in which button would you press?”*.
- Question 1b): *“Now imagine that you want to draw a depression on your terrain, which button would you choose?”*.
- Question 1c): *“And to draw a single elevation?”*.
- Question 1d): *“Imagine that you made a mistake, and you want to erase part of what you did, which button would you choose?”*.

The second group tested user comprehension regarding the generation flow (see Figure 3.12), such as the ability to determine what buttons and in which order should they be pressed to generate a terrain. This group was composed of the following five questions:

- Question 2a): *“Now imagine that you are feeling satisfied with the drawing you did and you want the drawing in your drawing area to be used to generate a terrain, where would you press?”*.
- Question 2b): *“Now you would like to see, in the visualization area, your terrain. Where would you press?”*.
- Question 2c): *“Now that you have your terrain on the screen, you would like to refresh the image Log menu so that your newly created drawing is stored in the log for future reutilization, where would you press?”*.

- Question 2d): *“Imagine that in the middle of drawing your next terrain, you decided that your drawing wasn’t what you wanted at all and decide to completely clean the drawing area, where would you press?”*.
- Question 2e): *“Finally, you’ve drawn something of your liking and you think the terrain should be stored. In which button would you press to export your terrain?”*.

Group three only has one question, and it tests users in whether they were able to make use of the Log menu and retrieve a specific image from its structure. The question posed to participants is as follows:

- Question 3a): *Imagine that you started a new session and decide to make alterations in the terrain I showed you in the beginning. What actions would you take?”*.

Finally, group four, tested user comprehension about camera control, concretely, if they were able to perform the basic camera movement operations within the developed tool. This group was composed of the following three questions:

- Question 4a): *“You took notice of a detail in your terrain and decide to zoom in on that part. How would you do it?”*.
- Question 4b): *“Now you want to see the same detail, but from another perspective, how would you do it?”*.
- Question 4c): *“Finally, you want to see the map from a top down view, where would you press?”*.

In phase two, participants were asked to read the tool’s tutorial that is present in the developed tool from start to finish. This was done, to mimic to the fullest how a real-world user would interact with the tool. This tutorial explains every aspect of how to operate the application, what each button does and how to use the existing features. This serves to normalize every participant. Even if they were unable to successfully answer previous questions, when moving to the last phase of testing, everyone is on equal footing. Having read the tutorial, users were informed that they were allowed to use the application in a non-supervised fashion for ten minutes. They were free to do as they pleased, including drawing anything, and using any part of the application as they saw fit. This had the objective of

allowing users to organically familiarize themselves with the application, to allow them to experiment with the caveats of NN generation and to explore for themselves what they have just learned.

After the non-supervised practicing, participants moved to the fourth testing phase, composed of seven questions. In this phase, participants were asked to perform actions that mimic a real-life user session, from atomic actions to complete actions, following the entirety of the tool workflow. This served to test if users understood how the application works. They are asked to do complex actions that require understanding of how the application works and a basic understanding of how the NN interprets their input drawings. Just like in phase one, users are timed but with a key difference: they had three chances to complete these actions, although no extra information is given between tries other than if they failed or succeeded. Times for each try were added together to give one single time measurement. Three chances were given because users have a small timeframe to accustom themselves to the tool, and sometimes the NN does not output a 3D terrain that corresponds exactly to the participant's expectations. Participants might sketch an input using the correct design logic but simply because the lines, for example, were not long enough, the output comes out differently than what the test is asking, even though the user had the correct design logic applied to the sketch. It also served to add granularity to the test results: a participant that fails three times in a row for that specific question is different from a participant that fails the first time but got it at the second time, or from a participant that gets it right in the first try. Phase 4 questions are as follows:

- Question 1 Phase 4): *"Draw a mountain range that crosses the terrain diagonally from left to right"*.
- Question 2 Phase 4): *"Draw a valley with mountains surrounding it"*.
- Question 3 Phase 4): *"Draw a single elevation in the middle of the terrain without anything around"*.
- Question 4 Phase 4): *"Position your camera up close to the tallest point of your terrain"*.
- Question 5 Phase 4): Users are asked to clear the canvas before proceeding to this question. *"You decide that you want to add something to your previously created terrain"*.
- Question 6 Phase 4): *"After the added details you conclude that your terrain could be a bit higher and decide to increase its altitude"*.
- Question 7 Phase 4): *"Finally the terrain is to your liking and you decide to export the terrain"*.

4.3 Terrain Generation Comparison Test Results

Figure 4.2 shows the bar graph with the results of the terrain comparison test, in which we had 113 responses for the duration of time that the link was active.

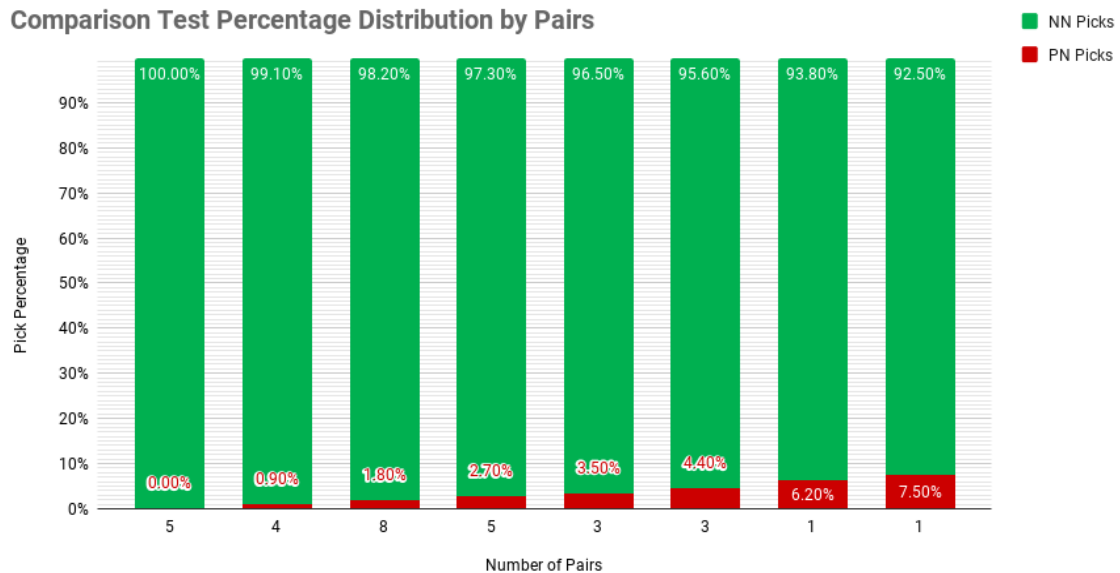


Figure 4.2 - Participant choice distribution between NN and PN by pair

In the horizontal axis, the bar graph has the number of image pairs, out of the thirty, that had the represented percentage distribution of choice between Perlin Noise (PN) and the NN seen in the vertical axis. Green represents the percentage of NN picks and red the percentage of PN picks. To clarify the data shown in Figure 4.2, if we look at the leftmost bar, we conclude that in five of the thirty terrain generated pairs in the test, everybody of the one hundred and thirteen answers picked the NN output as being the most realistic. The graph also shows that in one terrain generated pair, out of the thirty, 92.5 % of people chose the NN output as being the most realistic between the two, leaving a remainder 7.5 % that chose the PN output as the most realistic one.

In Figure 4.2, only two pairs out of thirty have more than 5 % of participants choosing PN as the more realistic option, the remaining 28 pairs are all below the 5 % mark, with the mode being 1.8 % of participants choosing PN as the most realistic case in eight pairs. Nine out of thirty pairs have participants picking PN at a rate below 1 % and out of these nine, five have a 0 % PN pick rate. To put this in total numbers, there was a total of 3390 comparisons, 113 participants responding to thirty questions each, and out of these only 77 responses were PN. This equates to 2.27 % of answers being PN. These results show a clear preference for the 3D terrains produced by the developed tool, showing that, in the majority of cases, the terrains produced by the tool were found to be more realistic looking when compared with the terrains produced by a PN algorithm.

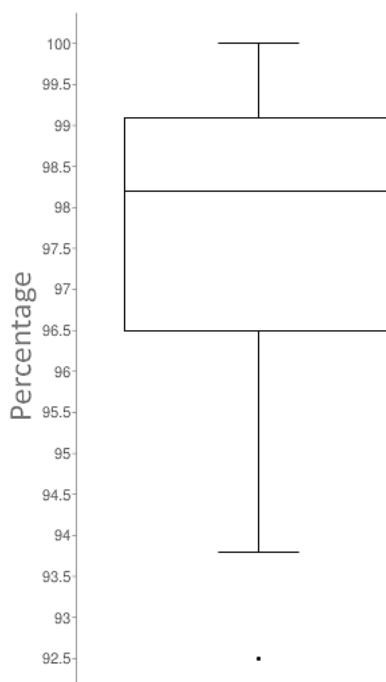


Figure 4.3 - Box plot distribution of NN pick percentage

Figure 4.3 depicts a box plot representing the percentage distribution and skewness of NN picks. Presenting a median of 98.2 %, a mean of 97.7 %, with an outlier of 92.5 %, a minimum of 93.8 % and maximum of 100 %. First quartile presents a value of 96,5 %, third quartile a value of 99.1 %. The total range is 7.5 and the interquartile range is 2.6. Standard Deviation (SD) came at 1.797. In Figure 4.3, we get a better sense of the answer's distribution. The total range of NN picks only varies by 7.5 %, this by itself is already a tight variation, but when looking at the value of interquartile range, obtained by subtracting the third quartile with the first, we have a range of 2.6. This indicates that the bottom 25 % percentile only diverges 2.6 % of the top 75 % percentile, indicating a high level of agreement between participants regarding which option is more realistic. The relative position of the box also leads to another illation. By being more towards the positive end of the axis, and by having a shorter upper whisker, this indicates a stronger agreement and less variation in participants responses at the upper NN pick percentages. This tells us participants are more inclined to choose, at a higher rate, the NN over PN. With this said, it is apparent that the participant base of this questionnaire found the terrain options produced by the NN to be more realistic when compared to PN, produced by the same graphical engine, and found it so with confidence, given the answer distribution found in Figure 4.3.

4.4 Usability Test Results

This section presents the results obtained in the usability test. The complete test applied to the participants can be found in the Appendix C. There are twenty question in total, which were answered by thirty participants, with an average age of 22,5 years old and a gender distribution of 56.6 % being male and 43 % female. Most participants were students at ISCTE-IUL. To our knowledge no one had ML experience. Has for any other related experience, e.g., gaming, we did not ask. We found that having gaming experience would not impact the perception of reality of the terrains. As explained in Section 4.2, Phase 1 is divided in four groups: drawing tool buttons comprehension; generation flow intuitiveness; log menu usability; and camera control. Questions regarding drawing tools and generation flow expect a concrete answer, so they are either correct or incorrect, and the corresponding graph data is distributed by answer given. Question pertaining the log menu and camera control do not expect a concrete answer, rather a small set of actions and key presses, which will be successful or will fail, so graph data referring to these questions is distributed by success or failure. The vertical axis represents the percentage of testers that gave the answers found on the horizontal axis. Both types of data distribution are color coded, being green the correct answer or success and red being an incorrect answer or failure.

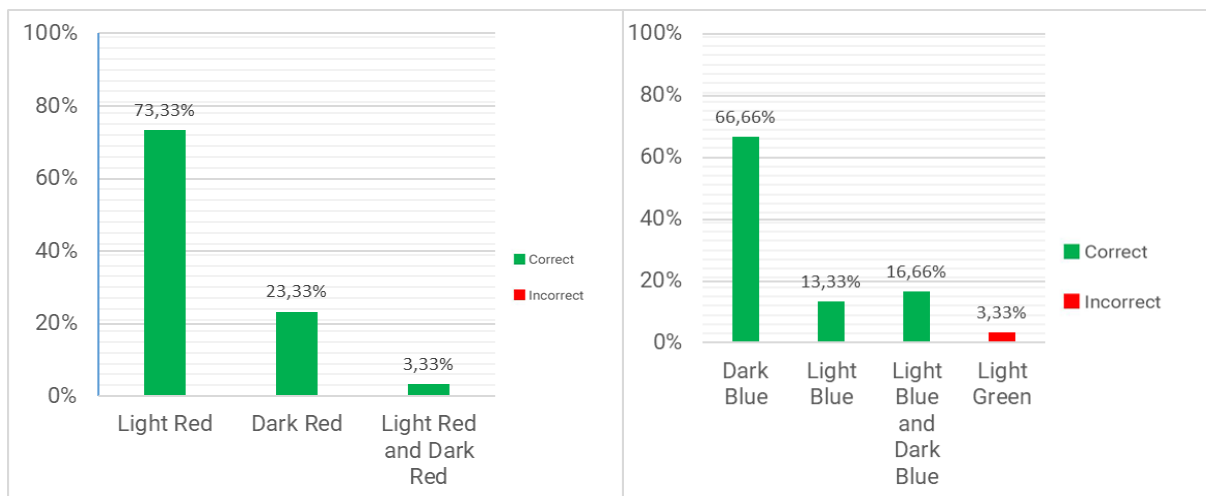


Figure 4.4 - Results for Phase 1 Group 1 question 1a) *"Imagine that you want to draw a mountain range, in which button would you press?"* (left) and Phase 1 Group 1 1b) *"Now imagine that you want to draw a depression on your terrain, which button would you choose?"* (right).

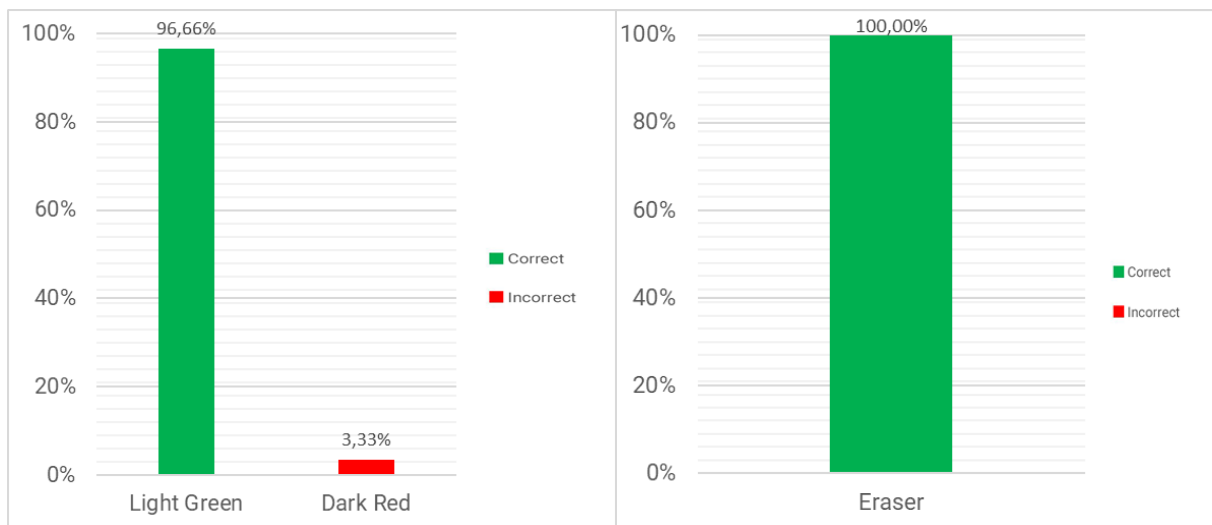


Figure 4.5 - Results for Phase 1 Group 1 question 1c) "And to draw a single elevation?" (left) and Phase 1 Group 1 question 1d) "Imagine that you made a mistake, and you want to erase part of what you did, which button would you choose?" (right).

Starting with the first group of questions, where users are tested about drawing tools comprehension, fairly good results were obtained. In question 1a) (see Figure 4.4) all participants correctly answered Light Red, followed by a smaller sample of participants who answered Dark Red and only one person answered that both could be used to form mountain ranged. These results show that participants had no trouble identifying the button's functionality and it was obvious to them what the button did. In question 1b) (see Figure 4.4) the majority chose Dark Blue as the button they would pick for drawing a terrain depression, but more participants answered that both colors could be used for depression drawing when compared to participants that chose Light Blue. One participant answered incorrectly by choosing Light Green. Even though one participant answered incorrectly, the majority of participants still found it easy to identify the correct button, showing that they had no troubles identifying the correct button. In question 1c) (see Figure 4.5), participants had no problem identifying Light Green as the button to produce single elevations on the terrain, with only one person failing to properly answer, showing that participants found no problems with this button's functionality as well. In question 1d), (see Figure 4.5), all participants positively identified the eraser as the correct answer.

In Table 4.2, we can see the average times for the first group of question. Answer time averages were low, varying between 1,5 seconds (s) to 5,6 s, with SD between 1,15 and 6,64 s. These times imply, as well as the percentage of correct answers, that participants were confident in their answers, were quick to analyze the interface and, with more correct answers than incorrect, the design choices for the drawing tools were successful in conveying their purpose.

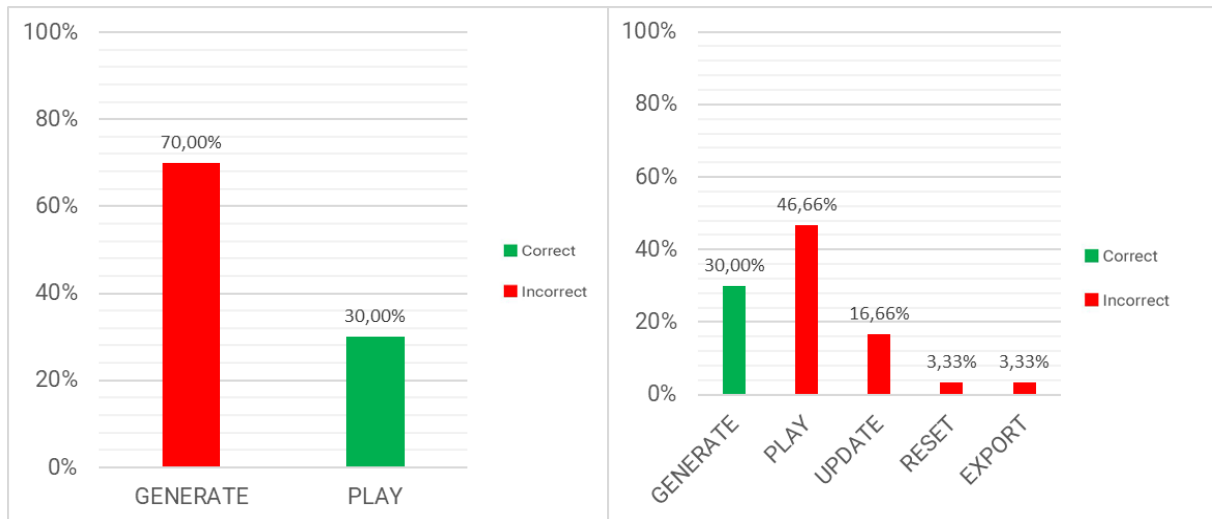


Figure 4.6 - Results for Phase 1 Group 2 question 2a) "Now imagine that you are feeling satisfied with the drawing you did, and you want the drawing in your drawing area to be used to generate a terrain, where would you press?" (left) and Phase 1 Group 2 question 2b) "Now you would like to see, in the visualization area, your terrain. Where would you press?" (right).

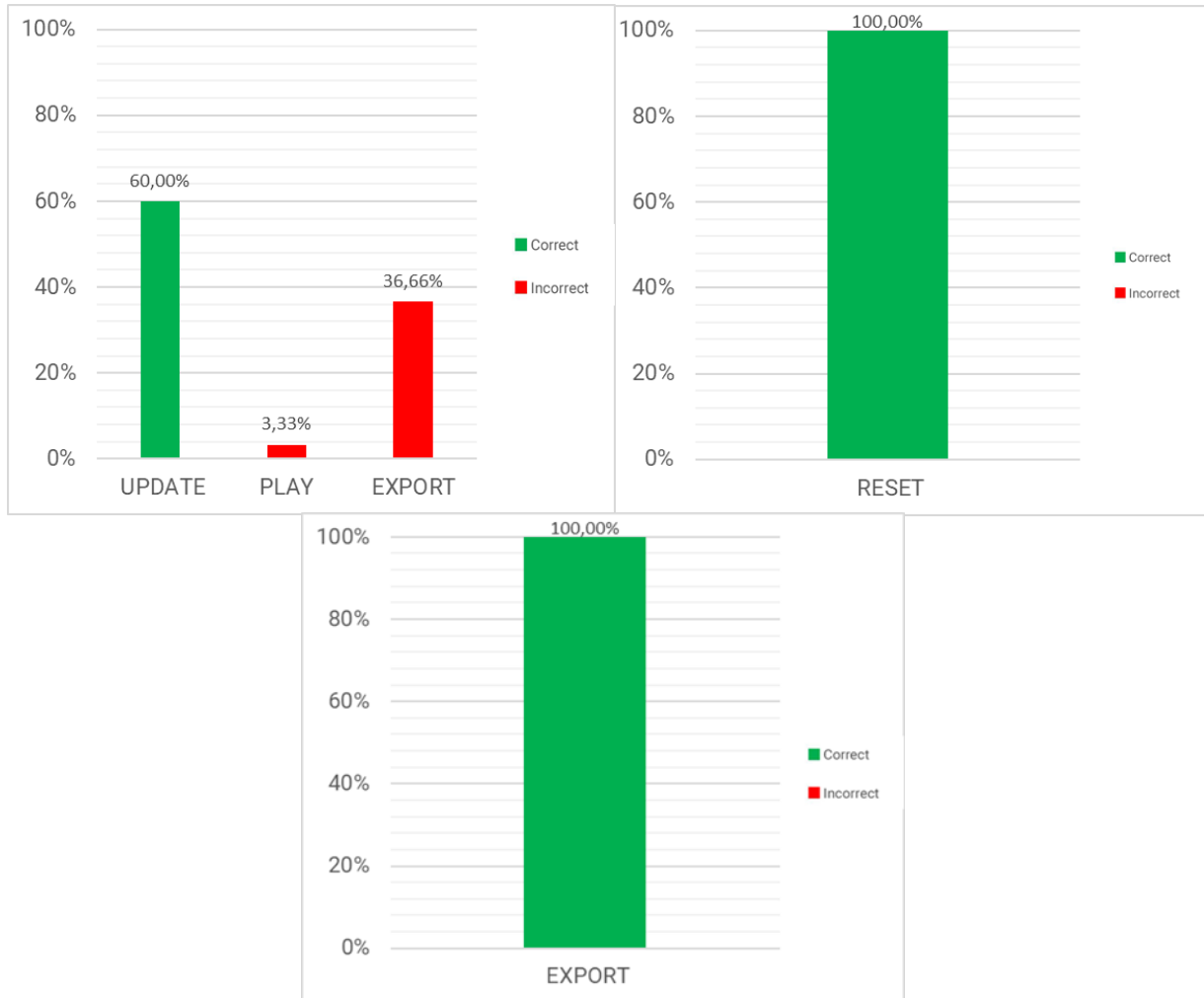


Figure 4.7 - Results for Phase 1 Group 2 question 2c) *"Now that you have your terrain on the screen, you would like to refresh the image Log menu so that your newly created drawing is stored in the log for future reutilization, where would you press?"*. (top left), Phase 1 Group 2 question 2d) *"Imagine that in the middle of drawing your next terrain, you decided that your drawing wasn't what you wanted at all and decide to completely clean the drawing area, where would you press?"* (top right) and Phase 1 Group 2 question 2e) *"Finally, you've drawn something of your liking and your think the terrain should be stored. In which button would you press to export your terrain?"*.(bottom center).

In the second group of questions, where participant comprehension of generation flow is tested, results were mostly negative. In question 2a) (see Figure 4.6), only 30 % of participants answered correctly, showing that participants were unable to properly identify the button's functionality. This leads to question 2b) (see Figure 4.6) also having poor results. Once participants got the first button wrong, most of them also answered incorrectly to question 2b), with the majority switching the order

between the play and the generate button. The data shows that if a participant got question 2a) right, they would also get 2b) right, with both questions having 30 % of participants answering correctly, meaning they understood the order in which buttons had to be pressed from the start or realized they were mistaken and answered 2b) with that in mind. Regardless, what was consistent was participants switching the order between the play button and generate button. Question 2c) (see Figure 4.7), presents some better results but nonetheless far from acceptable, with 36,66 % of participants giving the export button as answer instead of the log update button, once again showing that the buttons are not clear on their intended purpose, leading participants to guess and giving incorrect answers on other buttons. Both question 2d) and 2e) (see Figure 4.7) had all participants answering correctly. Even though 36.66 % of participants answered export in question 2c), all of them realized their error and corrected the answer when asked question 2e). This is due to both questions being easy to understand, using language similar to the names and functions of the buttons in question and, in the case of the export button, having a symbol similar to other application symbols for that function.

When analyzing time averages for group two of questions in Table 4.2, especially the first three questions, it becomes more obvious that participants struggled with this part of the test. Although time averages are higher than the previous group, what further cements the bad results for this group of questions are the Standard Deviation. Question 2a) has a SD of 6.6 s, which is the same as question 1a), but when analyzing SD for 2b) and 2c), the apparent low time averages gain a new interpretation. Question 2b) has a SD of 11.9 s and 2c) of 14.6 s. This reveals that participants were confident when answering the first question, but when realizing they answered incorrectly, they were forced to reassess their assumptions, not only leading to more wrong answers but also a higher Standard Deviation. Time averages and Standard Deviation for 2d) and 2e) are in the expected range, with both questions getting a low average time and an even lower SD time. This group of question revealed a clear problem with the interface: having these many buttons to press just to be able to put the generated terrain in the screen, as well as have all the UI's components updated, is detrimental to the understanding of the interface. Adding to that, the nomenclature used to identify these buttons is not the clearest possible. With such a complex button press sequence, naming should be clearer to help guide the users with the need of explicit training

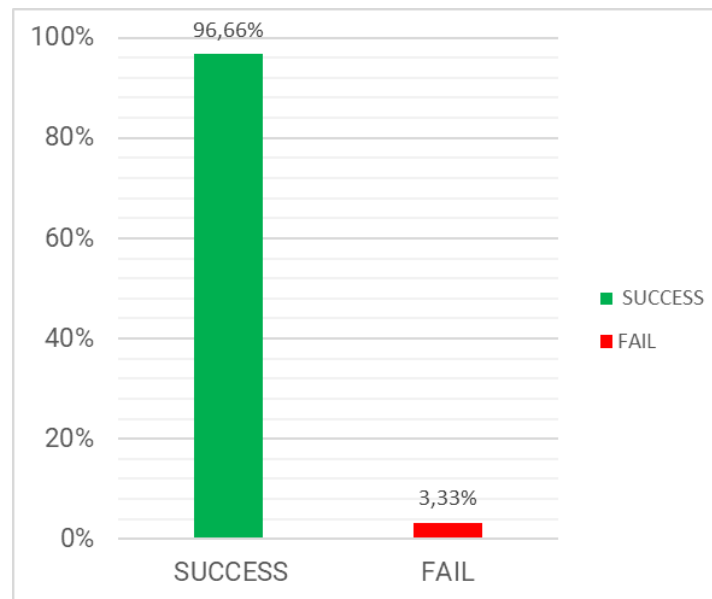


Figure 4.8 - Results for Phase 1 Group 3 Question 3a) *“Imagine that you started a new session and decide to make alterations in the terrain I showed you in the beginning. What actions would you take?”* .

Group three only has one question, 3a) (see Figure 4.8), and it is regarding the image log. Participants were asked to retrieve a specific sketch from the log menu and only one participant was unable to do so (the participant gave up 17.6 s after starting). Average time was 40.6 s (SD = 34.4 s), as presented in Table 4.2. Participants had some difficulty finding the asked for image. Reasons for this high average time might be due to participants not being able to immediately understand how the image log worked but also having trouble visualizing the small details of the sketch representation inside the log menu. Some users understood what was asked but clicked through all the images in the log before finding the one they wanted. This indicates that the feature either needs to be better placed and modified in form but also that details of the sketches inside it need to be more noticeable.



Figure 4.9 - Results for Phase 1 Group 4 questions 4a) "You took notice of a detail in your terrain and decide to zoom in on that part. How would you do it?" (top), 4b) "Now you want to see the same detail, but from another perspective, how would you do it?" (bottom left) and 4c) "Finally, you want to see the map from a top-down view, where would you press?". (bottom right).

Moving to the fourth group of question of phase one, participants were tested regarding camera control. Both question 4a) (see Figure 4.9), had 100 % of participants being able to execute the necessary controls. There was only one participant that failed in question 4b), bottom left Figure 4.9. Looking at the times for each question in Table 4.2, time averages and Standard Deviation were high, 44.5 s (SD = 38.5 s) for question 4a) and 34.8 s (SD = 28.1 s) for question 4b). Given that camera controls were displayed in the screen at all time, it is safe to assume that they were not evident enough to the participants, that some key combination might be unintuitive and that the questions might not be well formulated or obvious enough. As for question 4c), with an average answer time of 2.4 s (SD = 3.1 s), participants found no trouble in utilizing the correct key to reset the terrain position in the visualizer. Note that this information was on the same place as the other camera controls.

Before presenting Phase 4 results, a small note regarding graph interpretation: since users were given three chances to achieve the wanted goal, each chance could be either a success (S) or a failure (F). For example, a user that failed twice but had success on their last try, would have their result shown in the graph as F|F|S. There is a special case, which initially was going to be discarded, but it occurred enough times to warrant an entry in the data. S* means that the tester had success in the previous try but explicitly asked to retry the entry because they thought they could do better.

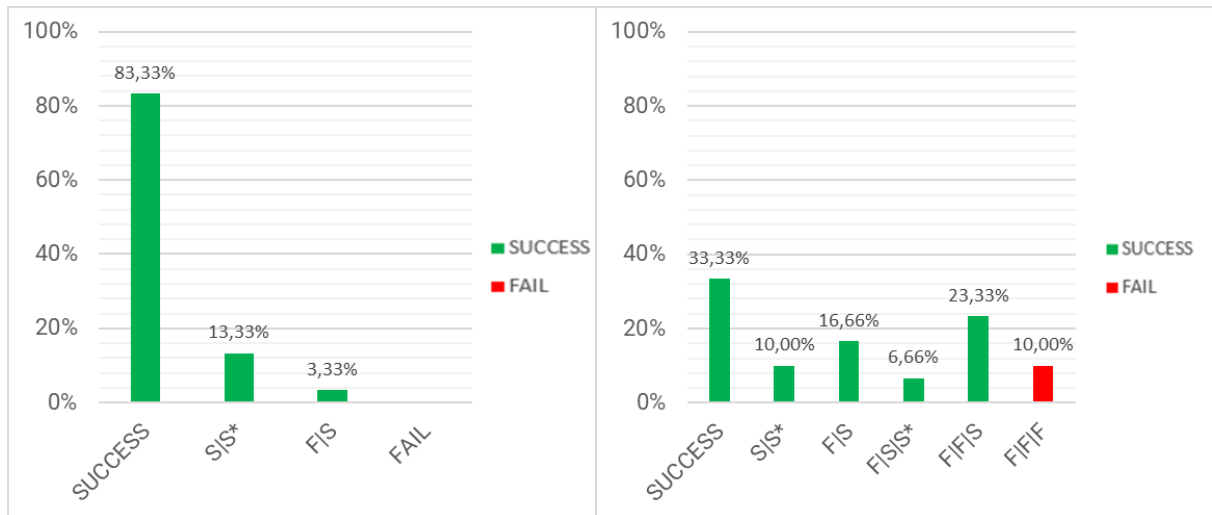


Figure 4.10 - Results for Phase 4 question 1 “Draw a mountain range that crosses the terrain diagonally from left to right” (left) and Phase 4 question 2 “Draw a valley with mountains surrounding it” (right).

After reading the tutorial and interacting freely with the tool, participants were initiated in the fourth phase of testing, which aimed at understanding if participants grasped how the tool worked in all its workflow and if they were able to materialize, in a terrain, what was described in the question. Starting with Phase 4 question 1 (see Figure 4.10), in which only one participant failed on the first try, while 83.3 % of participants were able to successfully complete the task on the first try. The remaining 13.33 % of user were able to also complete the task on the first try but felt like they could achieve a better result and asked for another try. In Table 4.3, question 1 has a 29.8 second average answer time (SD = 12.9 s). This time accounts for the entire process, from sketching, clicking on the appropriate buttons to initiate the NN and waiting for it to finish. Participants seemed to have little problems with this question, being able to produce the wanted terrain quickly. The vast majority on the first try and with a SD that shows low variability between participants. Since only one participant failed on the first try, average tries are nearly one and its SD is negligible.

In Phase 4 question 2, (see Figure 4.10), 10 % of participants did not manage to complete the task after three tries. A total of 33.33 % succeeded on the first try, plus 10 % that succeeded on the first try but asked to repeat the task because they were not satisfied with the result. From all participants,

16.66 % failed on the first try but succeeded on the second try, plus 6.66 % that failed on the first try, succeeded on the second but felt unsatisfied with the result and asked to repeat. A total of 23.33 % of participants required two attempts before succeeding. This gives a total of 43.33 % of participants that succeeded on the first try, 23.32 % that succeeded on second try, 23.33 % that succeed only at the third try, and 10 % that failed completely.

Looking at Table 4.3, there is a much higher time average and Standard Deviation comparatively with the previous question, 69.8 s (SD = 39.1 s). On average, participants required 1.90 tries (SD = 0885) to answer correctly. This question was more complex than the previous one. The terrain asked to draw was more complex and required more lines to achieve a proper result than the sketch required to achieve success in the first question. Participants clearly had more difficulty on this question comparatively with the previous one but still only 10 % did not manage to succeed. Part of the reason this time average is so much higher than question one can be in part attributed to the processing time. It is not instantaneous and since participants required more tries in general to achieve success, this will eventually add up. Needing more tries to succeed also led to a higher SD, since each try time is added to form a total time. On a general way, participants managed to achieve the desired terrain. This shows that even if unable to achieve success on the first try, the tool is understandable enough to allow users to siphon knowledge from their previous try and modify their approach on the second try, and in fewer cases, even on the third try and still achieve success.

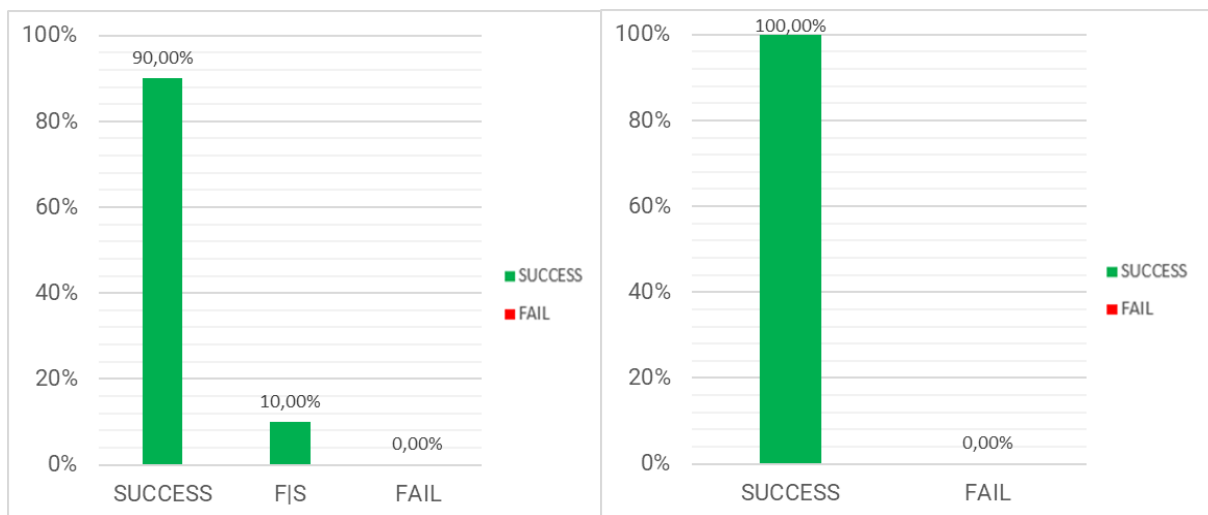


Figure 4.11 - Results for Phase 4 question 3 “Draw a single elevation in the middle of the terrain without anything around” (left) and Phase 4 question 4 “Position your camera up close to the tallest point of your terrain” (right).

Moving on to question 3 (see Figure 4.11), 90 % of participants only needed one try and only 10 % required a second try. No one failed completely. Participants were fairly quick to finish the task, with an average answer time of 24 s (SD = 18.2 s) as presented in Table 4.3. On average user needed 1,1 tries (SD = 0.305 tries), according to table 4.3. These values indicate that participants had no trouble in succeeding in this task and were confident in how to do it. The sketch required to succeed on this question was extremely simple but required from participants a full understanding of how the original green button worked. Any sketch with anything other than a green dot on the middle was not acceptable, and only 10 % of participants were unable to achieve it on the first try.

In Phase 4 question 4 (see Figure 4.11), none of the participants failed. Every participant was able to maneuver the terrain camera in order to place it in the asked position on the first try. In Table 4.3 it is possible to see that the average time for this task was 23.5 s (SD = 17.7 s). Participants already had experience with the camera control in question 4a) and 4b), where the average time and SD were higher than in this question. This is indicative of participants struggling initially with camera controls simply because they did not know what button combinations were supposed to be pressed, instead of inherent difficulty. This highlights that the camera controls present in the top left corner of the interface, seen in Figure 3.14, are not obvious enough to be pick up immediately by participants.

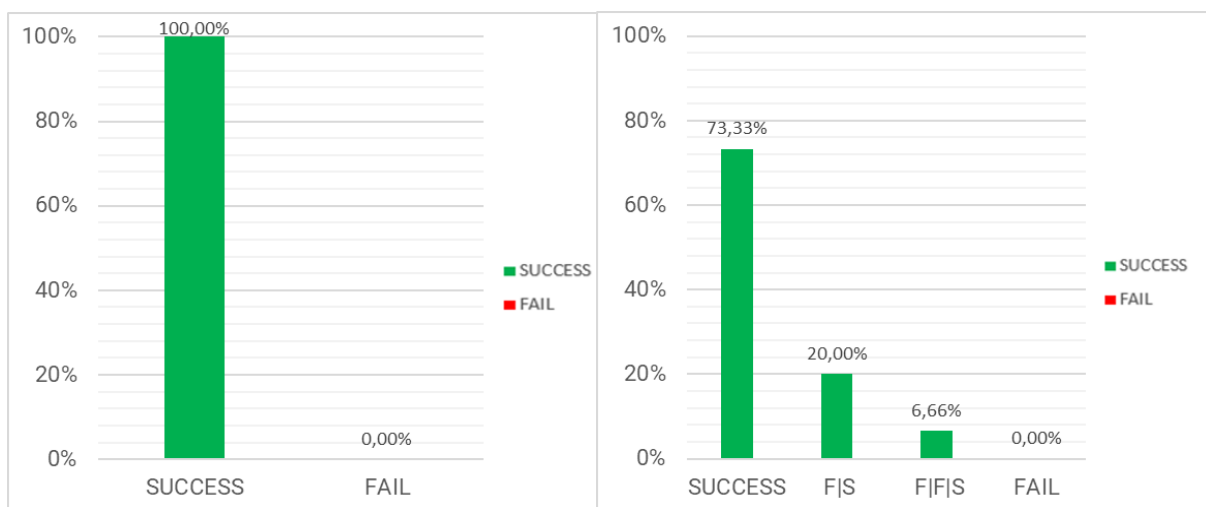


Figure 4.12 - Results for Phase 4 question 5 “You decide that you want to add something to your previously created terrain” (left) and Phase 4 question 6 “After the added details you conclude that your terrain could be a bit higher and decide to increase its altitude” (right).

Phase 4 question 5 (see Figure 4.12), is another question where no participants failed. This question served to retest participants regarding the image log usage. Time average was slightly higher comparatively with question 3a), averaging at 50.6 s (SD = 41.38 s), as per Table 4.3. Considering that users had to slightly alter the image retrieved from the log menu and generate a terrain, the values for this question were on par with question 3a), highlighting that the problem might actually be because of the feature itself and not because of the participants unfamiliarity with it. Further testing is needed to pinpoint the exact issue with the feature.

In Phase 4 question 6 (see Figure 4.12), 73.33 % of participants succeeded in the first try, while 20 % needed two tries and 6.66 % needed two tries to achieve success. It is possible to see in Table 4.3 that the time average was 28.2 s (SD = 40.0 s). Average tries for this question are 1.37 (SD = 0.6 tries). The task needed to succeed at this question was simple, it only required participants to move the mesh height slider to change terrain height. Given this time disparity between participants and relative easiness of the task, there is only one conclusion possible: the slider's name is not clear. Throughout the test the terrain generated is never addressed as "mesh" expect on the mesh height slider. This is unintuitive and confusing for the participants, that might not even know what mesh means.

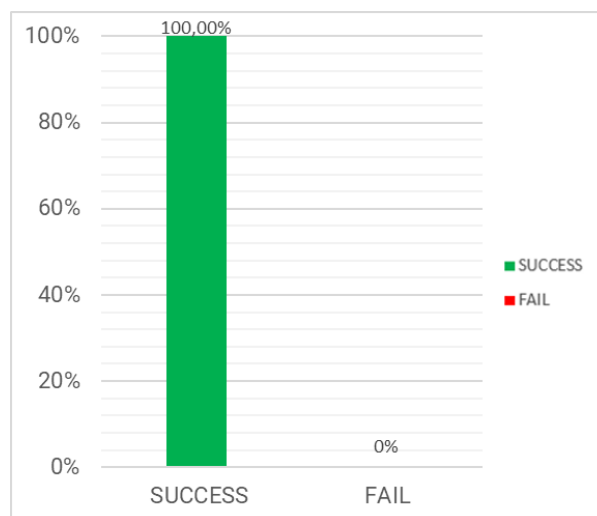


Figure 4.13 - Results for Phase 4 question 7 *"Finally the terrain is to your liking and you decide to export the terrain"*.

Finally, Phase 4 question 7 (see Figure 4.13), none of the participants failed or required more than one try to achieve success. Answer time average was 2,9 s (SD = 2 s), as per Table 4.3, meaning all participants were confident on what do to achieve what was asked of them and that the export button is clear on its function.

Adding to the graphs and timetables for each question, there is also the frequency of which participants used the tool’s features. This information was retrieved while participants were performing tasks. In terms of extra colors, 19 out of 30 participants used extra colors throughout the test. Dark red was used 28 times, with 16 different users using it. Light blue was used 16 times, with 10 out of 30 participants using it. Dark green was used 6 times by 5 out of 30 participants. The image log was used by everyone but only 4 participants used it outside of the required question, for a total of 8 extra uses. The copy and paste feature was used only by one participant and in the training session. The eraser was used by 8 participants, with a total of 11 times. The reset canvas button was used by 29 out of 30 users, for a total of 70 uses. This information is summarized in Table 4.1.

Implemented features	Unique participant uses	Total uses
Dark red	16 out of 30	28
Light blue	10 out of 30	16
Dark Green	5 out of 30	6
Image log	4 out of 30	8
Copy/Paste	1 out of 30	1
Eraser	8 out of 30	11
Reset canvas	29 out of 30	70

Table 4.1 - Implemented features usage by participant and total uses

Going through these test results, one thing becomes evident: the major flaw in this tool is the generation flow. The amount of button presses necessary to get a terrain on the screen and update all structures is detrimental to users experience and tool usability. The tool’s intuitiveness can be improved by reducing the number of clicks to reach the desired result. Aside from that, the majority of participants had little to no trouble identifying what the drawing buttons did nor had trouble with the log menu, although, improvements are necessary to reduce visual clutter and make the log menu functionality clearer. As for extra colors, there seems to be use cases for them, as participants found uses for them throughout this test. The same cannot be said for the copy paste functionality. In fairness, there were not many moments in which copying a piece of another sketch was strictly necessary or made sense. Perhaps in a real setting, this feature would be more used, but for now it seems to be seldomly used. In phase 4, most participants were able to produce the asked terrains and were able to express their design ideas without problems, indicating that participants understood how the tool works. These results, especially the results regarding generation flow, prompted a revision of the UI (detailed in Section 3.5.1) and subsequent smaller batch of tests to verify if changes had an impact on the problematic areas.

Table 4.2 shows the average times for each individual question of the four groups of questions in phase 1, calculated from each participant time to answer the question, starting from when the tester ended the question, as well as Standard Deviation (SD). Considerations regarding these times are done near the appropriate graphical information for each question.

Question	Response Time (s)	
	Average	Standard Deviation
1a)	5,6	6,643
1b)	3,1	2,034
1c)	2,8	2,305
1d)	1,5	1,154
2a)	5	6,611
2b)	8,1	11,945
2c)	8,3	14,67
2d)	1,8	1,282
2e)	1,9	1,356
3a)	40,6	34,442
4a)	44,5	38,539
4b)	34,8	28,124
4c)	2,4	3,151

Table 4.2 - Phase 1 Questions Average Times and Standard Deviation

Table 4.3 shows the time averages, their corresponding standard deviations as well as average tries for each question and their SD for the question in phase four of test one. Considerations regarding these times and deviations are done near the graphical results.

Question	Response Time (s)		Tries	
	Average	Standard Deviation	Average	Standard Deviation
Phase 4 Question 1	29,8	12,948	1,03	0,183
Phase 4 Question 2	69,8	39,105	1,90	0,885
Phase 4 Question 3	24,0	18,238	1,10	0,305
Phase 4 Question 4	23,5	17,702	1	0,000
Phase 4 Question 5	50,6	41,381	1	0,000
Phase 4 Question 6	28,2	40,933	1,37	0,615
Phase 4 Question 7	2,9	2,059	1	0,000

Table 4.3 - Phase 5 Questions Average Times, respective Standard Deviation and average tries and respective Standard Deviation.

4.5 Second Usability Test Results

As mentioned, a second Usability Test was done after the new interface was implemented. The objective was to assess if a less cluttered and more direct generation flow translated in a better ease of use for the user. In this test, only Phase one of the first usability test was applied. Phase one questions (see Section 4.4) were not altered in their contents, but question 2b) and 2c) were removed

because the corresponding buttons no longer exist. This change in the UI also required reordering of some questions, namely: question 2d) became question 2b) and question 2e) became question 2c). Check the Appendix D for the complete altered test document. Only five participants were tested, four males and one female, with an age average of 22.4 years old. None of these participants had any knowledge or participated on any of the two previous tests.

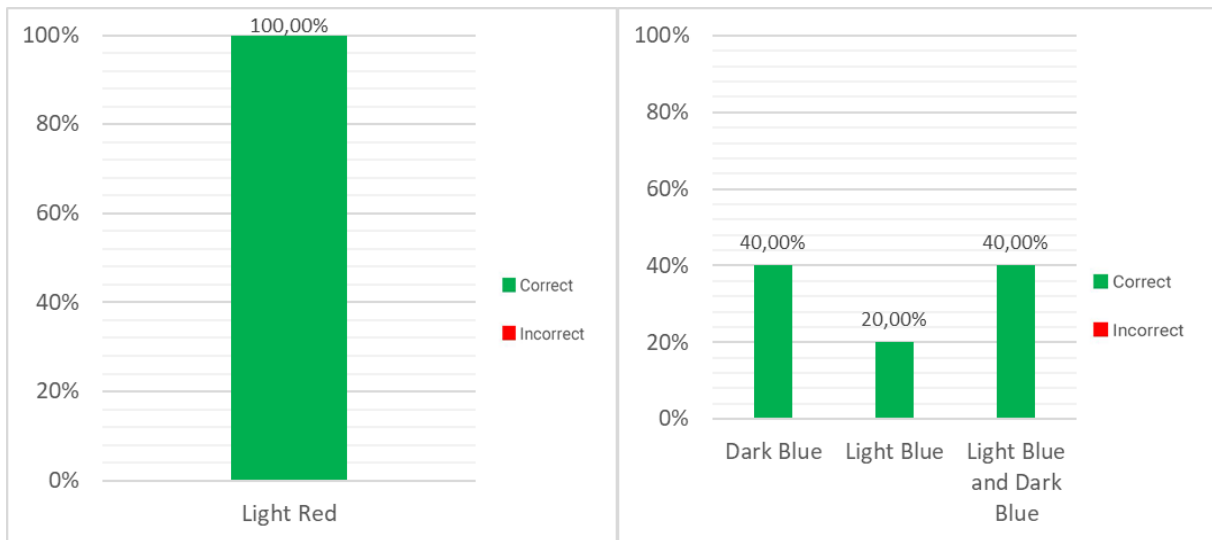


Figure 4.14 - Results for question 1a) "Imagine that you want to draw a mountain range, in which button would you press?" (left) and 1b) "Now imagine that you want to draw a depression on your terrain, which button would you choose?" (right), second usability test.

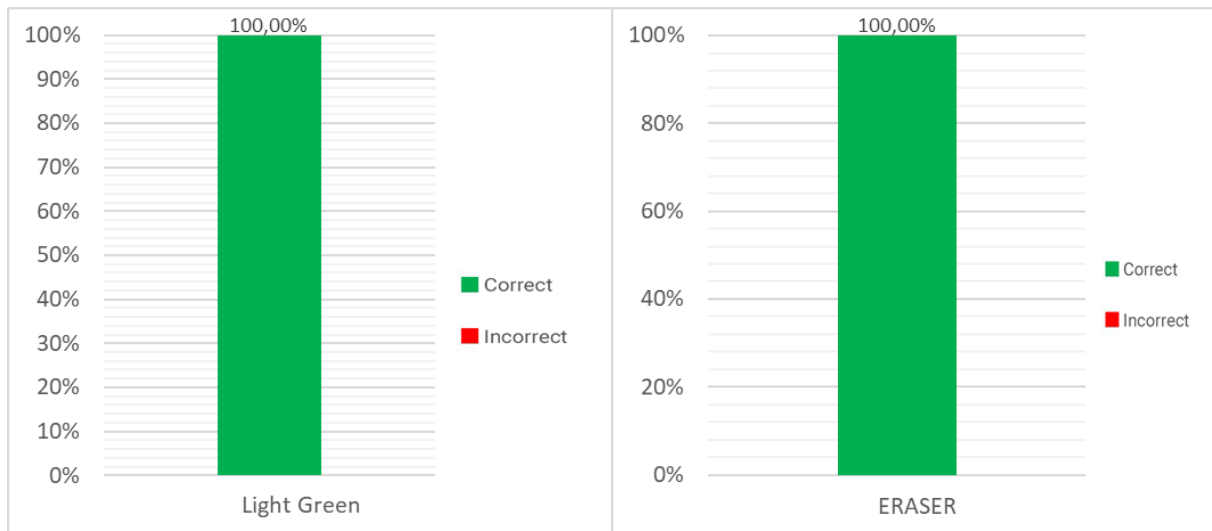


Figure 4.15 - Results for question 1c) "And to draw a single elevation?" (left) and 1d) "Imagine that you made a mistake, and you want to erase part of what you did, which button would you choose?" (right), second usability test.

In question 1a) (see Figure 4.14), 100 % of participants answered correctly, although this time, no one picked more than one color as their answer. Participants had a time average of 3.3 second (SD = 1.8 s). Moving to question 1b) (see Figure 4.14), again 100 % of participants answered correctly, with 40 % picking dark blue, 20 % light blue and 40 % answering that both light blue and dark blue could be used to produce depression on the terrain. For this question, participants had a time average of 2.4 s (SD = 1.6 s). In question 1c) (see Figure 4.15), again 100 % of participants chose the correct answer and their time average was 1.8 second (SD = 0.3 s), according with Table 4.4. In question 1d) (see Figure 4.15), 100 % of participants answered correctly as well. Participants had a time average of 1.5 s (SD = 0.7 s), as seen in Table 4.4. These results do not askew from the results obtained in the previous test, reiterating that participants have no trouble identifying the drawing tools button’s functionality.

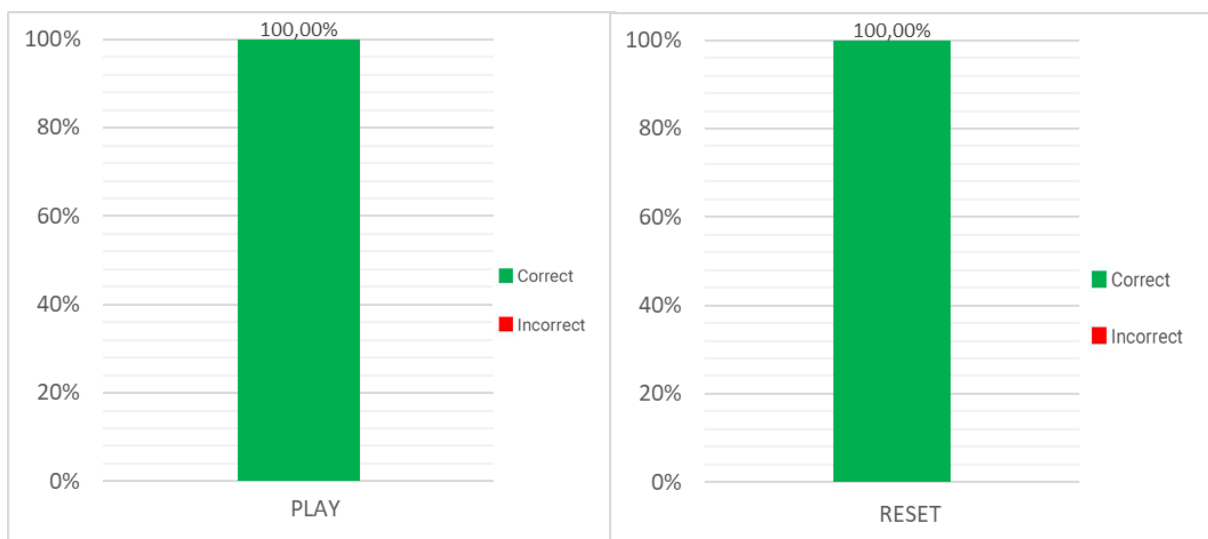


Figure 4.16 - Results for question 2a) *“Now imagine that you are feeling satisfied with the drawing you did, and you want the drawing in your drawing area to be used to generate a terrain, where would you press?”* (left) and 2b) *“Imagine that in the middle of drawing your next terrain, you decided that your drawing wasn’t what you wanted at all and decide to completely clean the drawing area, where would you press?”*. (right), second usability test.

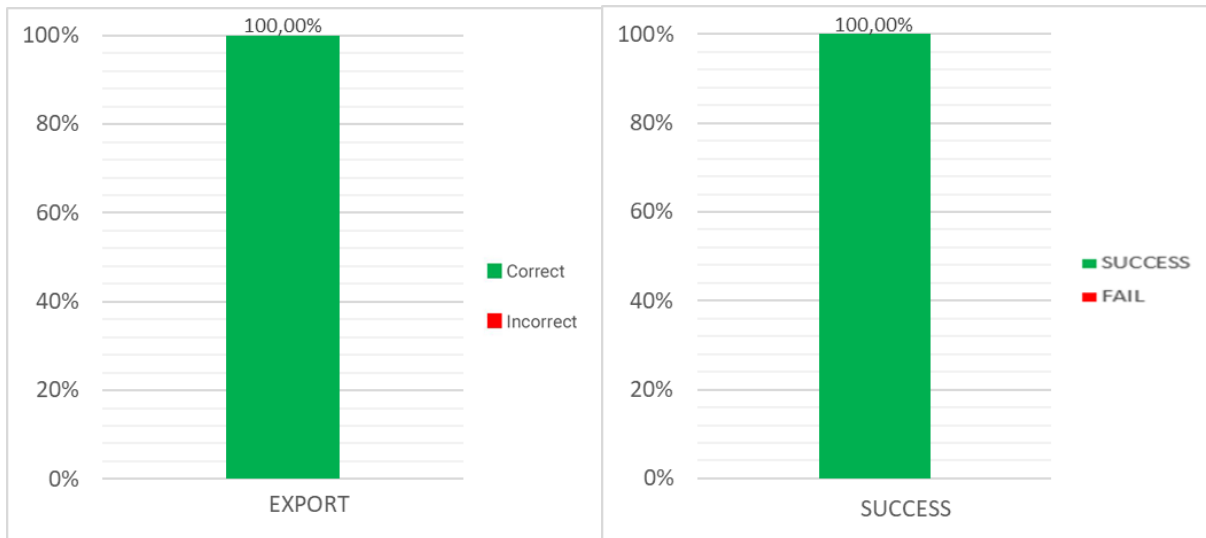


Figure 4.17 - Results for question 2c) “Finally, you’ve drawn something of your liking and you think the terrain should be stored. In which button would you press to export your terrain?” (left) and 3a) “Imagine that you started a new session and decide to make alterations in the terrain I showed you in the beginning. What actions would you take?” (right), second usability test.

In question 2a) (see Figure 4.16), 100 % of users correctly identified the play button as the button that initiates the generative process, averaging 1.6 s to answer (SD = 1 second), according to Table 4.4. This is a massive improvement over the previous test. What once was achieved with 3 button click is now achieved with just one click, reducing the amount of time needed to go from sketch to 3D terrain as well as reducing the complexity of having to click another two buttons to be able to see a resulting terrain and update the image log. Question 2b) (see Figure 4.16), 100 % of participants correctly identified the reset button as the button that clear the canvas of any sketches. Time average for this question was 1.7 s (SD = 1 second), as seen in table 4.4. This result does not differ from the initial test results, no participant has issues identifying the reset button previously and this test was not an exception. In question 2c) (see Figure 4.17), again, 100 % of participants answered correctly and averaged 0.9 s (SD = 0.2 s), as seen in table 4.2. Just like question 2b), no participant had trouble correctly identifying the export button function, on the previous interface and on the newly created one. Question 3a) (see Figure 4.17), tests the image log. 100 % of participants answered correctly and had a 16.5 second time average (SD = 6 s), as seen in Table 4.3.

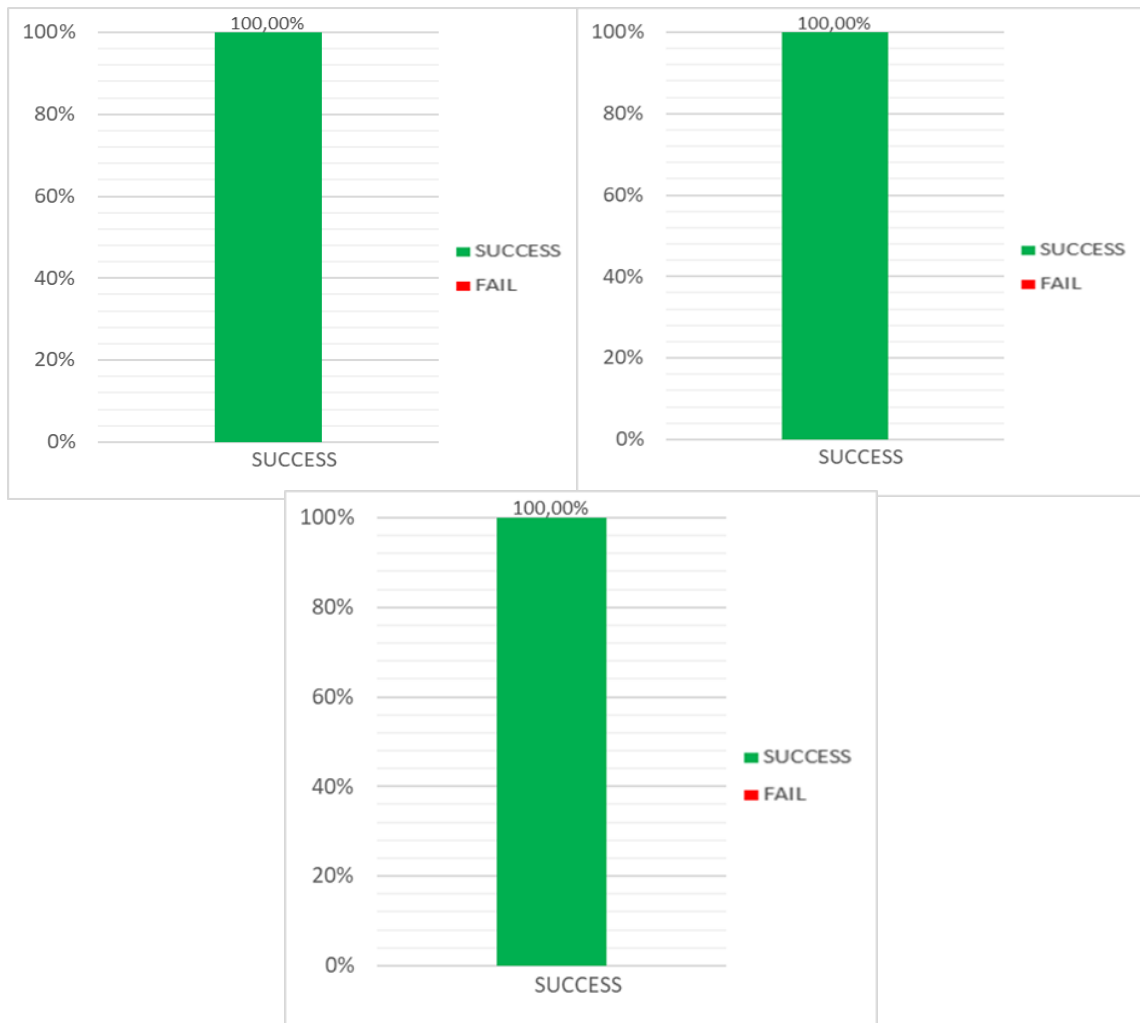


Figure 4.18 - Results for question 4a) "You took notice of a detail in your terrain and decide to zoom in on that part. How would you do it?" (top left), 4b) "Now you want to see the same detail, but from another perspective, how would you do it?" (top right) and 4c) "Finally, you want to see the map from a top down view, where would you press?" (bottom) of second usability test.

Question 4a) (see Figure 4.18), 100 % of participants managed to control the camera successfully, with a time average of 18.3 s (SD = 12.5 s). In question 4b) (see Figure 4.18), again, 100 % of participants were able to complete the rest of camera control actions successfully, averaging 18.3 s (SD = 5,6 s). And finally, question 4c) (see Figure 4.19), 100 % of participants were able to reset the terrain position on the terrain visualizer. Time average was 1.4 s (SD = 0.8 s), as seen in Table 4.4.

When looking at these results and comparing them with the same questions on the previous test, it can be concluded that the UI changes had an impact on overall clarity and usability. For the first group of questions, 1a) through 1d), results were similar, in terms of correct answer percentage and time averages. Although time averages are slightly lower, but this can be attributed to the smaller

sample size. The second group of questions, however, is where we see the biggest impact. No participants switched button functionality and every participant answered correctly. The biggest change, even though the sample size is small, is the times. Participants went from averaging 5 s to answer 2a) to 1.6 s and since the other two buttons, the generate and update log buttons, no longer exist, the impact is even bigger. Participants took approximately 8 s on average to answer what buttons had the functions of updating the terrain and updating the log menu, since those two do not exist anymore and their functions are all condensed in one button, the play button, our time gain is much bigger than just the direct difference. Now the full generative flow requires one single click.

On question 3a), participants took less time on average than on the first test. Since no major changes were made to the image log, this time improvement can be a result of the small sample size. There is also the possibility that the streamlining of the generative flow made things less cluttered, enough for users to focus more of their attention in the image log and not on the group of buttons that used to be on the bottom left. The same can be said for questions 4a) and 4b). Nothing was changed on the camera controls that can justify the big-time average difference. A bigger sample size would be needed to find out if indeed the uncluttering of the UI had impact on unaltered parts of it. Question 4c) is had similar results to the initial test. We can safely say that this interface is more usable than the original interface. Less clutter and confusing names led to a much more concise and usable interface. Just as in the first Usability test, each tester's time to answer the questions posed were clocked and time averages and SD were compiled. Considerations regarding these results are done in the previous section.

Questions	Response time (s)	
	Average	Standard Deviation
1a)	3,3	1,864
1b)	2,4	1,694
1c)	1,8	0,353
1d)	1,5	0,736
2a)	1,6	1,004
2b)	1,7	1,045
2c)	0,9	0,289
3a)	16,5	6,023
4a)	18,3	12,509
4b)	14,8	5,62
4c)	1,4	0,86

Table 4.4 - Second Usability Test Time Averages and Standard Deviation.

Conclusions

The tool presented in this dissertation, is a mixed initiative tool that allows its users to produce realistic 3D terrains without formal terrain design training, leveraging a NN, trained in real terrain, as its generative engine.

The performed literature survey in Chapter 2 revealed that NNs, although not novel, have come a long way, especially when it comes to image generation. GANs opened up generative possibilities that were not possible in the past. When considering cGANs, these further enhanced the possibility of guiding GANs towards desired representations, allowing for a bigger refinement of the learned distribution. These refinements paved the way for PCG to use this technology as its main generative engine. PCG, likewise NNs, has come a long way, especially when it comes to industry adoption, where we see a growing number of big content creation studios, more specifically, gaming studios, adopting PCG solutions. These solutions started as being implemented as a solution for hardware shortcomings and evolved to solutions that enable developers to reduce time and resources in content creation as well as allowing them to produce massive amounts of content, that would otherwise be humanly impossible to create, given the timeframe. But even with this coming of age of PCG, there is still a lack of intuitive, simple and controllable tools that allow developers to create the assets they need, with the necessary authoring level to avoid bland and repetitive content. The literature review revealed a lack of tools with these characteristics: mixed-authorship tools with a simple design language and removing the necessity of in-depth training, wasting manhours in learning the tool instead of producing with it. Even more so, it revealed the lack of these kinds of tool for 3D terrains, arguably one of the cornerstones of any game genres and one that requires a lot of time investment to create. This dissertation was therefore developed to address this lack of mixed-authorship tools for 3D terrain generation by contributing with a new tool based on Neural Networks (NN) and by assessing its actual utility and usability. This is achieved by refining the existing work of Guérin et al. (Guérin et al., 2017), allowing user to have a finer authoring control over the generated 3D terrains.

To validate the implemented mixed-authorship tool, a set of two different tests was run. The first test was an online questionnaire that compared 3D terrains using Perlin Noise, a widely used algorithmic solution for terrain generation, with terrains generated by the tool's NN. All terrains were rendered inside the tool to maintain visual coherence. This test had as objective understanding if potential users found the solution generated with the NN to be more realistic looking than those generated with Perlin Noise. The second test was done in-person to participants that accepted

participating in this test. The objective was to gauge whether uninformed users would be able to understand the tool without in-depth training and if they would be able to operate it without in-depth training. Participants were guided through the test that tried to mimic how a real session would be, being tests from interface comprehension to actual 3D terrain production.

The online questionnaire, in which 113 participants answered which terrain looked more realistic between 30 pairs of terrains produced by Perlin Noise and with terrains produced by the Neural Network, yielded positive results. Most participants choose the terrain generated by the Neural Network as the most realistic looking terrains. This effectively answers this dissertation's second research question, that questioned whether users would find the terrains produced by this tool to be more realistic when compared to randomly generated terrain generated without user input.

The second test, in which 30 participants were tests regarding the tools usability and utility, produced mixed results. Most participants found the interface buttons' representations to be intuitive and clear. Still on the second test, in the group of questions where participants were tested regarding their comprehension of buttons necessary to press to obtain a 3D terrain based on their sketch and to update all existing structures (generation flow), the majority of users was unable to understand which buttons were necessary to press, and in which order. After a normalizing tutorial reading and a non-supervised familiarization session, participants were tested in complete terrain generation tasks using the tool. Positive results were obtained in this part of the second test, with the majority of the participants being able to express their terrain design intentions based on what was being asked.

Although these results already answered this dissertation's first research question, which asked if it is possible to leverage Machine Learning (ML) to create a tool that allows users of varied fields of experience to create a realistic looking 3D terrain without requiring any formal training, the results on the generative flow section of the second test prompted for an interface revision. The interface was streamlined, with all superfluous buttons taken out and all their functions being condensed in a single button. A smaller second round of testing was performed, to gauge if the changes had any impact on usability. The results were positive, with all five participants being able to now identify every part of the interface with problems and without previous explanation. This further cements the answer for the first research question: it is in fact possible to leverage machine learning to create a tool that allows users from all experience levels to create a realistic looking 3D terrain without formal training. Given the test results and the overall positive reception by participants regarding the developed tool, we can conclude that it is indeed possible to create a mixed-authorship tool that is easy to use and powerful enough to generate better results than broadly used random generative methods.

Even though the results are positive, there are still several steps needed to further improve the developed tool in order to make it more useful to use in a production setting. The outputted terrain size needs to be controllable, in order to be possible to scale terrains for any project, large or small. Having more NNs to choose from, each trained in specific terrain types, could also broaden the authorship of the terrains as well as allow for a more biome-oriented generation. An integration with a texture generator would greatly increase productivity and terrain realism, as well as an erosion engine to further refine the terrain. The addition of 3D water would also be a feature that would greatly improve terrain realism. There is still a long way to go for mixed-authorship tools, but there is indeed a valid use for this technology.

Bibliographical references

- Archer, T. (2011). Procedurally Generating Terrain. *3rd International Conference on Computer Supported Education*. Retrieved from http://micsymposium.org/mics_2011_proceedings/mics2011_submission_30.pdf
- Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein Generative Adversarial Network. *International Conference on Machine Learning*, 1–44. <https://doi.org/10.1080/15563650600584519>
- Aziza, R., Borgi, A., Zgaya, H., & Guinhouya, B. (2016). Simulating Complex systems: Complex system theories, their behavioural characteristics and their simulation. *ICAART 2016 - Proceedings of the 8th International Conference on Agents and Artificial Intelligence, 2*, 298–305. <https://doi.org/10.5220/0005684602980305>
- Barreto, N., Cardoso, A., & Roque, L. (2014). Computational Creativity in Procedural Content Generation: A State of the Art Survey. *Proceedings of the 2014 Conference of Science and Art of Video Games*, (Cc). <https://doi.org/10.13140/2.1.1477.0882>
- Barriga, N. A. (2019). A Short Introduction to Procedural Content Generation Algorithms for Videogames. *International Journal on Artificial Intelligence Tools*, 28(2), 1–11. <https://doi.org/10.1142/S0218213019300011>
- Beckham, C., & Pal, C. (2017). A step towards procedural terrain generation with GANs, 0–4. Retrieved from <http://arxiv.org/abs/1707.03383>
- Coello, C. A. (2005). An introduction to evolutionary algorithms and their applications. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3563 LNCS(2508), 425–442. https://doi.org/10.1007/11533962_39
- De Carpentier, G. J. P., & Bidarra, R. (2009). Interactive GPU-based procedural heightfield brushes. *FDG 2009 - 4th International Conference on the Foundations of Digital Games, Proceedings*, 55–62. <https://doi.org/10.1145/1536513.1536532>
- Dix, A., Finlay, J., Abowd, G. D., & Beale, R. (2004). *Human-Computer Interaction Ch. 9 Evaluation Techniques*. Retrieved from www.hcibook.com
- Doran, J., & Parberry, I. (2010). Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2), 111–119. <https://doi.org/10.1109/TCIAIG.2010.2049020>

- E. Catmull, J. C. (1978). Recursively generated B-spline surfaces on arbitrary topological meshes, 350–355.
- Ebert, D. S., F. Kenton Musgrave, Peachey, D., Perlin, K., Worley, S., Mark, W. R., & Hart, J. C. (2003). Texturing & modeling: A Procedural Approach, 687. Retrieved from <http://books.google.com/books?hl=en&lr=&id=bDISJd8GfMcC&pgis=1>
- Eiben, A. E., & Smith, J. E. (2003). What is an Evolutionary Algorithm?, 15–35. https://doi.org/10.1007/978-3-662-05094-1_2
- Fêo, M., Santos, F., & Santana, P. (2017). Visualising and Editing Graphical Representations for Procedurally Generated Designs using Shape Grammars. *EPCGI*.
- Forbus, K. D., Mahoney, J. V., & Dill, K. (2002). How qualitative spatial reasoning can improve strategy game AIs, 1–8.
- Fournier, A., Fussell, D., & Carpenter, L. (1982). Computer Rendering of Stochastic Models. *Communications of the ACM*, 25(6), 371–384. Retrieved from <http://delivery.acm.org/10.1145/360000/358553/p371-fournier.pdf?ip=82.37.62.65&id=358553&acc=ACTIVE> SERVICE&key=BF07A2EE685417C5.3DCFD3605FE4B4CE.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=1006996970&CFTOKEN=70335065&__acm__=1511240484_99dbdb143d720086daf7878
- Frade, M., de Vega, F. F., & Cotta, C. (2012). Automatic evolution of programs for procedural generation of terrains for video games: Accessibility and edge length constraints. *Soft Computing*, 16(11), 1893–1914. <https://doi.org/10.1007/s00500-012-0863-z>
- Freiknecht, J., & Effelsberg, W. (2017). A Survey on the Procedural Generation of Virtual Worlds. *Multimodal Technologies and Interaction*, 1(4), 27. <https://doi.org/10.3390/mti1040027>
- Goodfellow, I. (2016). NIPS 2016 Tutorial: Generative Adversarial Networks. Retrieved from <http://arxiv.org/abs/1701.00160>
- Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets, 2672–2680.
- Guérin, É., Digne, J., Galin, É., Peytavie, A., Wolf, C., Benes, B., & Martinez, B. (2017). Interactive example-based terrain authoring with conditional generative adversarial networks. *ACM Transactions on Graphics*, 36(6). <https://doi.org/10.1145/3130800.3130804>
- Guzdial, M., Liao, N., & Riedl, M. (2018). Co-creative level design via machine learning. *CEUR Workshop Proceedings*, 2282.

- Hendriks, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013a). Online Appendix to: Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, (1), 1–4. <https://doi.org/10.1145/2422956.2422958>
- Hendriks, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013b). Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, 9(1), 1–22. <https://doi.org/10.1145/2422956.2422957>
- Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>
- Hyttinen, T. (2017). Terrain synthesis using noise, (May).
- IFPI. (2019). IFPI Global Music Report 2019. *International Federation of the Phonographic Industry (IFPI)*.
- Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2016). Image-to-Image Translation with Conditional Adversarial Networks. *ArXiv.Org*, 1125–1134. <https://doi.org/10.1109/CVPR.2017.632>
- Jain, A. K., Mao, J., Road, H., & Jose, S. (1996). Artificial Neural Networks : A Tutorial, 1–49.
- Jain, R., Isaksen, A., Holmgård, C., & Togelius, J. (2016). Autoencoders for Level Generation, Repair, and Recognition. *ICCC Workshop on Computational Creativity and Games*.
- Jetchev, N., Bergmann, U., & Vollgraf, R. (2017). Texture Synthesis with Spatial Generative Adversarial Networks. Retrieved from <http://arxiv.org/abs/1611.08207>
- Karth, I., & Smith, A. M. (2019). Addressing the fundamental tension of PCGML with discriminative learning. *ACM International Conference Proceeding Series*. <https://doi.org/10.1145/3337722.3341845>
- Katzenbach, C., Herweg, S., & Van Roessel, L. (2016). Copies, clones, and genre building: Discourses on imitation and innovation in digital games. *International Journal of Communication*, 10, 838–859.
- Khaled, R., Nelson, M. J., & Barr, P. (2013). Design metaphors for procedural content generation in games. *Conference on Human Factors in Computing Systems - Proceedings*, 1509–1518. <https://doi.org/10.1145/2470654.2466201>
- Kishore Papineni, Salim Roukos, T. W., & Zhu, W.-J. (2002). BLEU: a Method for Automatic Evaluation of Machine Translation. *Design Review : Challenging Urban Aesthetic Control*, (July), 219. <https://doi.org/10.3115/1073083.1073135>

- Koh, E., & Hearn, D. D. (1992). Fast Generation and Surface Structuring Methods for Terrain and Other Natural Phenomena. *Computer Graphics Forum*, 11(3), 169–180.
<https://doi.org/10.1111/1467-8659.1130169>
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, 1–9.
<https://doi.org/http://dx.doi.org/10.1016/j.protcy.2014.09.007>
- Lagae, A., Lefebvre, S., Cook, R., Derose, T., Drettakis, G., Ebert, D. S., ... Survey, A. (2010). A Survey of Procedural Noise Functions.
- Lewis, C., & Rieman, J. (1993). Task-Centered User Interface Design: A Practical Introduction. *Text*, 190. Retrieved from <http://hcibib.org/tcuid/tcuid.pdf>
- Lewis, J. P. (1987). Generalized Stochastic Subdivision. *ACM Transactions on Graphics (TOG)*, 6(3), 167–190. <https://doi.org/10.1145/35068.35069>
- Li, C., & Wand, M. (2016). Precomputed real-time texture synthesis with markovian generative adversarial networks. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9907 LNCS, 702–716.
https://doi.org/10.1007/978-3-319-46487-9_43
- Liapis, A., Yannakakis, G. N., & Togelius, J. (2013). Sentient sketchbook: Computer-aided game level authoring. *Proceedings of the 8th International Conference on the Foundations of Digital Games (FDG 2013)*, 213–220. Retrieved from
http://www.itu.dk/people/anli/mixedinitiative/sentient_sketchbook.pdf
http://www.fdg2013.org/program/papers/paper28_liapis_etal.pdf
- Lin, M., Chen, Q., & Yan, S. (2014). Network In Network, 1–10. Retrieved from
<http://arxiv.org/abs/1312.4400>
- McCulloch, W. S., & Pitts, W. H. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5, 115–133.
- Miller, G. S. P. (1986). The Definition and Rendering of Terrain Maps. *SIGGRAPH*, 20(4), 39–48.
<https://doi.org/10.1145/15886.15890>
- Mirza, M., & Osindero, S. (2014). Conditional Generative Adversarial Nets, 1–7. Retrieved from
<http://arxiv.org/abs/1411.1784>

- Nareyek, A. (2001). *Constraint-Based Agents An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.
- Nielsen, M. (2018). Neural Networks and Deep Learning. *Artificial Intelligence*, 389–411. <https://doi.org/10.1201/b22400-15>
- Paul Werbos. (1974). Beyond regression : new tools for prediction and analysis in the behavioral sciences, (February). Retrieved from <https://cul.worldcat.org/title/beyond-regression-new-tools-for-prediction-and-analysis-in-the-behavioral-sciences/oclc/77001455>
- Peachey, D. R. (1985). Solid Texturing of Complex Surfaces, *19*(3), 279–286.
- Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics*, *19*(3), 287–296. <https://doi.org/10.1145/325165.325247>
- Perlin, K. (2002). Improving noise. *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '02*, 681. <https://doi.org/10.1145/566570.566636>
- Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 1–16. <https://doi.org/10.1051/0004-6361/201527329>
- Raffe, W. L., Zambetta, F., & Li, X. (2012). A survey of procedural terrain generation techniques using evolutionary algorithms. *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, 10–15. <https://doi.org/10.1109/CEC.2012.6256610>
- Risi, S., & Togelius, J. (2019). Procedural Content Generation: From Automatically Generating Game Levels to Increasing Generality in Machine Learning. Retrieved from <http://arxiv.org/abs/1911.13071>
- Roden, T., & Parberry, I. (2004). From artistry to automation: A structured methodology for procedural content creation. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3166*, 151–156. https://doi.org/10.1007/978-3-540-28643-1_19
- Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation, 1–8.
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 1–23. <https://doi.org/10.1098/rspb.1976.0087>

- Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved Techniques for Training GANs, 1–10. Retrieved from <http://arxiv.org/abs/1606.03498>
- Sampath, D. (2004). ABRCon, adaptive object Re-configuration: An approach to enhance, repeat payability of games and repeat watchability of movies. *ACM International Conference Proceeding Series, 74*, 313–320.
- Sarle, W. S. (1994). Neural networks and statistical models. *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, 1538–1550. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.699>
- Schaller, R. R. (1997). Moore’s law: past, present and future. *IEEE Spectrum Magazine, 34*(6), 52–59.
- Shaker, M., Shaker, N., & Togelius, J. (2013). Ropossum: An authoring tool for designing, optimizing and solving cut the Rope levels. *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2013*, 215–216.
- Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural Content Generation in Games*. Springer. https://doi.org/10.1007/978-3-319-30004-7_6
- Smelik, R. M., Tutenel, T., De Kraker, K. J., & Bidarra, R. (2010). A declarative approach to procedural modeling of virtual worlds. *Computers and Graphics (Pergamon), 35*(2), 352–363. <https://doi.org/10.1016/j.cag.2010.11.011>
- Smith, G. (2014). Procedural Content Generation. *Handbook of Digital Games*, 501–518. <https://doi.org/10.1002/9781118796443.ch2>
- Smith, G., Whitehead, J., & Mateas, M. (2011). Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games, 3*(3), 201–215. <https://doi.org/10.1109/TCIAIG.2011.2159716>
- Snodgrass, S., & Ontañón, S. (2017). Learning to generate video game maps using markov models. *IEEE Transactions on Computational Intelligence and AI in Games, 9*(4), 410–422. <https://doi.org/10.1109/TCIAIG.2016.2623560>
- Spick, R. J., Cowling, P., & Walker, J. A. (2019). Procedural generation using spatial GANs for region-specific learning of elevation data. *IEEE Conference on Computational Intelligence and Games, CIG, 2019-Augus*. <https://doi.org/10.1109/CIG.2019.8848120>
- Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2014). Striving for Simplicity: The All Convolutional Net, 1–14. https://doi.org/10.1163/_q3_SIM_00374
- Summerville, A., & Mateas, M. (2016). Super Mario as a String: Platformer Level Generation Via LSTMs. Retrieved from <http://arxiv.org/abs/1603.00930>

- Summerville, A., Snodgrass, S., Guzdial, M., Holmgard, C., Hoover, A. K., Isaksen, A., ... Togelius, J. (2018). Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*, 10(3), 257–270. <https://doi.org/10.1109/tg.2018.2846639>
- Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., & Stanley, K. O. (2013). Procedural Content Generation : Goals, Challenges and Actionable Steps. *Artificial and Computational Intelligence in Games*, 6(January 2013), 61–75. <https://doi.org/10.4230/DFU.Vol6.12191.61>
- Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. N. (2011). What is Procedural Content Generation? Mario on the borderline. *Level Design*, 159–182. <https://doi.org/10.1201/9781315313412-9>
- Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., & Yannakakis, G. N. (2010). Multiobjective exploration of the StarCraft map space. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010*, 265–272. <https://doi.org/10.1109/ITW.2010.5593346>
- Togelius, J., Preuss, M., & Yannakakis, G. (2010). Towards multiobjective procedural map generation. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games PCGames 10*, 1–8. <https://doi.org/10.1145/1814256.1814259>
- Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3), 172–186. <https://doi.org/10.1109/TCIAIG.2011.2148116>
- Torrado, R. R., Khalifa, A., Green, M. C., Justesen, N., Risi, S., & Togelius, J. (2019). Bootstrapping Conditional GANs for Video Game Level Generation. Retrieved from <http://arxiv.org/abs/1910.01603>
- Van Der Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1), 78–89. <https://doi.org/10.1109/TCIAIG.2013.2290371>
- Vikhar, P. A. (2017). Evolutionary algorithms: A critical review and its future prospects. *Proceedings - International Conference on Global Trends in Signal Processing, Information Computing and Communication, ICGTSPICC 2016*, 261–265. <https://doi.org/10.1109/ICGTSPICC.2016.7955308>
- White, M. M. (2009). The Senescence of Creativity: How Market Forces are Killing Digital Games, 3(4).
- Widrow, B. (1960). An Adaptive “ADALINE” Neuron using chemical “Memistors.”

- Wulff-Jensen, A., Rant, N. N., Møller, T. N., & Billeskov, J. A. (2018). Deep convolutional generative adversarial network for procedural 3D landscape generation based on DEM. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST, 229*, 85–94. https://doi.org/10.1007/978-3-319-76908-0_9
- Yann Lecun, L'Éon Bottou, Yoshua Bengio, P. H. (1998). Gradient-Based Learning Applied to Document Recognition. <https://doi.org/10.1016/j.bbrc.2005.03.111>
- Yannakakis, G. N., & Togelius, J. (2011). Experience-driven procedural content generation. *IEEE Transactions on Affective Computing, 2*(3), 147–161. <https://doi.org/10.1109/T-AFFC.2011.6>
- Yuval Fisher, Michael McGuire, Richard F. Voss, Michael F. Barnsley, Robert L. Devaney, B. B. M. (1988). The Science of Fractal Images.
- Saha, S. (2018, December 15). A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. Medium. Retrieved 1 July 2019, <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>
- Stanford, c. (2019). *CS231n Convolutional Neural Networks for Visual Recognition*. Cs231n.github.io. Retrieved 1 July 2019, <http://cs231n.github.io/convolutional-networks/> .
- Colah, C. (2015). Understanding LSTM Networks -- colah's blog. Colah.github.io. Retrieved 8 July 2019, from <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>..
- Carey, O. (2019). Generative Adversarial Networks (GANs) — A Beginner's Guide. Towards Data Science. Retrieved 2 July 2019, <https://towardsdatascience.com/generative-adversarial-networks-gans-a-beginners-guide-5b38eceece24> .
- Hao, Z. (2019). Hao, Z. IsaacChanghau. IsaacChanghau. Retrieved 8 July 2019, from <https://isaacchanghau.github.io>.
- Nayak, m. (2019). Nayak, M. (2019). Pix2Pix Network, An Image-To-Image Translation Using Conditional GANs (CGANs). Medium. Retrieved 3 July 2019, from <https://medium.com/towards-artificial-intelligence/pix2pix-network-an-image-to-image-translation-using-conditional-gans-cgans-8a08b661d206>.
- Top Countries & Markets by Game Revenues | Newzoo. (2020). Retrieved 21 February 2020, from <https://newzoo.com/insights/rankings/top-10-countries-by-game-revenues/>
- Theesa.com. (2020). Retrieved 21 February 2020, from https://www.theesa.com/wp-content/uploads/2019/05/ESA_Essential_facts_2019_final.pdf.

- www.gamedevmap.com (2020). Retrieved 21 February 2020, from <https://www.gamedevmap.com/>
- Investing in the Soaring Popularity of Gaming. U.S. (2020). Retrieved 21 February 2020, from <https://www.reuters.com/article/sponsored/popularity-of-gaming>.
- U.S. average age of video gamers 2019 | Statista. (2020). Retrieved 21 February 2020, from <https://www.statista.com/statistics/189582/age-of-us-video-game-players-since-2010/>
- The Global Games Market Will Generate \$152.1 Billion in 2019 as the U.S. Overtakes China as the Biggest Market | Newzoo. (2020). Retrieved 21 February 2020, from <https://newzoo.com/insights/articles/the-global-games-market-will-generate-152-1-billion-in-2019-as-the-u-s-overtakes-china-as-the-biggest-market/>
- Kushner, D. (2004). *Masters of Doom*. Random House Trade Paperbacks.
- Krueger, B. D., Brand, O., & Burton, D. (2005). Reinventing your company without reinventing the wheel. In Game Developers Conference.
- Johnson, S. (2006). *The Long Zoom (Published 2006)*. Nytimes.com. Retrieved 21 February 2020, from <https://www.nytimes.com/2006/10/08/magazine/08games.html>.
- Androvich, m. (2008). *GTA IV: Most expensive game ever developed?*. GamesIndustry.biz. Retrieved 22 February 2020, from <https://www.gamesindustry.biz/articles/gta-iv-most-expensive-game-ever-developed>.
- Semuels, A. (2019). *Video Game Creators Are Burned Out and Desperate for Change*. Time. Retrieved 22 February 2020, from <https://time.com/5603329/e3-video-game-creators-union/>.
- FARBRAUSCH PROD. 2006. Website of. kkrieger producer, .theprodukt. <http://www.theprodukt.com/>.
- Gilbert, B. (2019). *Grueling, 100-hour work weeks and 'crunch culture' are pushing the video game industry to a breaking point. Here's what's going on*. Business Insider. Retrieved 2 March 2020, from <https://www.businessinsider.com/video-game-development-problems-crunch-culture-ea-rockstar-epic-explained-2019-5>.
- Persson, M. (2011). *Terrain generation, Part 1*. The Word of Notch. Retrieved 30 April 2020, from <https://notch.tumblr.com/post/3746989361/terrain-generation-part-1>.
- Biome*. Minecraft Wiki. Retrieved 30 April 2020, from <https://minecraft.gamepedia.com/Biome>.
- Whittaker, R. (1962). Classification of natural communities. *The Botanical Review*, 28(1), 1-239. <https://doi.org/10.1007/bf02860872>
- Skyrim: Radiant - The Unofficial Elder Scrolls Pages (UESP)*. Uesp.net. (2020). Retrieved 30 April 2020, from <http://www.uesp.net/wiki/Skyrim:Radiant>.
- Cook, M. (2016). *We've Run Out of Numbers - Procedural Generation After No Man's Sky* - Falmouth University Research Repository (FURR). Repository.falmouth.ac.uk. Retrieved 1 May 2020, from <https://repository.falmouth.ac.uk/2402/>.

SpeedTree - 3D Vegetation Modeling and Middleware. Store.speedtree.com. Retrieved 2 May 2020, from <https://store.speedtree.com/>.

Xfrog - Retrieved 2 May 2020, from <https://xfrog.com/>.

Substance | The leading software solution for 3D digital materials. Substance 3D. Retrieved 4 May 2020, from <https://www.substance3d.com/>.

Darkling Simulation's. Darksim.com. Retrieved 4 May 2020, from <http://www.darksim.com/html/products.html>.

The Substance Art of Rogelio Olguin. Substance. (2016). Retrieved 4 May 2020, from <https://store.substance3d.com/blog/substance-art-rogelio-olguin>.

Substance Designer. Docs.substance3d.com. (2020). Retrieved 4 May 2020, from <https://docs.substance3d.com/sddoc/substance-designer-102400008.html>.

L3DT. (2017). Retrieved 4 May 2020, from <https://www.bundysoft.com/docs/doku.php?id=l3dt:about>.

L3DT documentation. (2017). Retrieved 4 May 2020, from https://www.bundysoft.com/docs/doku.php?id=l3dt:userguide:ops:dm:gen_from_hf.

Advanced 3D City Design Software | Esri CityEngine. Esri.com. Retrieved 4 May 2020, from <https://www.esri.com/en-us/arcgis/products/esri-cityengine/overview>.

Seed (level generation). Minecraft Wiki. Retrieved 15 May 2020, from [https://minecraft.gamepedia.com/Seed_\(level_generation\)](https://minecraft.gamepedia.com/Seed_(level_generation)).

Koeune F. (2005) Pseudo-random number generator. In: van Tilborg H.C.A. (eds) Encyclopedia of Cryptography and Security. Springer, Boston, MA

Berto, F., & Tagliabue, J. (2017). *Cellular Automata (Stanford Encyclopedia of Philosophy)*. Plato.stanford.edu. Retrieved 27 May 2020, from <https://plato.stanford.edu/entries/cellular-automata/>.

EarthExplorer, U. Retrieved 22 July 2020, from <https://earthexplorer.usgs.gov/>.

Godot Engine - Free and open source 2D and 3D game engine. Godotengine.org. (2020). Retrieved 31 July 2020, from <https://godotengine.org/>.

Unity. (2020). Retrieved 31 July 2020, from <https://unity.com/>.

Guérin, E. (2018). *eric-guerin/pix2pix-tensorflow*. GitHub. Retrieved 1 March 2019, from <https://github.com/eric-guerin/pix2pix-tensorflow/tree/png16bits-support>.

Guérin, E. (2018b). *Deep Terrains - Code and data | Page perso - Eric Guérin*. Page perso - Eric Guérin. Retrieved 1 March 2019, from <https://perso.liris.cnrs.fr/eguerin/new/blog/deep-terrains-code-and-data/>.

Installation Guide Linux : CUDA Toolkit Documentation. (2020). Retrieved 29 November 2019, from <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>

Olson, R., Calmels, J., Abecassis, F., & Rogers, P. (2016). NVIDIA Docker: GPU Server Application Deployment Made Easy | NVIDIA Developer Blog. Retrieved 31 November 2019, from <https://developer.nvidia.com/blog/nvidia-docker-gpu-server-application-deployment-made-easy/>

Unity - Scripting API: Mathf.PerlinNoise. (2020). Retrieved 20 August 2020, from <https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html>

Barbosa Anda, F. (2019). Using trained/exported model · Issue #151 · affinelayer/pix2pix-tensorflow. Retrieved 10 July 2019, from <https://github.com/affinelayer/pix2pix-tensorflow/issues/151>

ANNEX

Annex A NN Behavioral Test Pairs

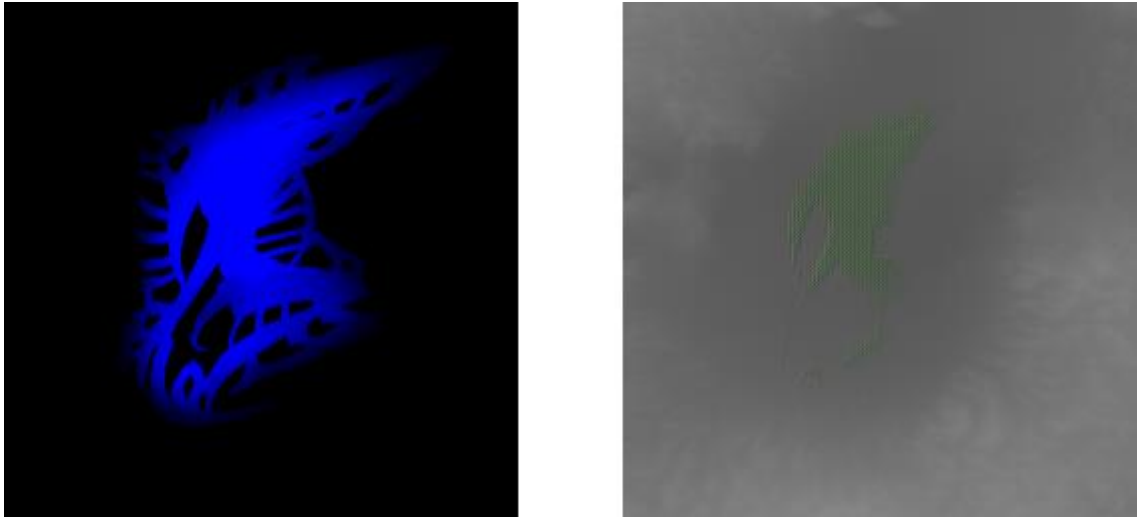


Figure A.1 - Input/Output testing pair. Right side is the input and left side is the output. Output presents corruption due to over exposure of colors in the input.

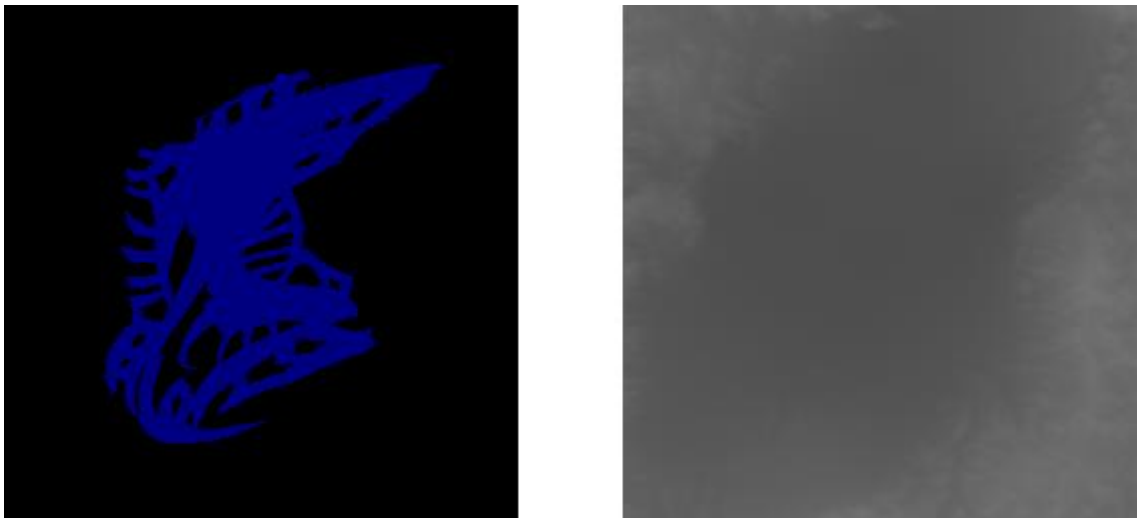


Figure A.2 - Input/Output testing pair. Right side is the input and left side is the output

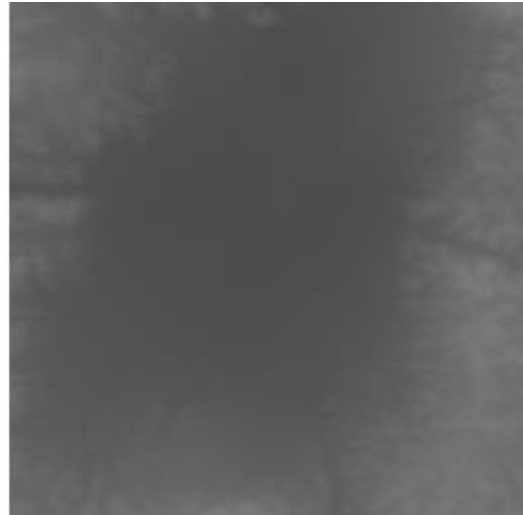
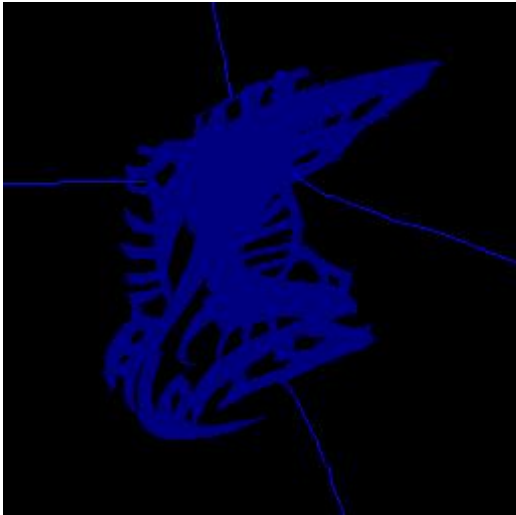


Figure A.3 - Input/Output testing pair. Right side is the input and left side is the output

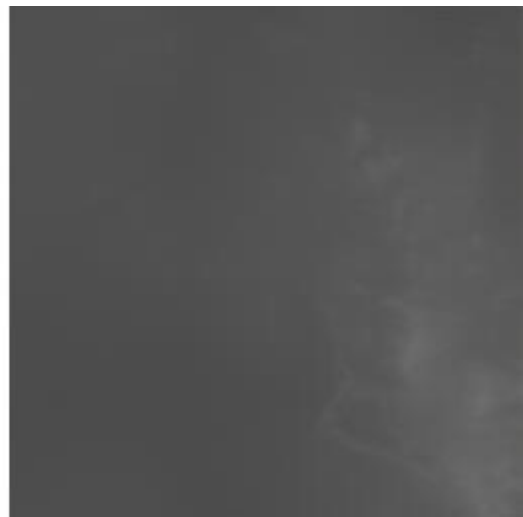


Figure A.5 - Input/Output testing pair. Right side is the input and left side is the output. Over exposed side of the output is give more weight in the output, making the left structure disappear from the output.



Figure A.6 - Input/Output testing pair. Right side is the input and left side is the output



Figure A.7 - Input/Output testing pair. Right side is the input and left side is the output



Figure A.8 - Input/Output testing pair. Right side is the input and left side is the output



Figure A.9 - Input/Output testing pair. Right side is the input and left side is the output

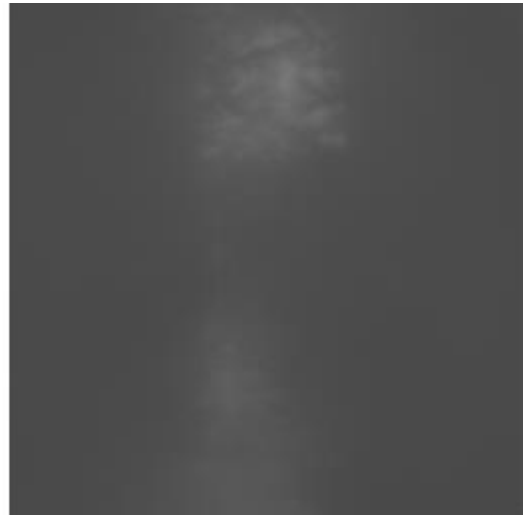


Figure A.10 - Input/Output testing pair. Right side is the input and left side is the output

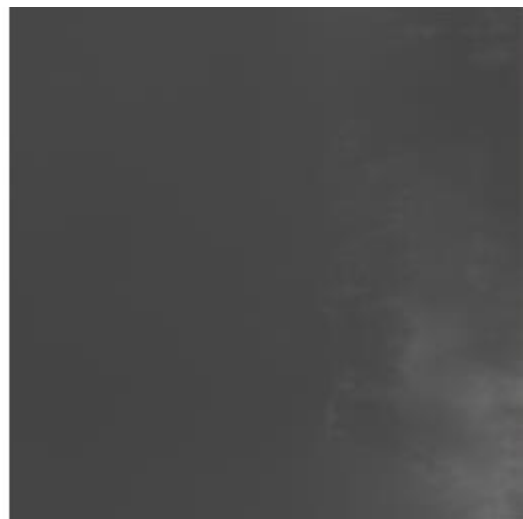


Figure A.11 - Input/Output testing pair. Right side is the input and left side is the output

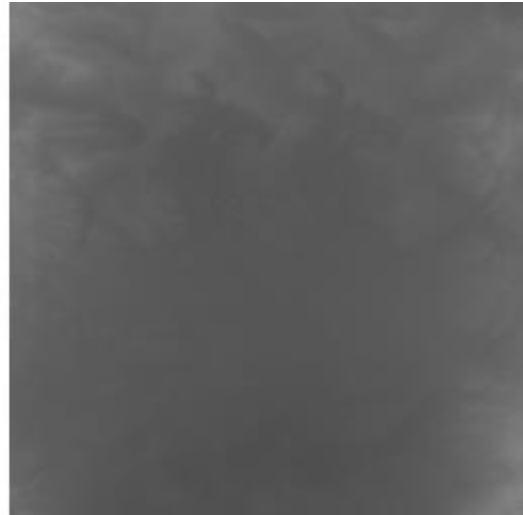


Figure A.12 - Input/Output testing pair. Right side is the input and left side is the output. Black input will always output the heightmap present in the right side of the Figure.

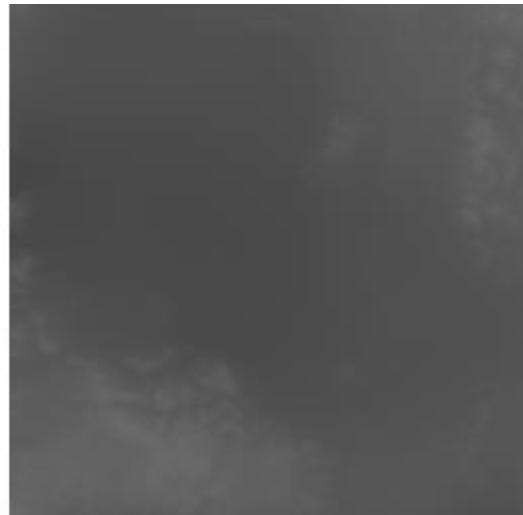
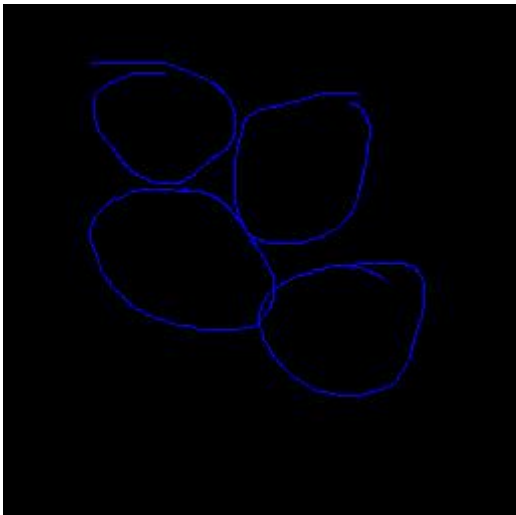


Figure A.13 - Input/Output testing pair. Right side is the input and left side is the output

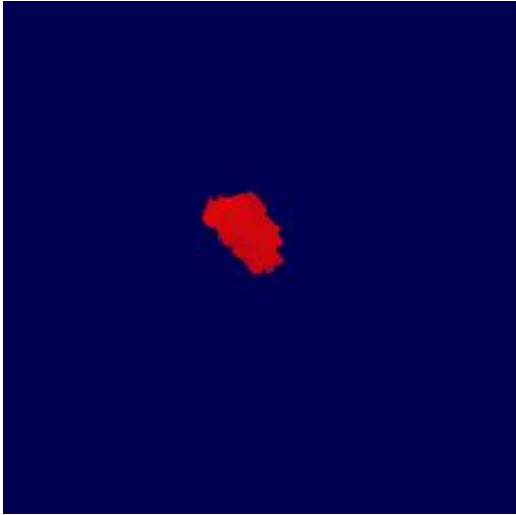


Figure A.14 - Input/Output testing pair. Right side is the input and left side is the output. Output presents output corruption. Transforming this output in a 3D terrain would result in a uniform spike patters in the center of the terrain

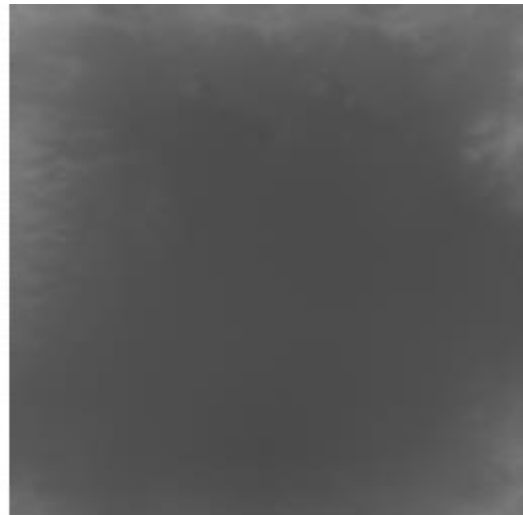


Figure A.15 - Input/Output testing pair. Right side is the input and left side is the output

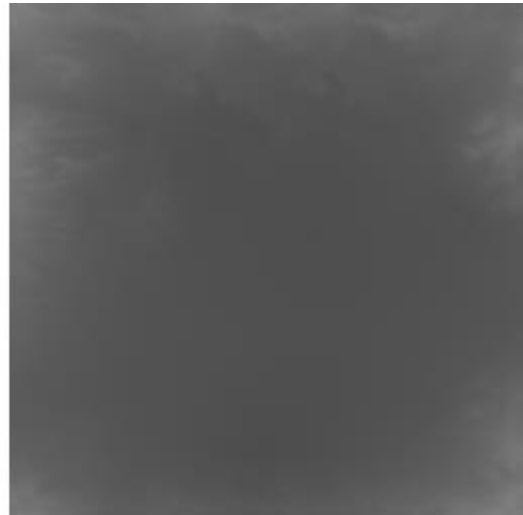


Figure A.16 - Input/Output testing pair. Right side is the input and left side is the output

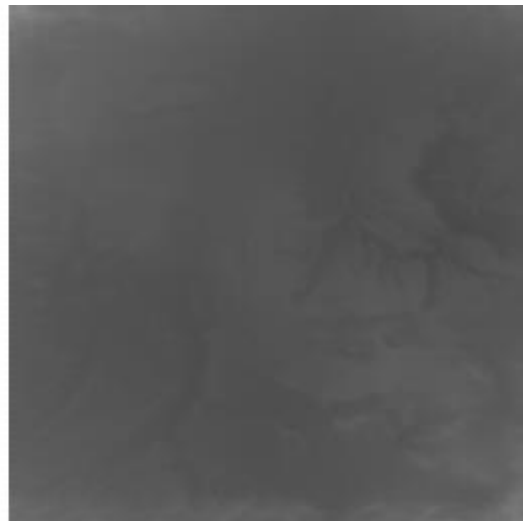
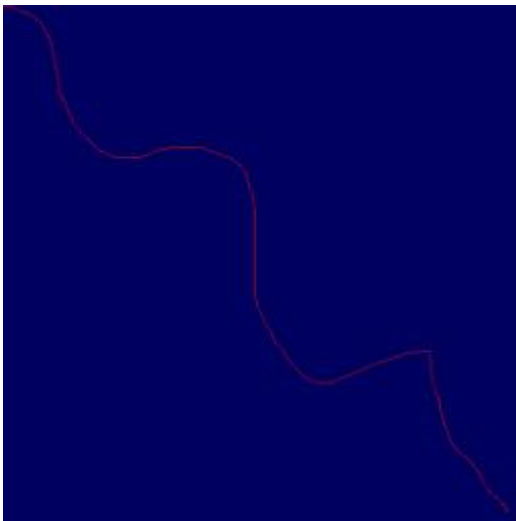


Figure A.17 - Input/Output testing pair. Right side is the input and left side is the output

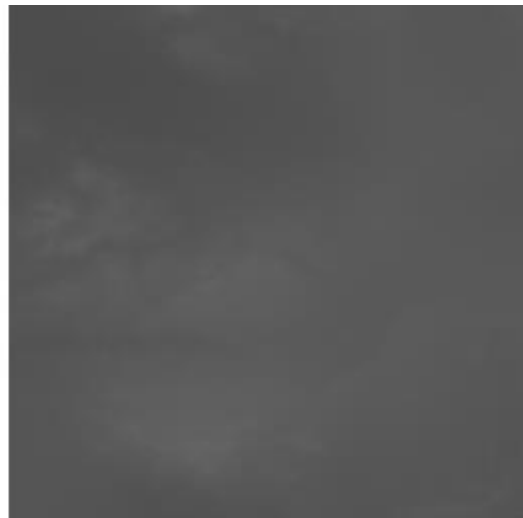
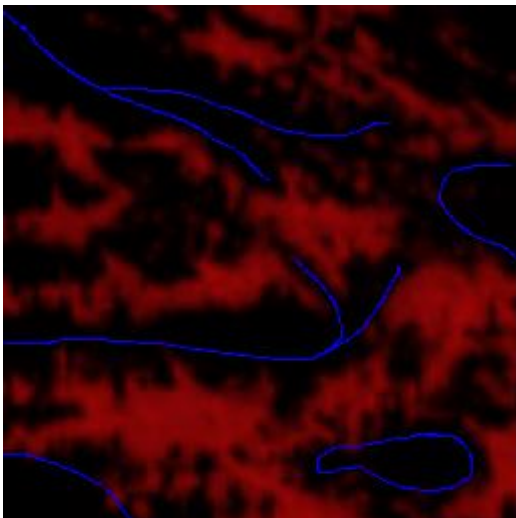


Figure A.18 - Input/Output testing pair. Right side is the input and left side is the output

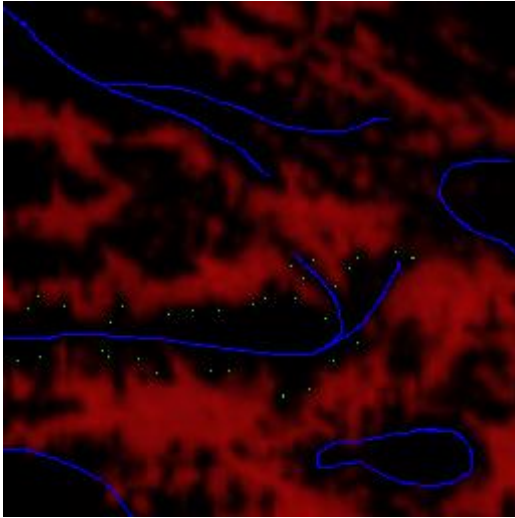


Figure A.19 - Input/Output testing pair. Right side is the input and left side is the output

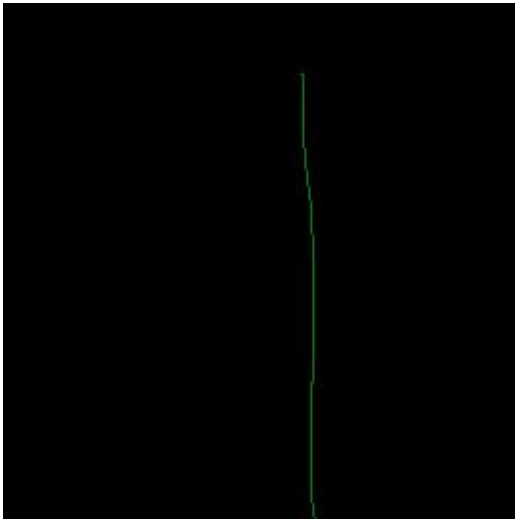


Figure A.20 - Input/Output testing pair. Right side is the input and left side is the output. Green used has 50% opacity comparatively with the original green.

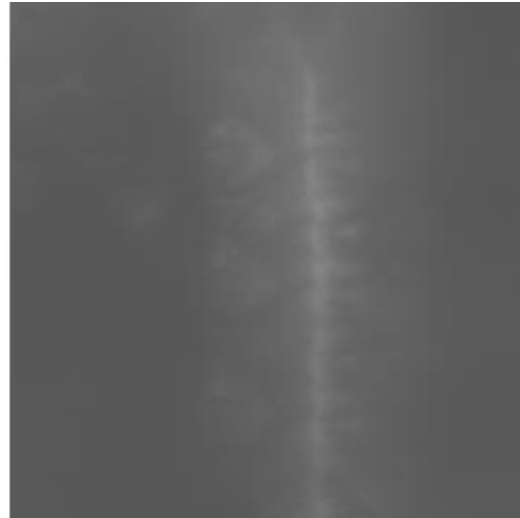
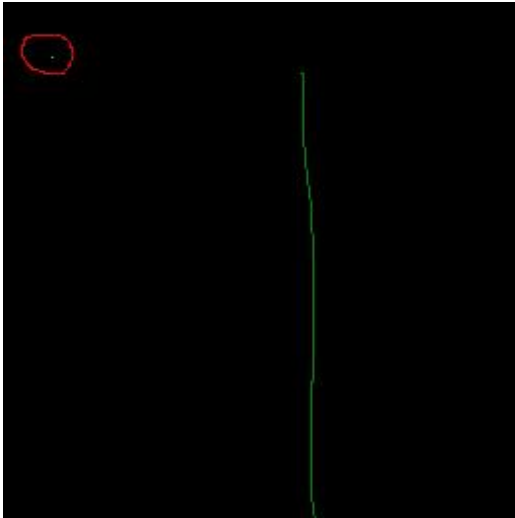


Figure A.21 - Input/Output testing pair. Right side is the input and left side is the output

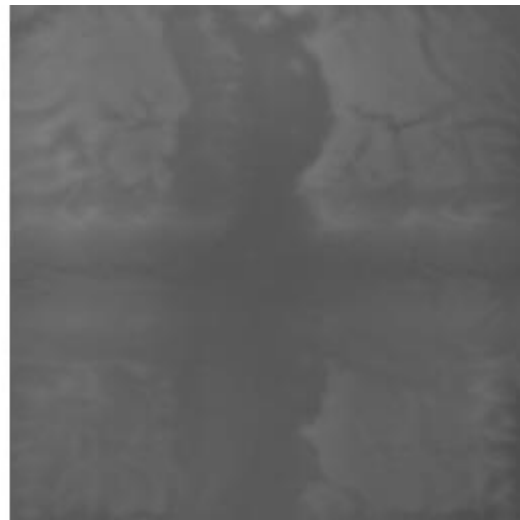


Figure A.22 - Input/Output testing pair. Right side is the input and left side is the output

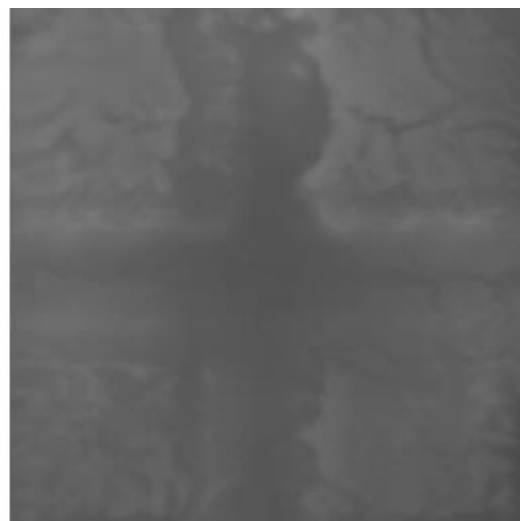
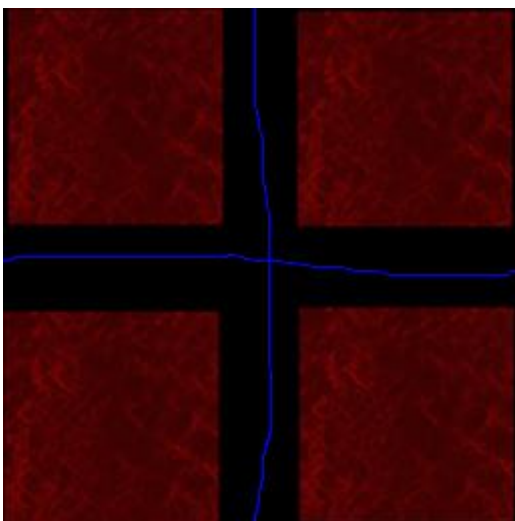


Figure A.22 - Input/Output testing pair. Right side is the input and left side is the output

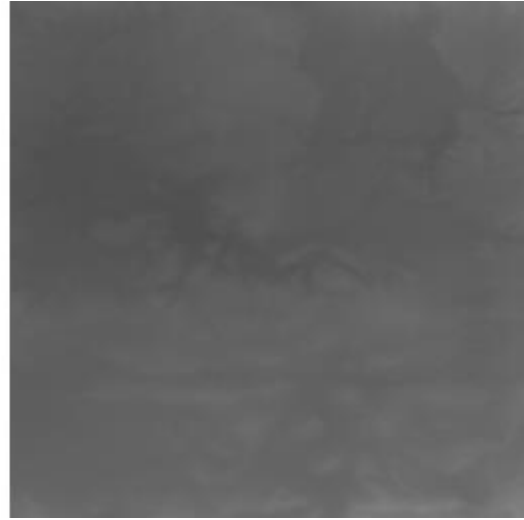
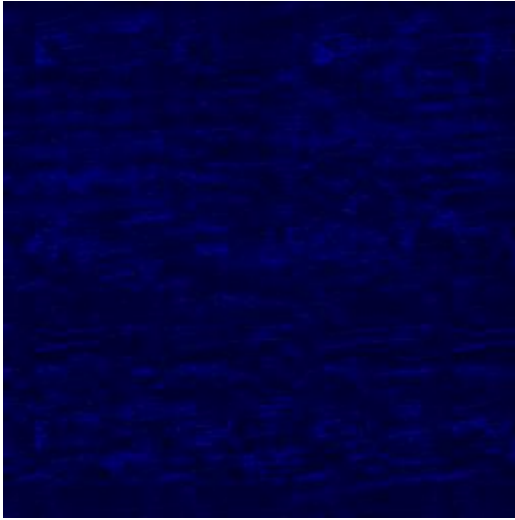


Figure A.23 - Input/Output testing pair. Right side is the input and left side is the output

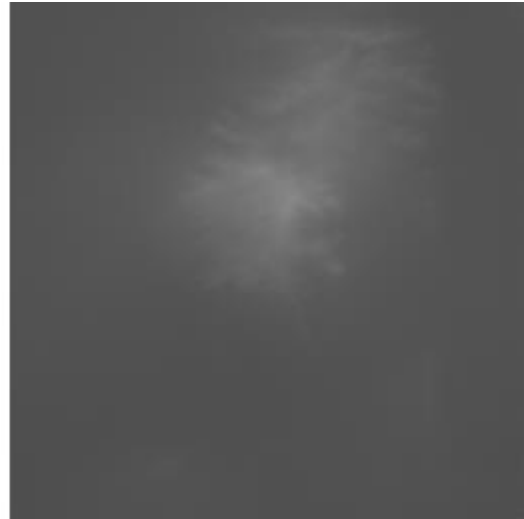
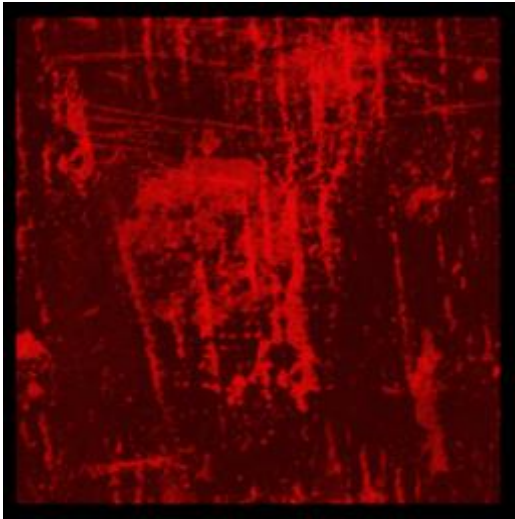


Figure A.24 - Input/Output testing pair. Right side is the input and left side is the output. As possible to observe, the output is focused on the higher exposure areas of the input, making the output unpredictable when using patterns

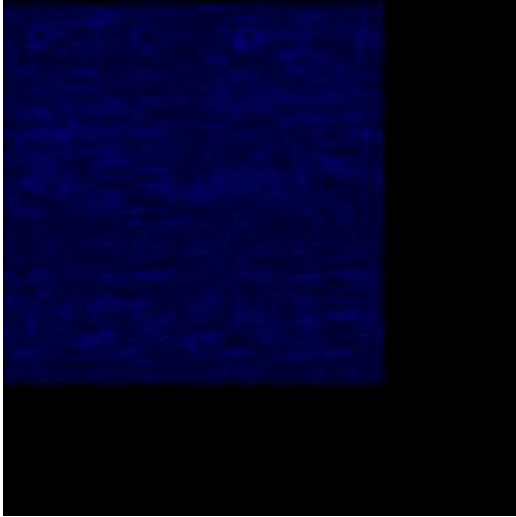


Figure A.24 - Input/Output testing pair. Right side is the input and left side is the output. Right edge of the pattern presents a lot of unpredictability.

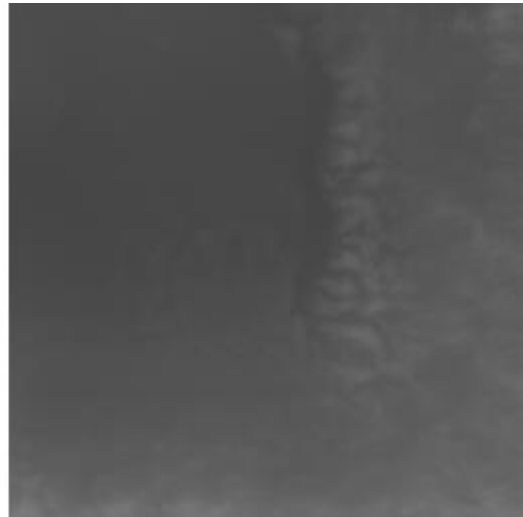
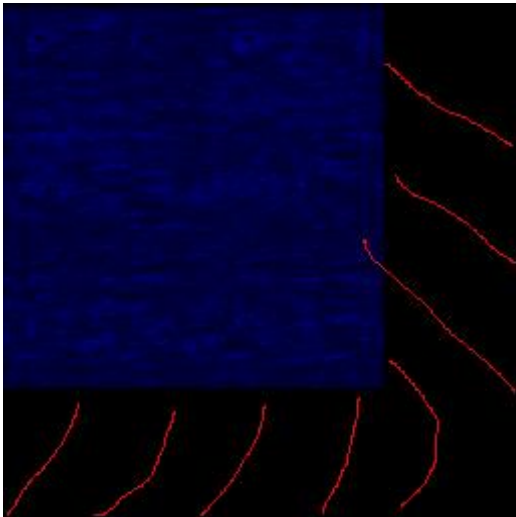


Figure A.25 - Input/Output testing pair. Right side is the input and left side is the output

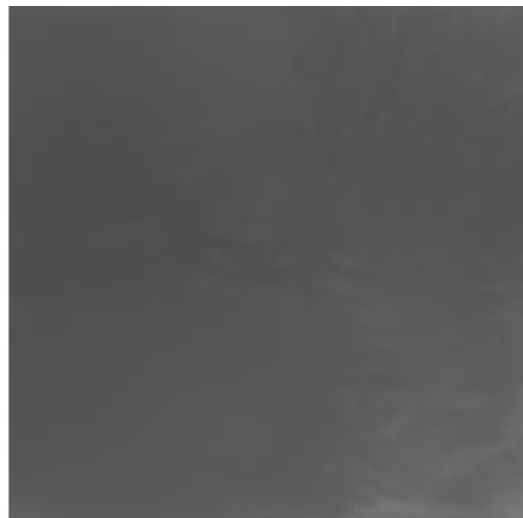
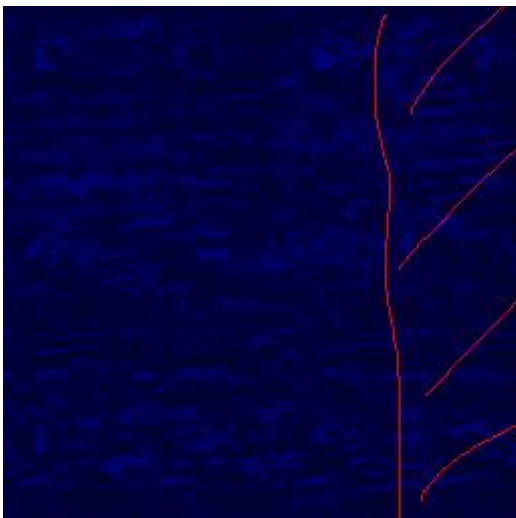


Figure A.26 - Input/Output testing pair. Right side is the input and left side is the output

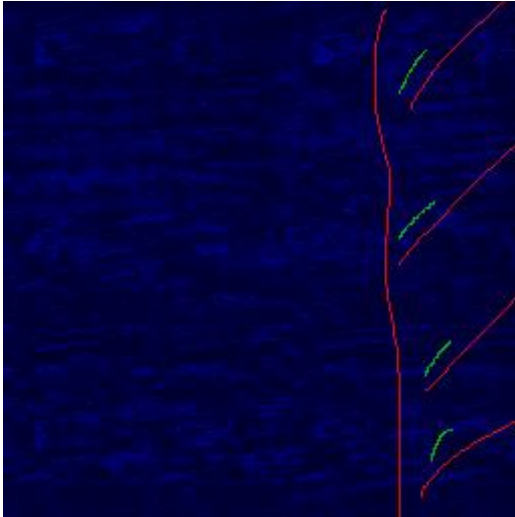


Figure A.27 - Input/Output testing pair. Right side is the input and left side is the output. Original green with lines corrupts the output. It is possible to see the pixelated effect.

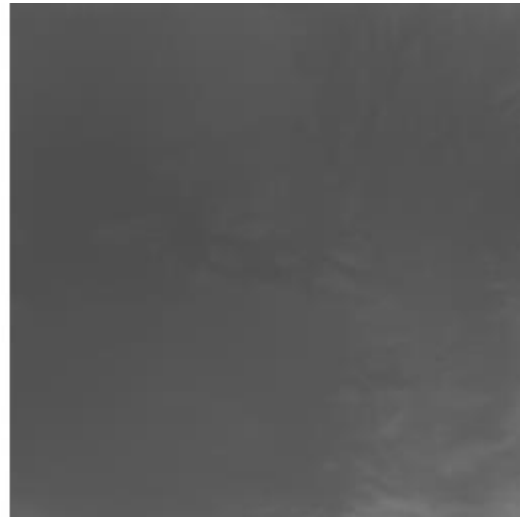
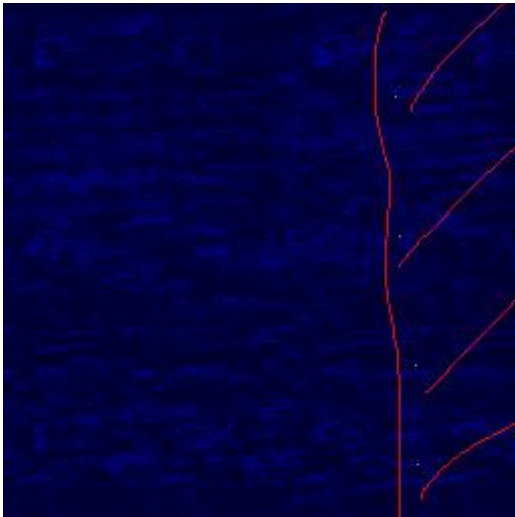


Figure A.28 - Input/Output testing pair. Right side is the input and left side is the output

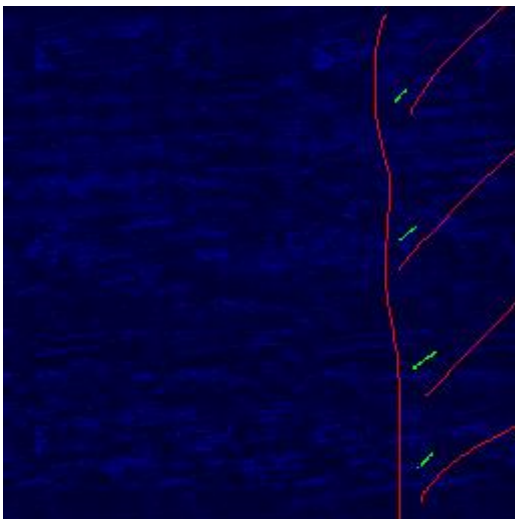


Figure A.29 - Input/Output testing pair. Right side is the input and left side is the output. Even with smaller green lines, the output gets corrupted.

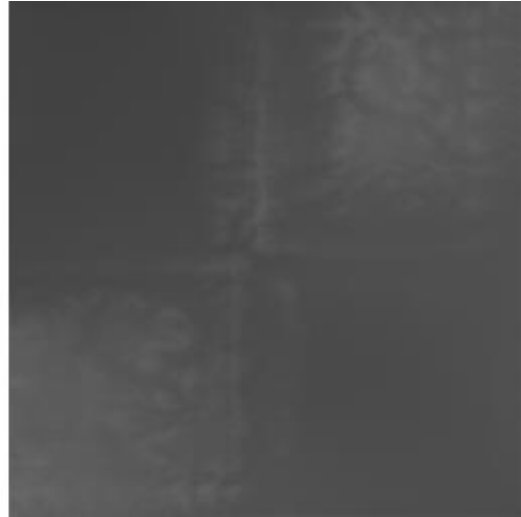
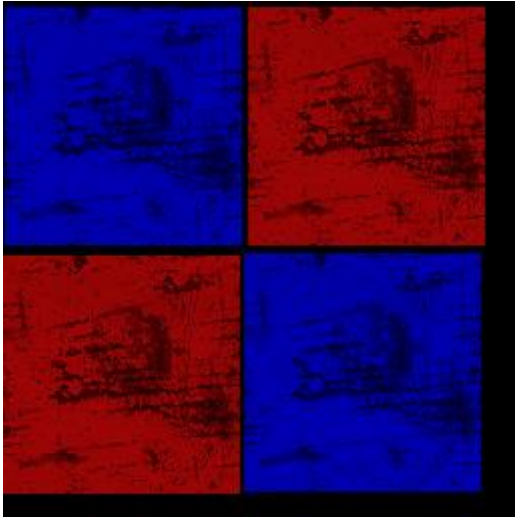


Figure A.30 - Input/Output testing pair. Right side is the input and left side is the output. Blue patterns present less detail than red patterns. Yet another layer of unpredictability.

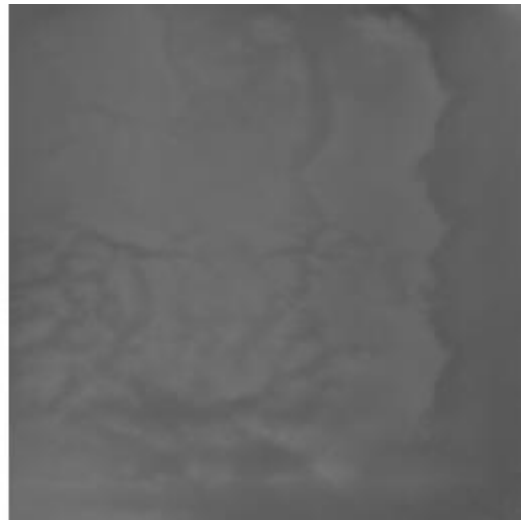
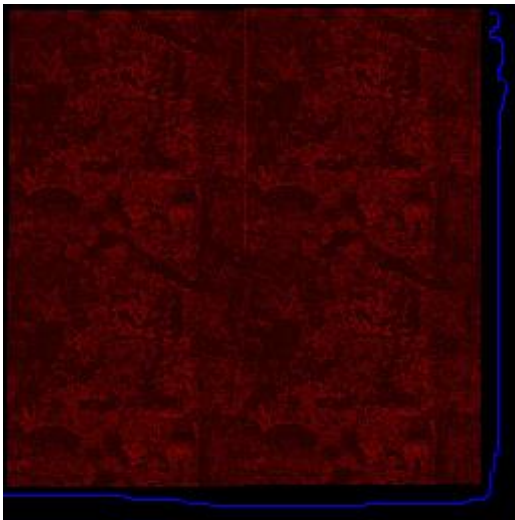


Figure A.31 - Input/Output testing pair. Right side is the input and left side is the output

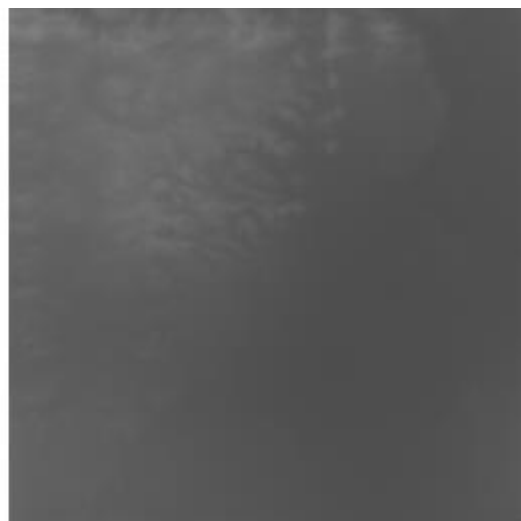
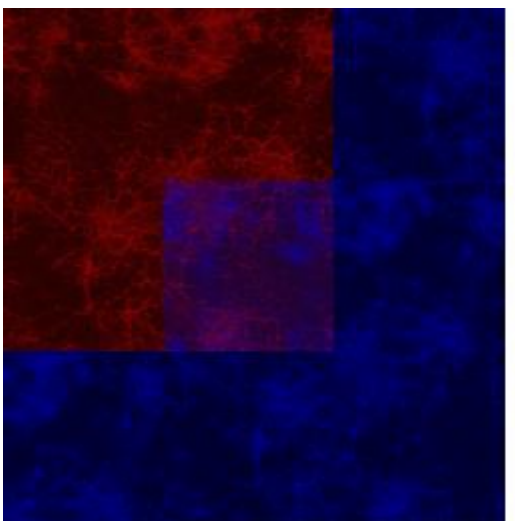


Figure A.31 - Input/Output testing pair. Right side is the input and left side is the output

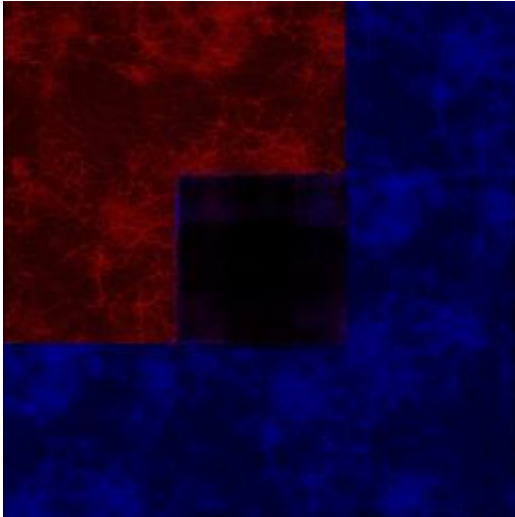


Figure A.32 - Input/Output testing pair. Right side is the input and left side is the output

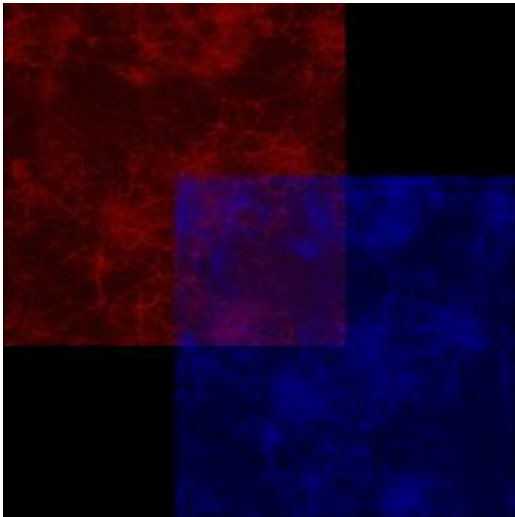


Figure A.33 - Input/Output testing pair. Right side is the input and left side is the output

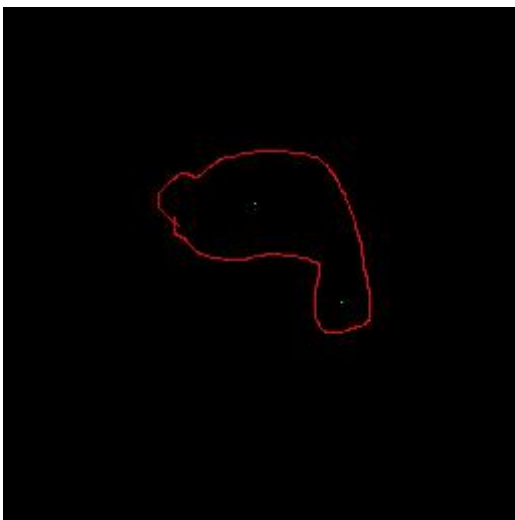


Figure A.34 - Input/Output testing pair. Right side is the input and left side is the output

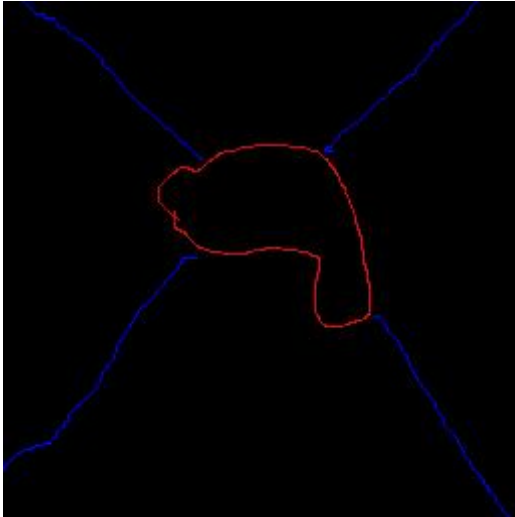


Figure A.35 - Input/Output testing pair. Right side is the input and left side is the output

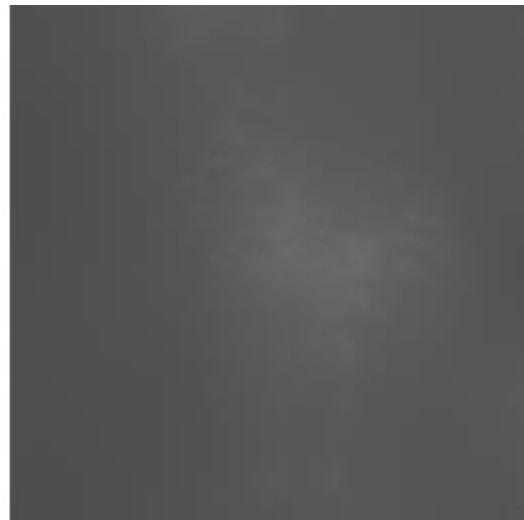
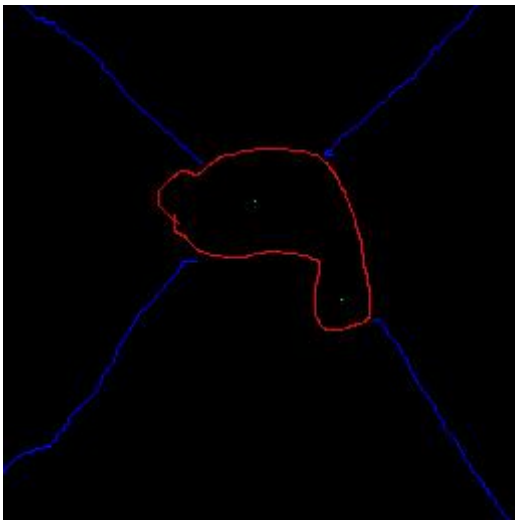


Figure A.36 - Input/Output testing pair. Right side is the input and left side is the output.

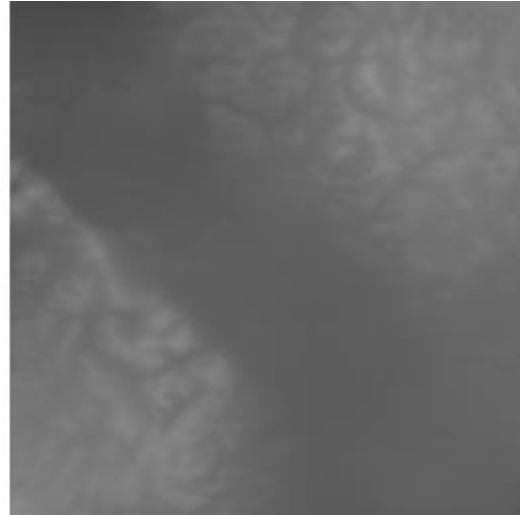
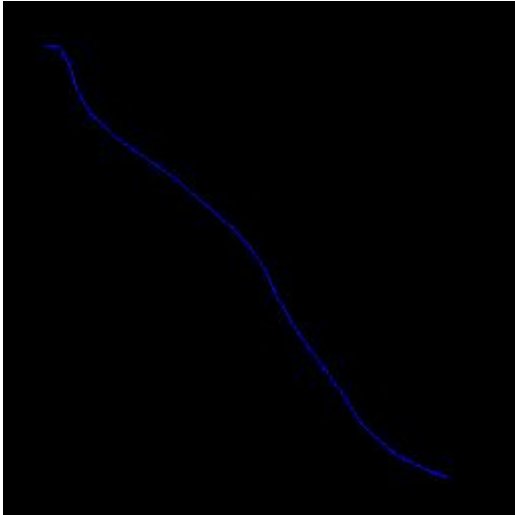


Figure A.37 - Input/Output testing pair. Right side is the input and left side is the output. Possible to see that the NN fills spaced without input with what it sees fit given the input present in the sketch.

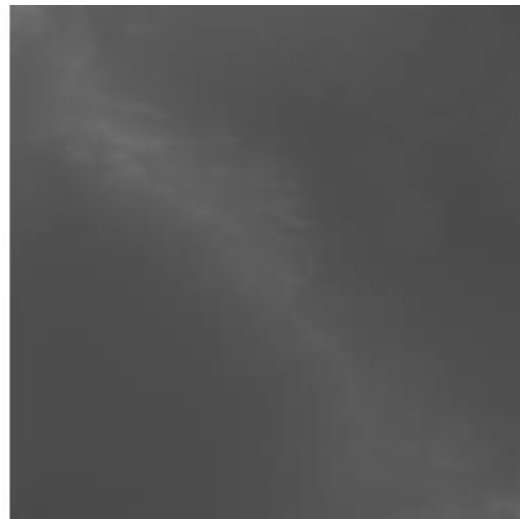
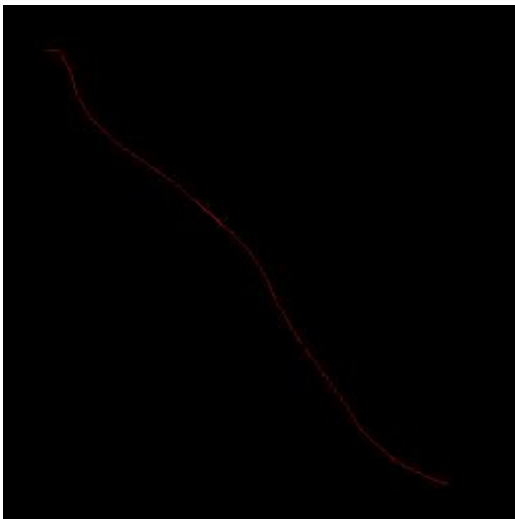


Figure A.38 - Input/Output testing pair. Right side is the input and left side is the output

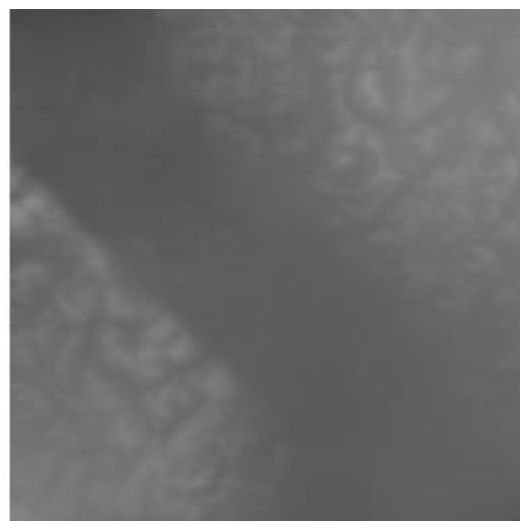
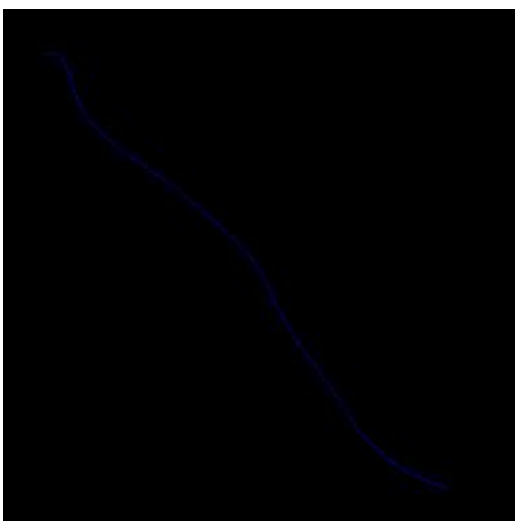


Figure A.39 - Input/Output testing pair. Right side is the input and left side is the output

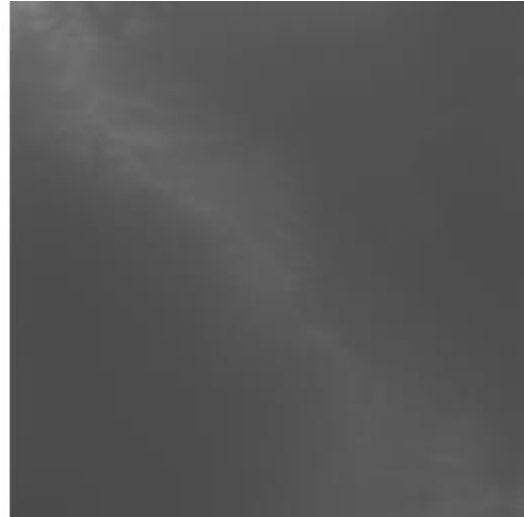
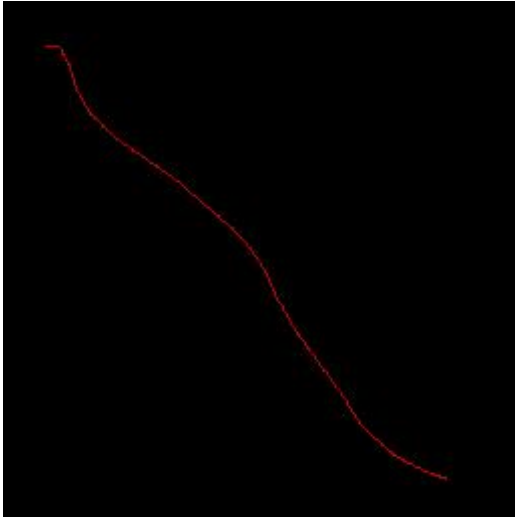


Figure A.40 - Input/Output testing pair. Right side is the input and left side is the output

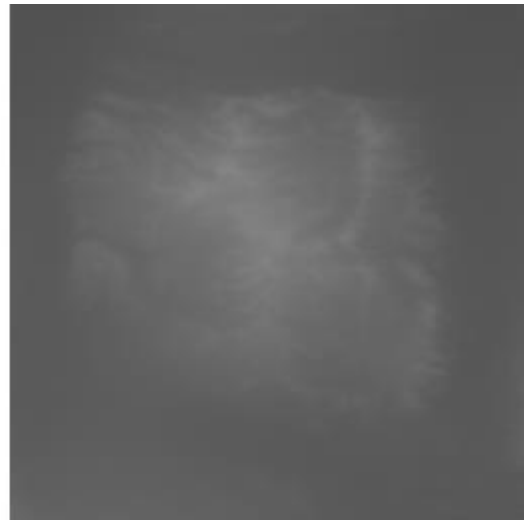
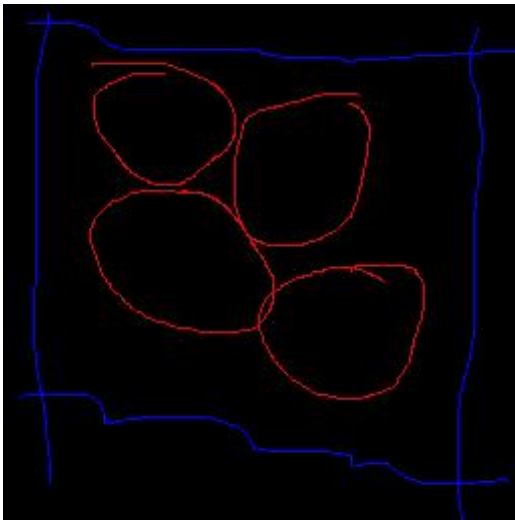


Figure A.41 - Input/Output testing pair. Right side is the input and left side is the output

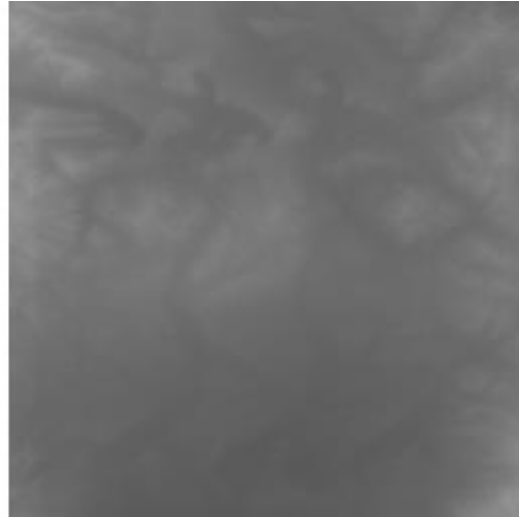
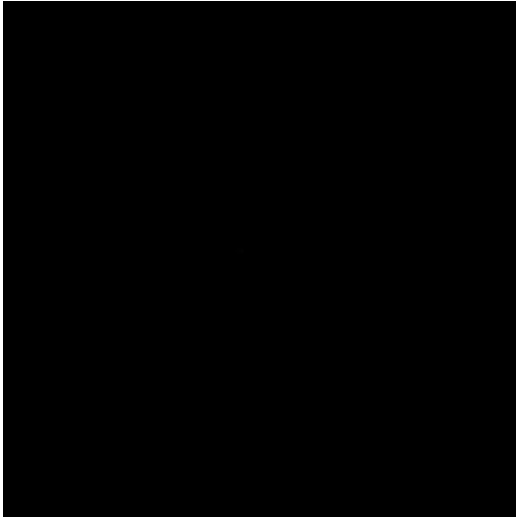


Figure A.42 - Input/Output testing pair. Right side is the input and left side is the output. Input with minimal information will be extremely similar to the output of A.12

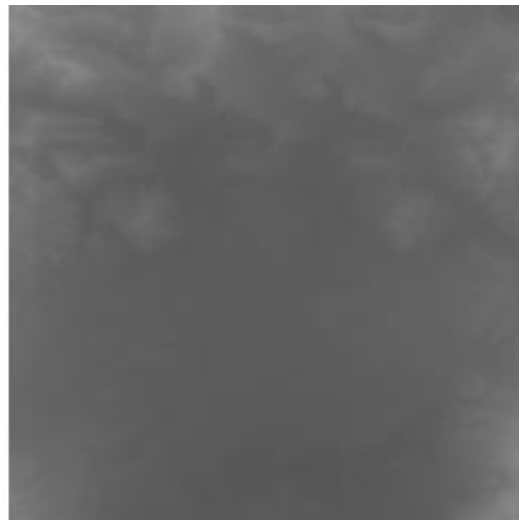
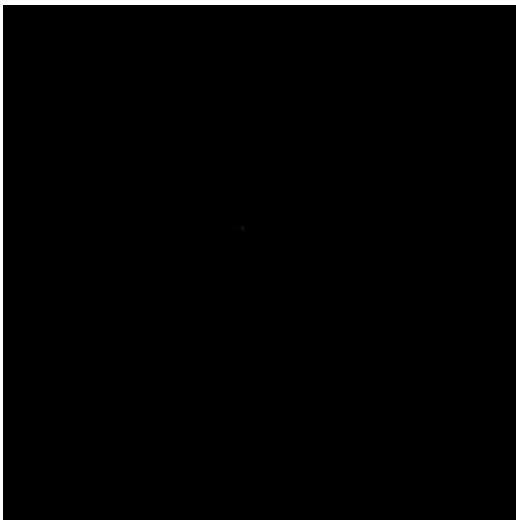


Figure A.43 - Input/Output testing pair. Right side is the input and left side is the output. Input with minimal information will be extremely similar to the output of A.12



Figure A.44 - Input/Output testing pair. Right side is the input and left side is the output.

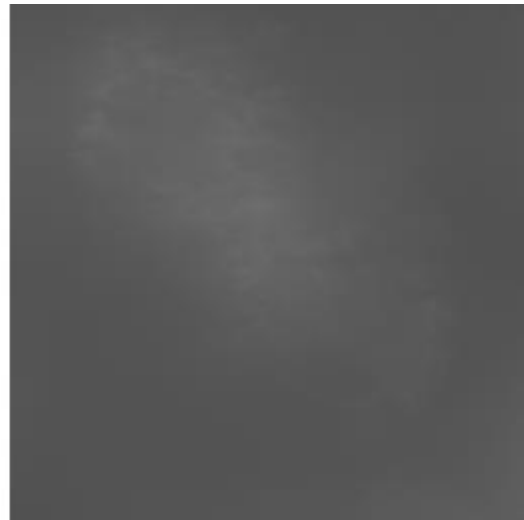
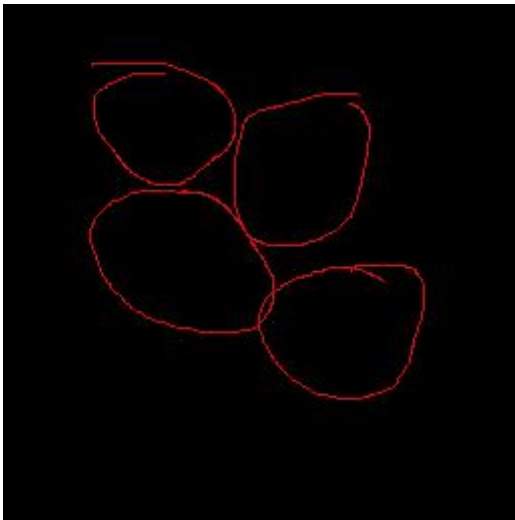


Figure A.45 - Input/Output testing pair. Right side is the input and left side is the output.

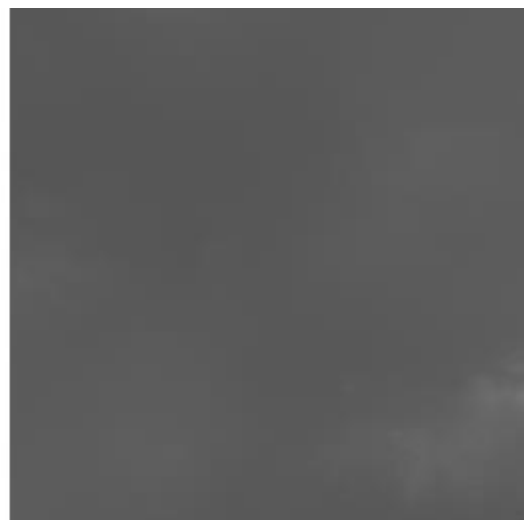


Figure A.46 - Input/Output testing pair. Right side is the input and left side is the output

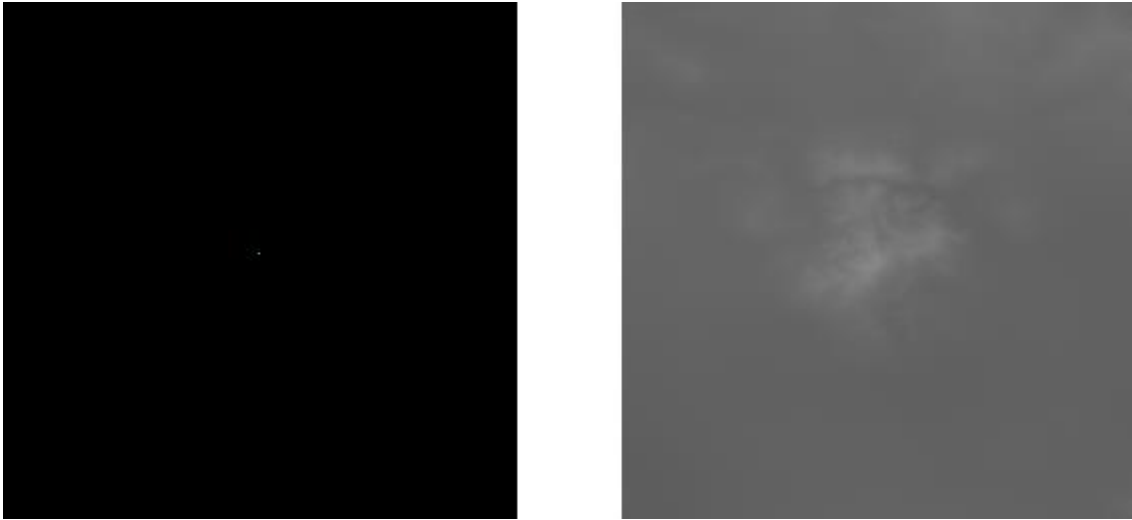


Figure A.47 - Input/Output testing pair. Right side is the input and left side is the output.

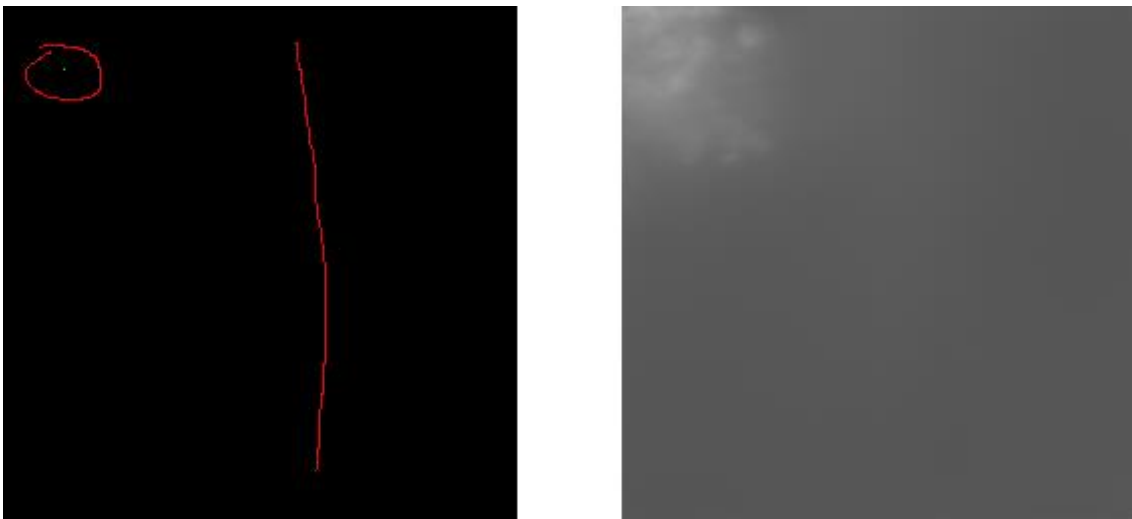


Figure A.48 - Input/Output testing pair. Right side is the input and left side is the output. It is possible to see that the NN attributes a bigger weight to green, because it ignores the majority of the input in favor of attributing the entire of height to the structure that has a green dot.

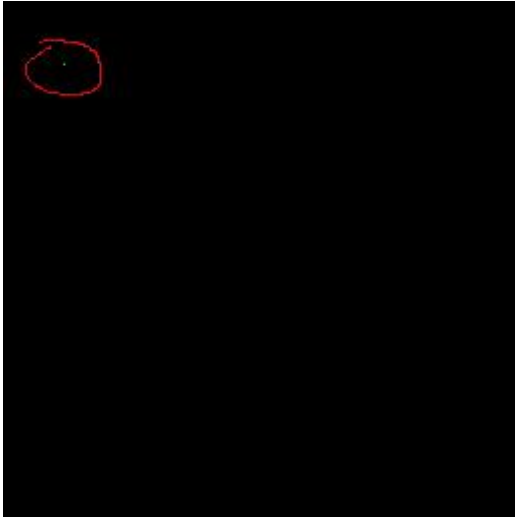


Figure A.49 - Input/Output testing pair. Right side is the input and left side is the output.

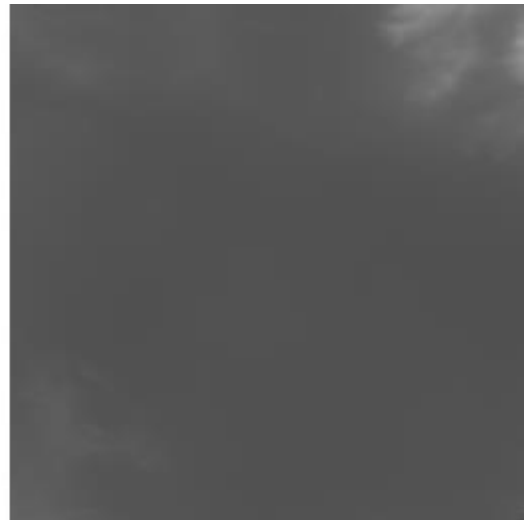


Figure A.50 - Input/Output testing pair. Right side is the input and left side is the output.

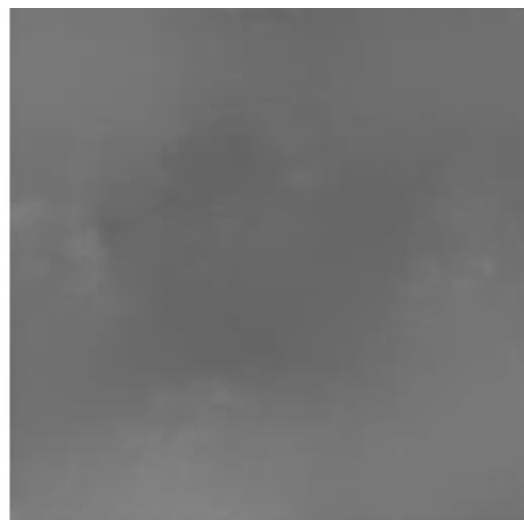


Figure A.51 - Input/Output testing pair. Right side is the input and left side is the output.

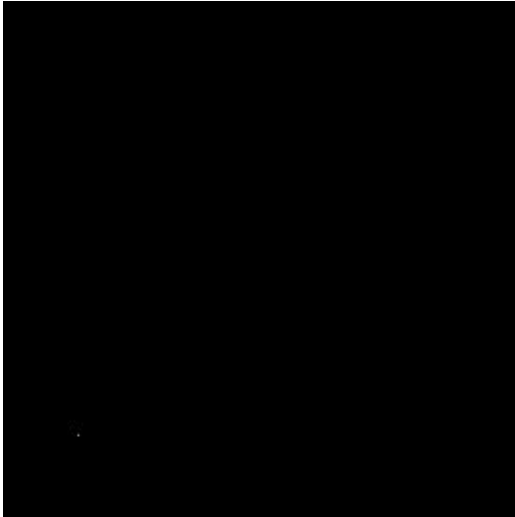


Figure A.52 - Input/Output testing pair. Right side is the input and left side is the output.

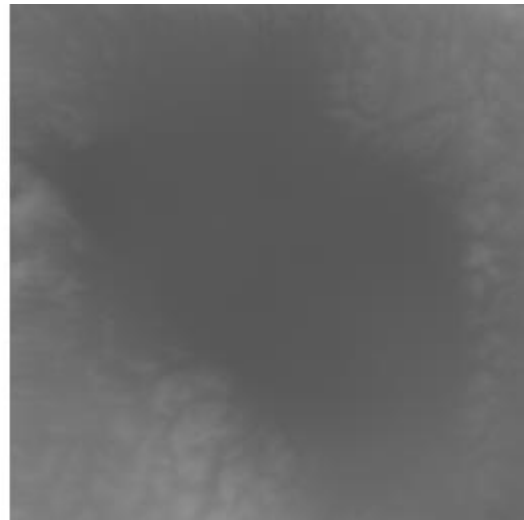
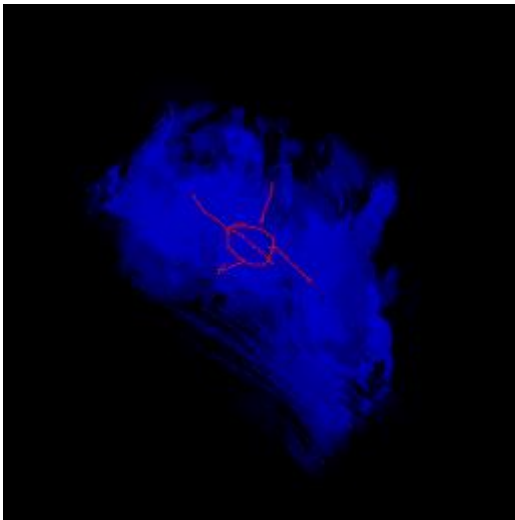


Figure A.53 - Input/Output testing pair. Right side is the input and left side is the output.

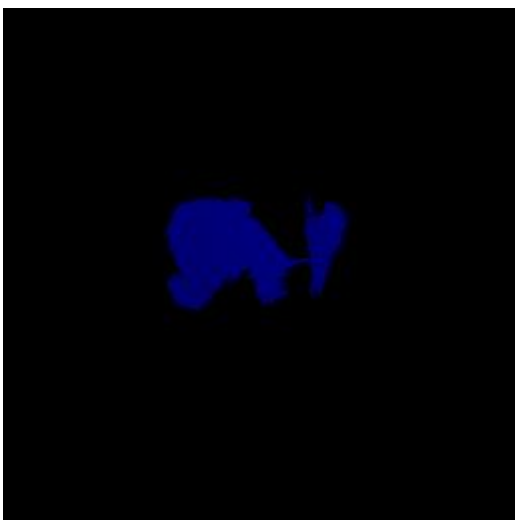


Figure A.54 - Input/Output testing pair. Right side is the input and left side is the output.

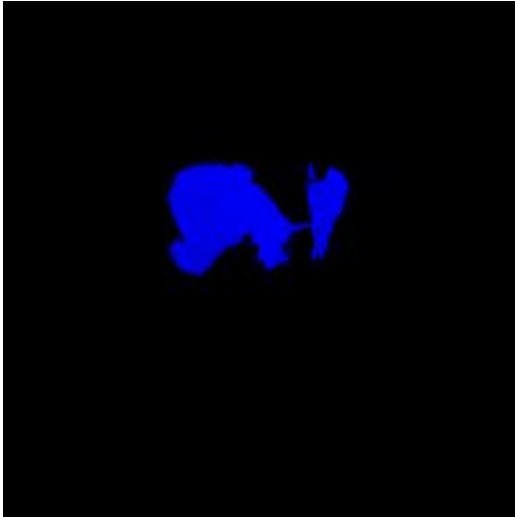


Figure A.55 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output due to overexposure of the blue color.

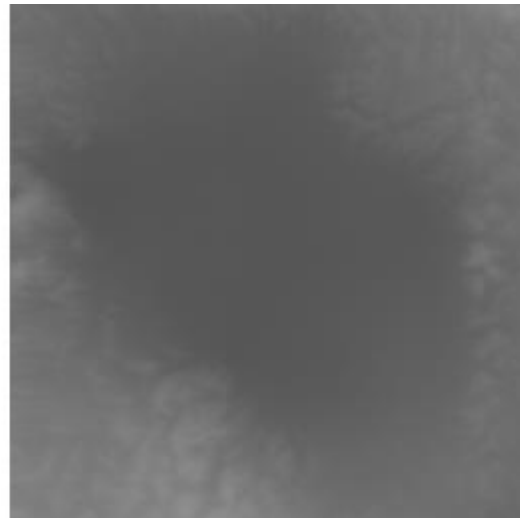
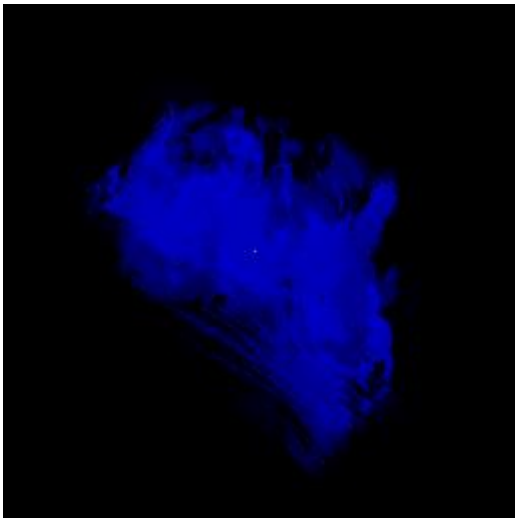


Figure A.56 - Input/Output testing pair. Right side is the input and left side is the output.

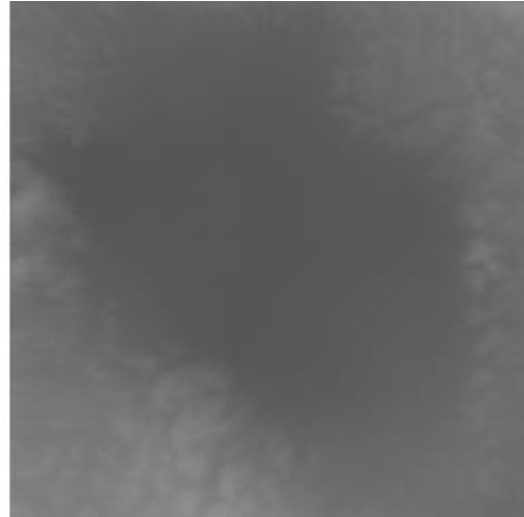
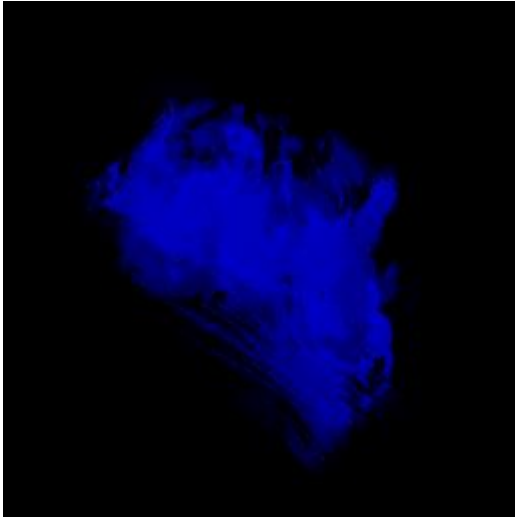


Figure A.57 - Input/Output testing pair. Right side is the input and left side is the output.

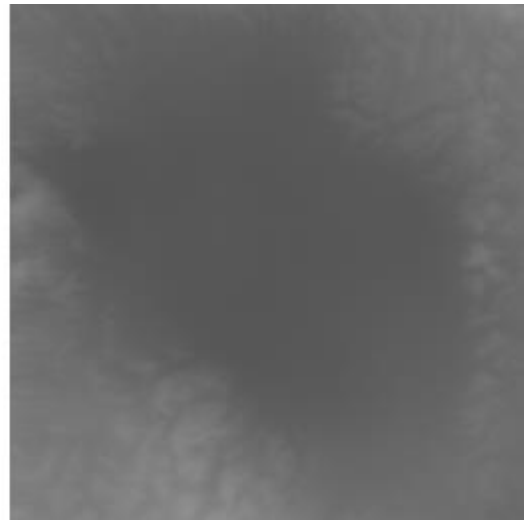
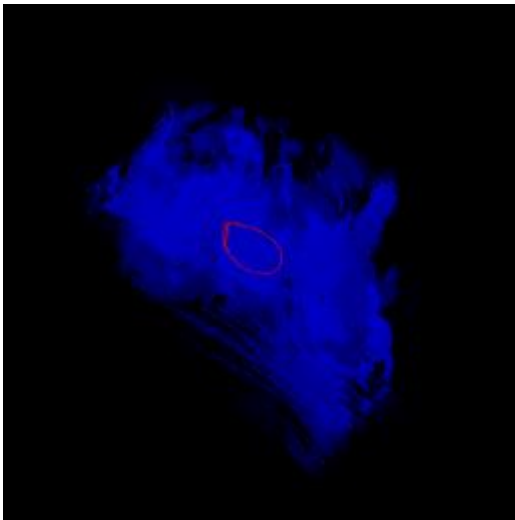


Figure A.58 - Input/Output testing pair. Right side is the input and left side is the output.

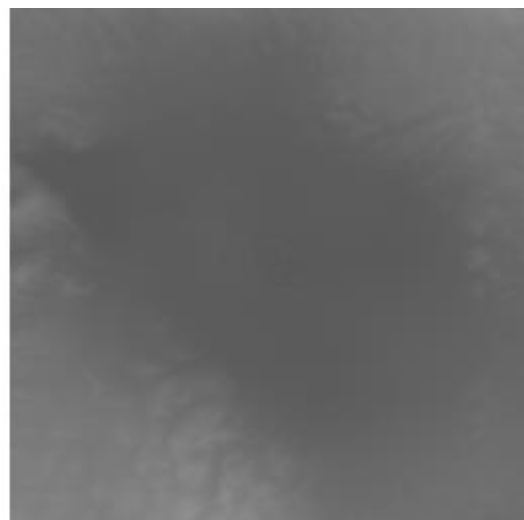
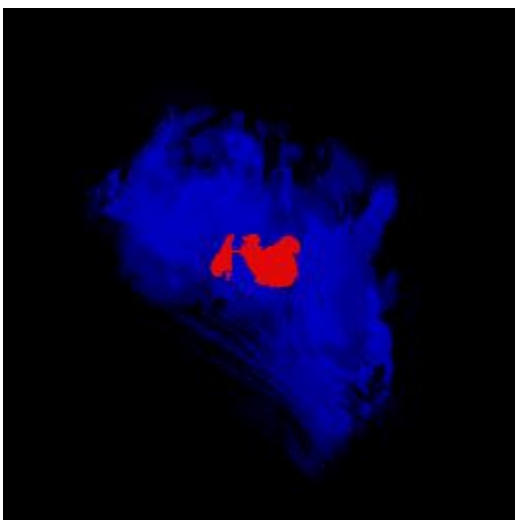


Figure A.59 - Input/Output testing pair. Right side is the input and left side is the output.

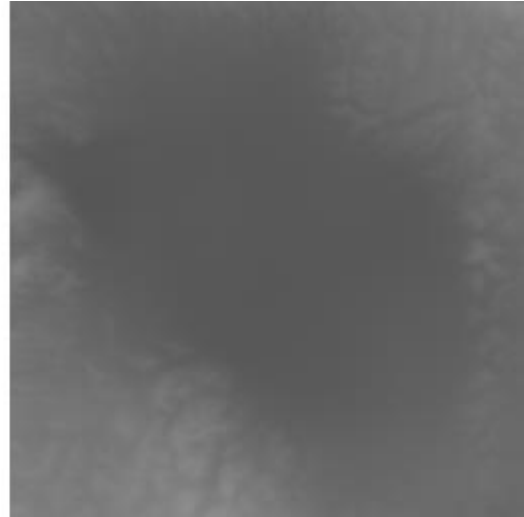
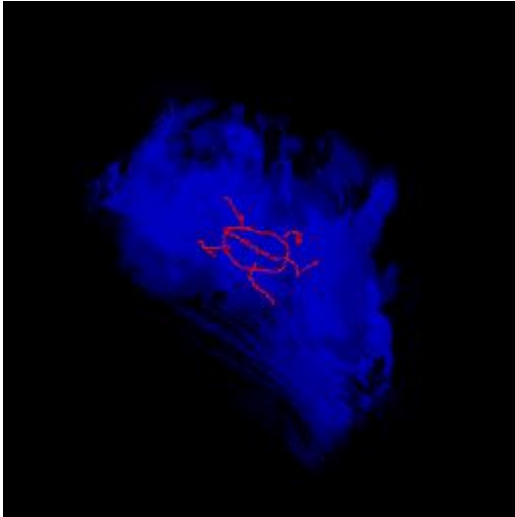


Figure A.60 - Input/Output testing pair. Right side is the input and left side is the output.

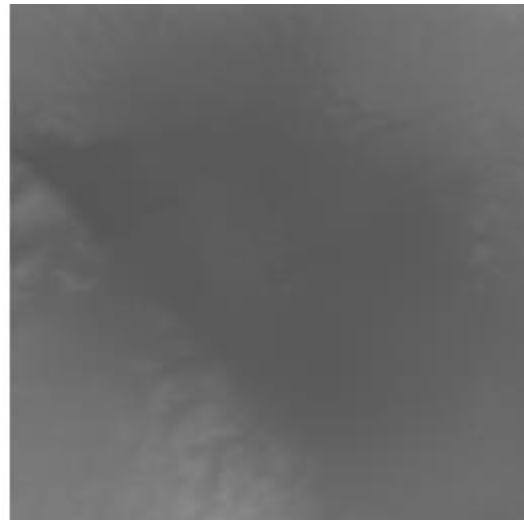
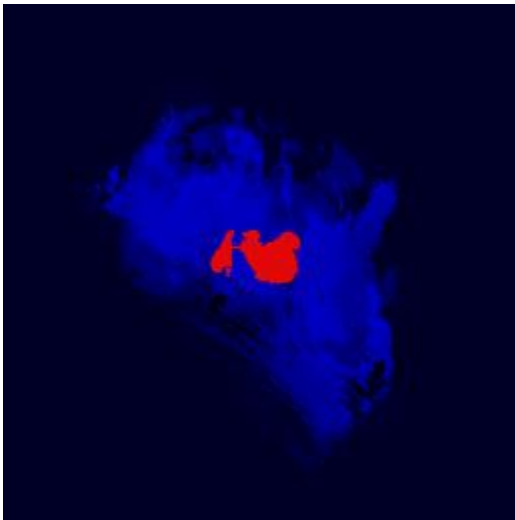


Figure A.61 - Input/Output testing pair. Right side is the input and left side is the output.

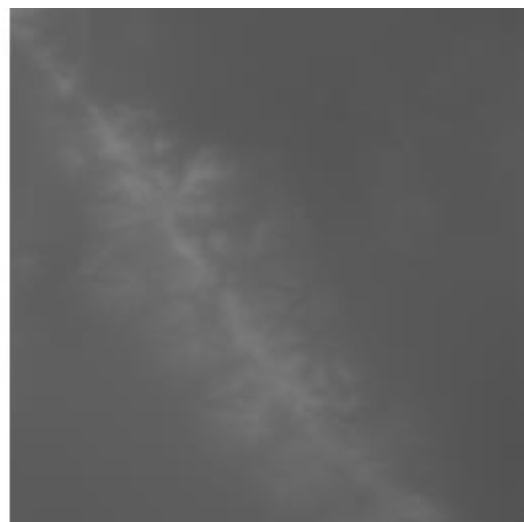
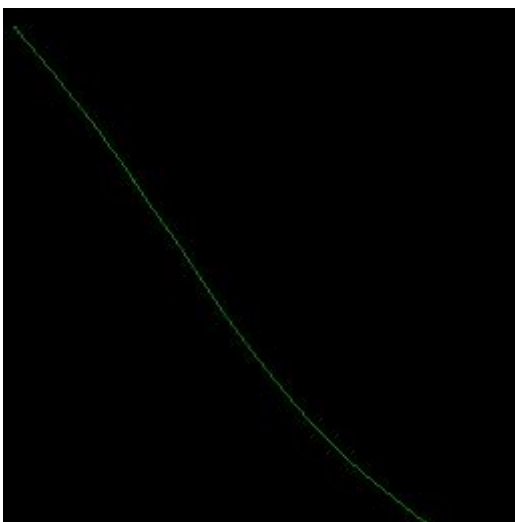


Figure A.62 - Input/Output testing pair. Right side is the input and left side is the output.

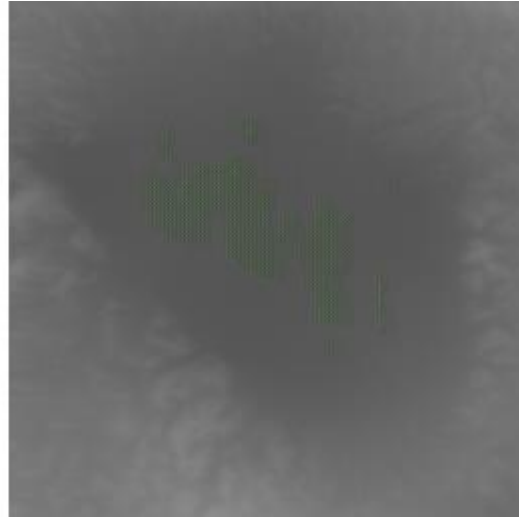
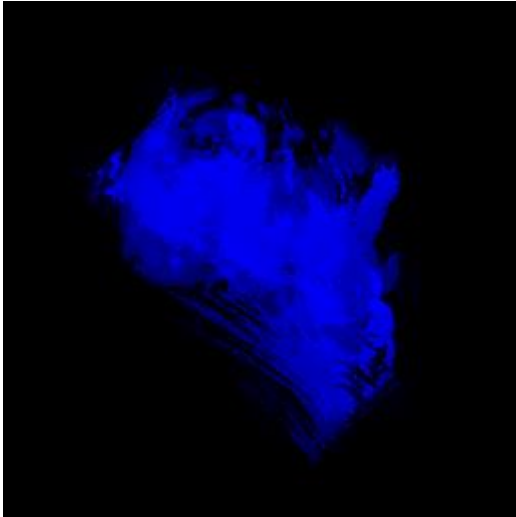


Figure A.53 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output due to color overexposure.

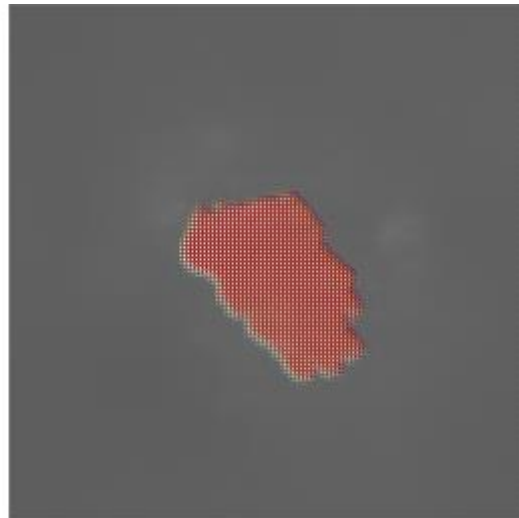
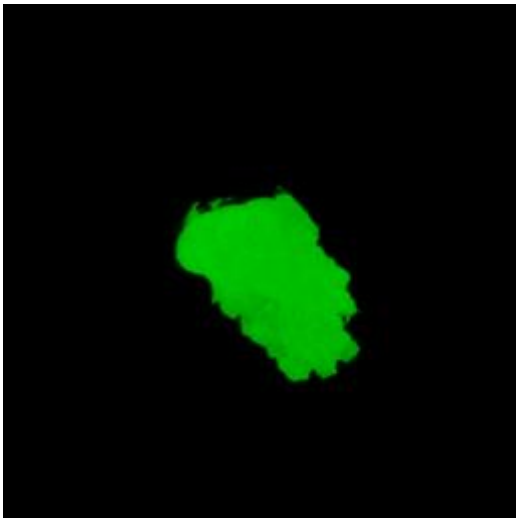


Figure A.54 - Input/Output testing pair. Right side is the input and left side is the output. Clear corruption due to green overexposure.

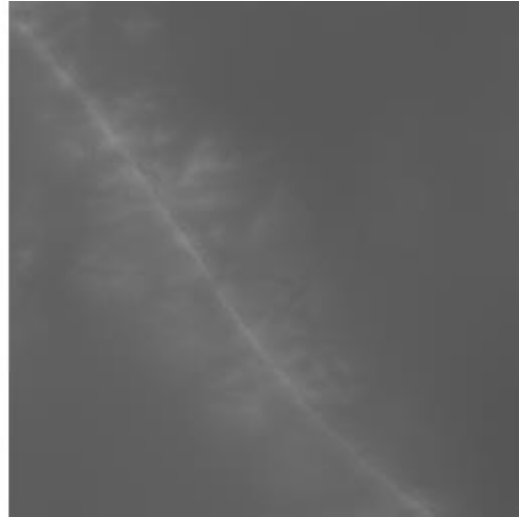
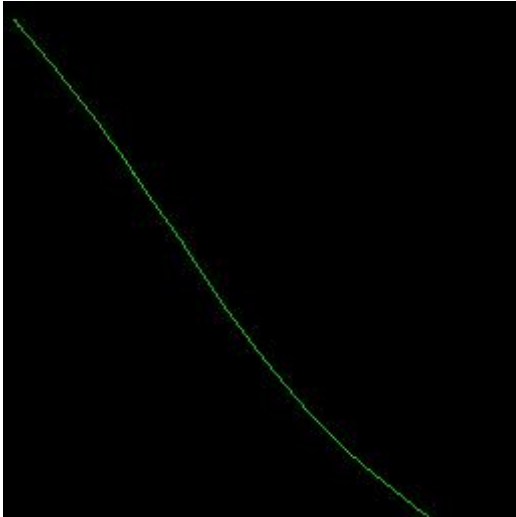


Figure A.55 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output.
Note the pixilation of the colors.

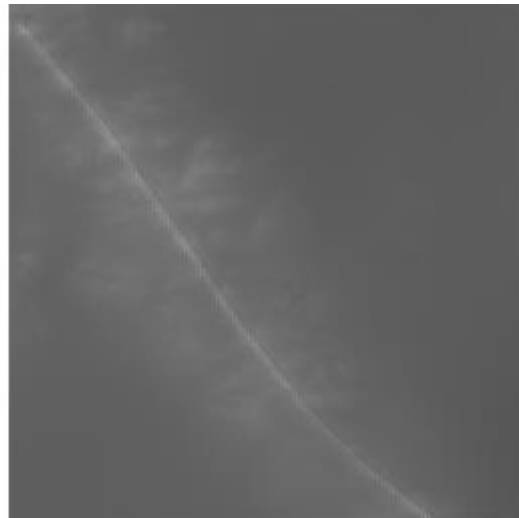
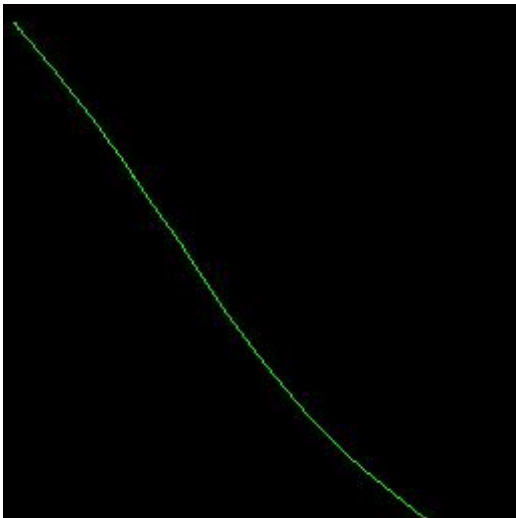


Figure A.56 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output.
Note the pixilation of the colors.

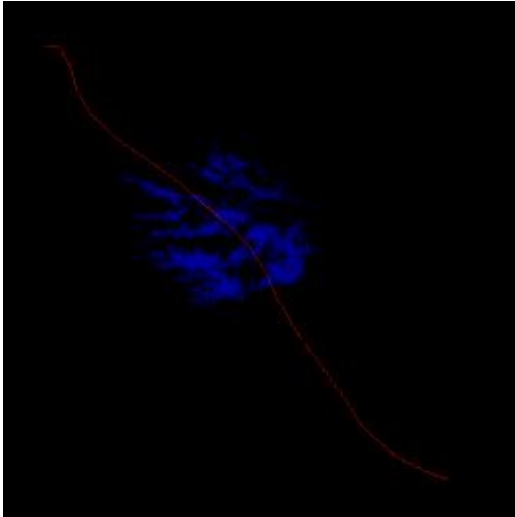


Figure A.57 - Input/Output testing pair. Right side is the input and left side is the output.

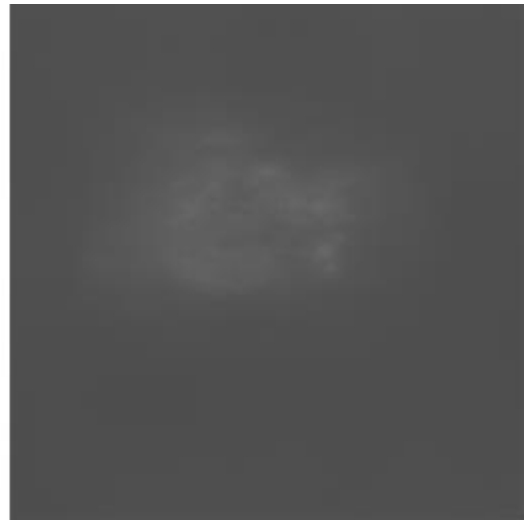
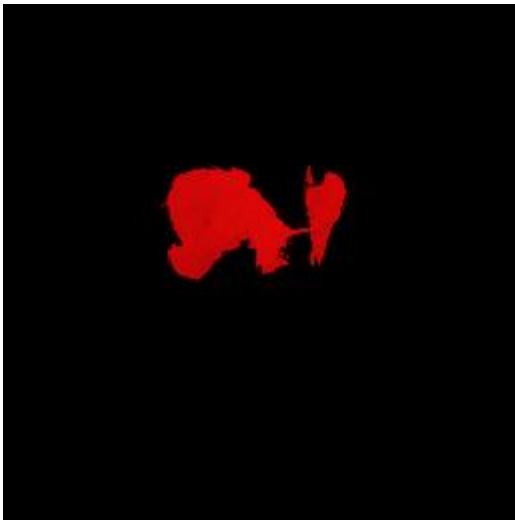


Figure A.57 - Input/Output testing pair. Right side is the input and left side is the output.

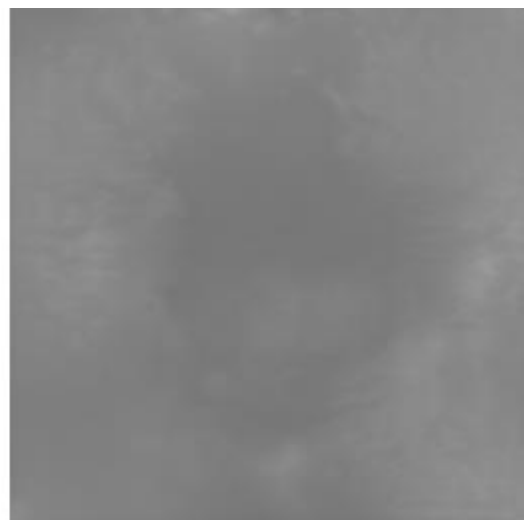
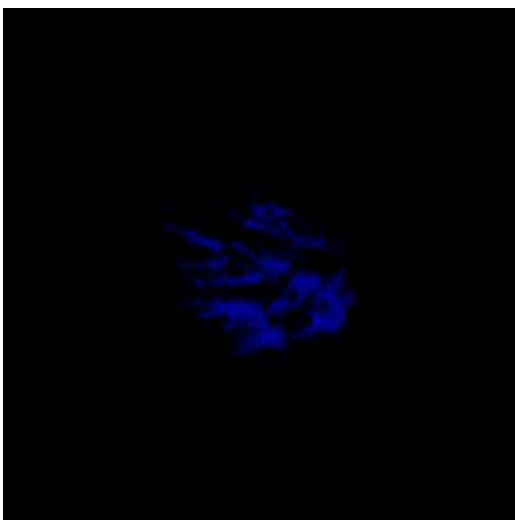


Figure A.58 - Input/Output testing pair. Right side is the input and left side is the output.

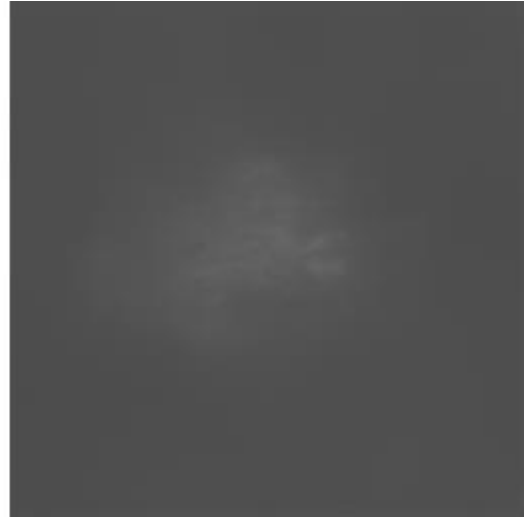


Figure A.59 - Input/Output testing pair. Right side is the input and left side is the output.

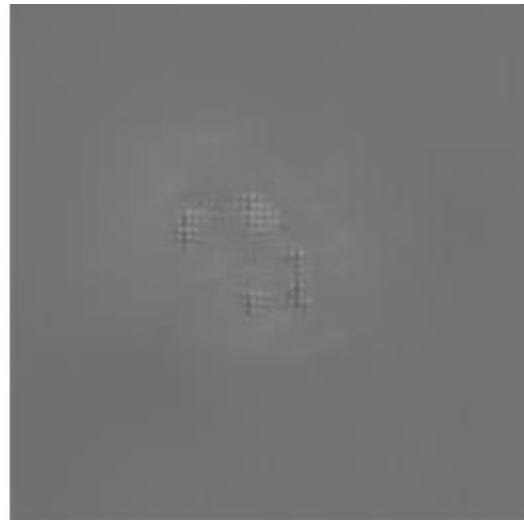
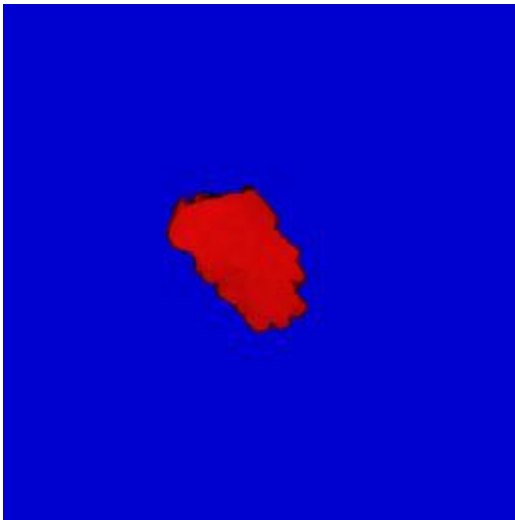


Figure A.60 - Input/Output testing pair. Right side is the input and left side is the output. A.60 to A.62 use the same red color value. This output seems to be the most heavily corrupted of the three, due to the abrupt color shift from deep blue to bright red.

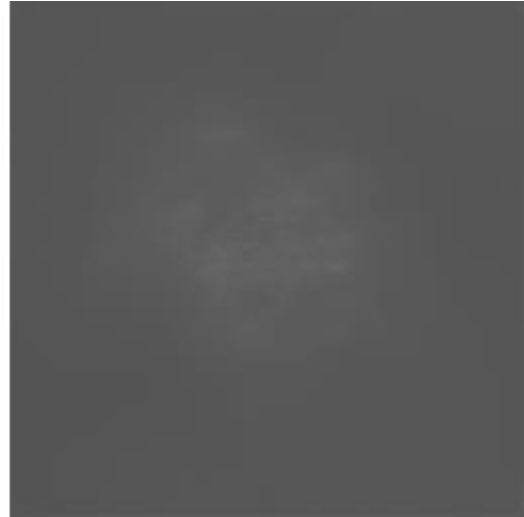


Figure A.61 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output.

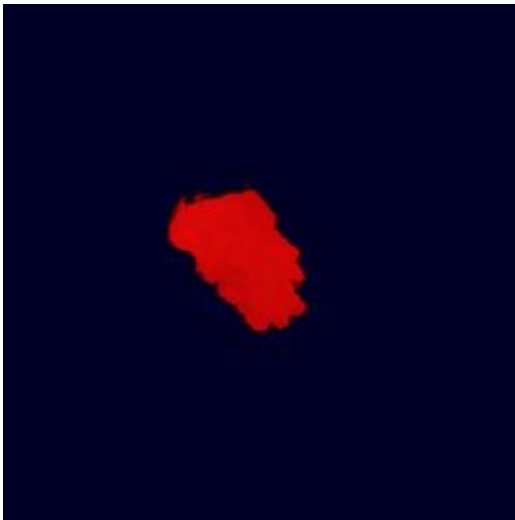


Figure A.62 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output.

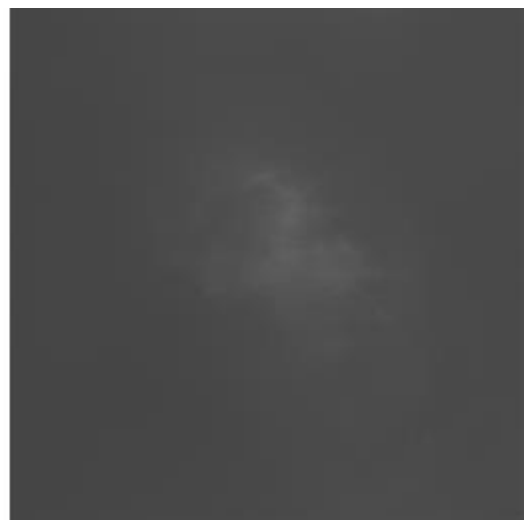
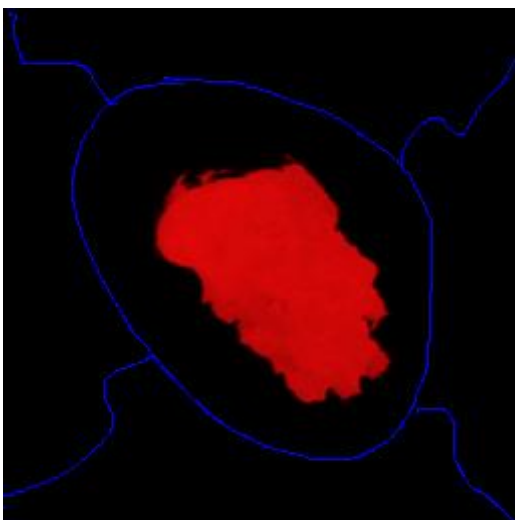


Figure A.63 - Input/Output testing pair. Right side is the input and left side is the output.

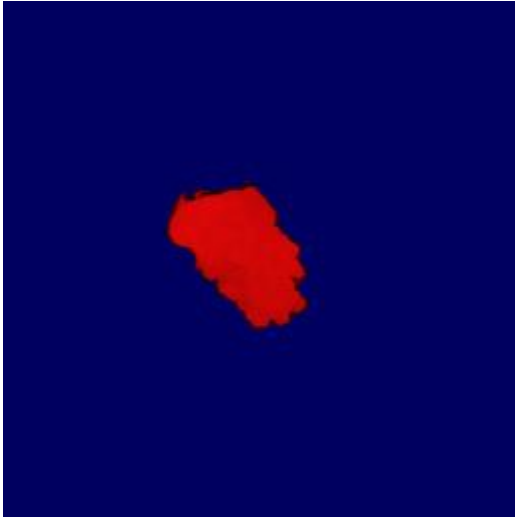


Figure A.64 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output.

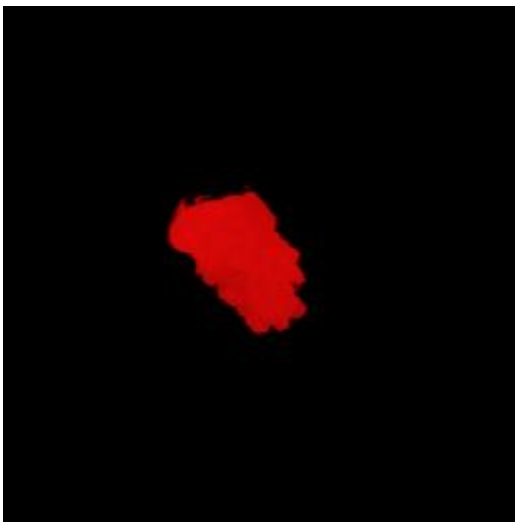


Figure A.65 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output.

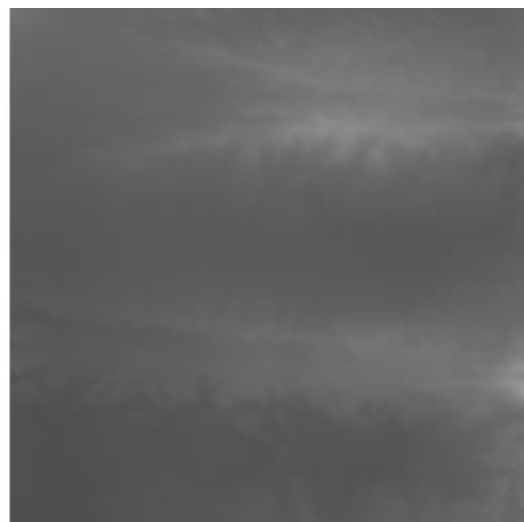
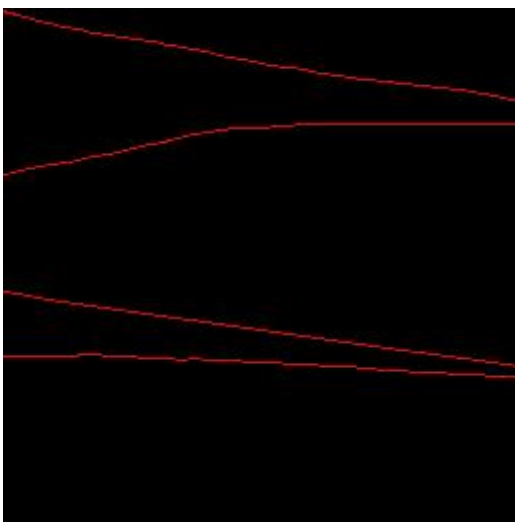


Figure A.66 - Input/Output testing pair. Right side is the input and left side is the output.

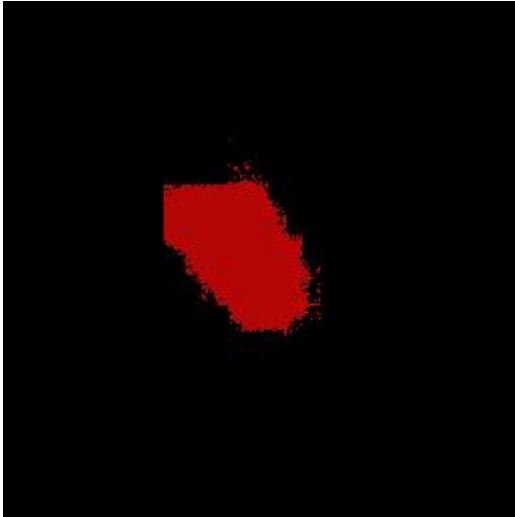


Figure A.67 - Input/Output testing pair. Right side is the input and left side is the output.

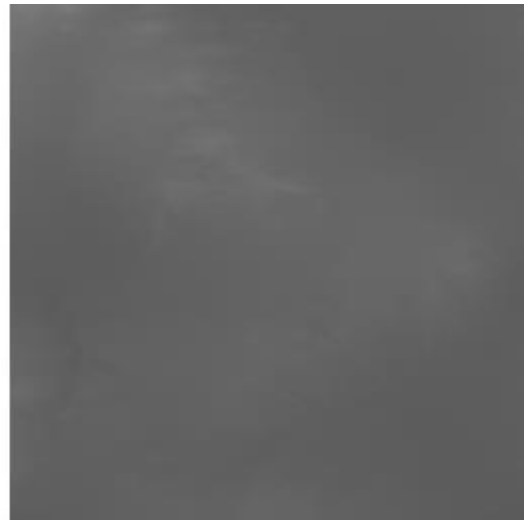


Figure A.68 - Input/Output testing pair. Right side is the input and left side is the output.

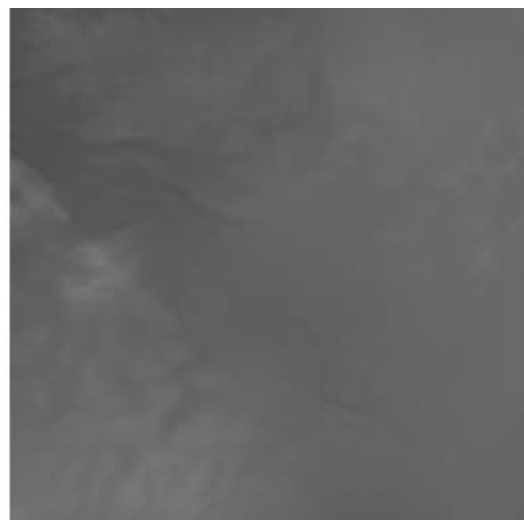
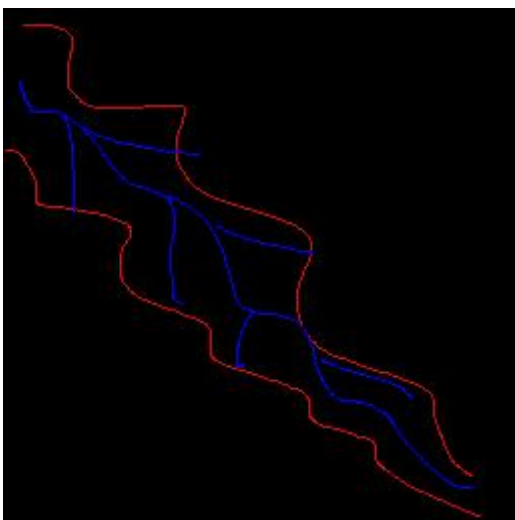


Figure A.69 - Input/Output testing pair. Right side is the input and left side is the output.

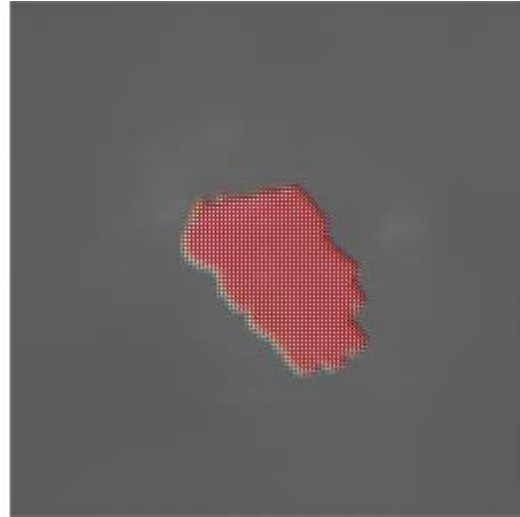
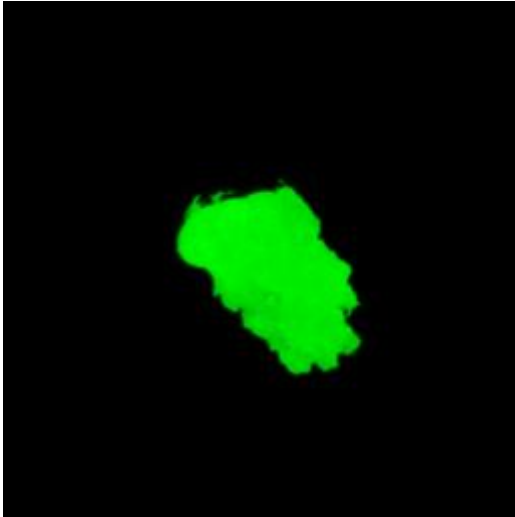


Figure A.70 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output

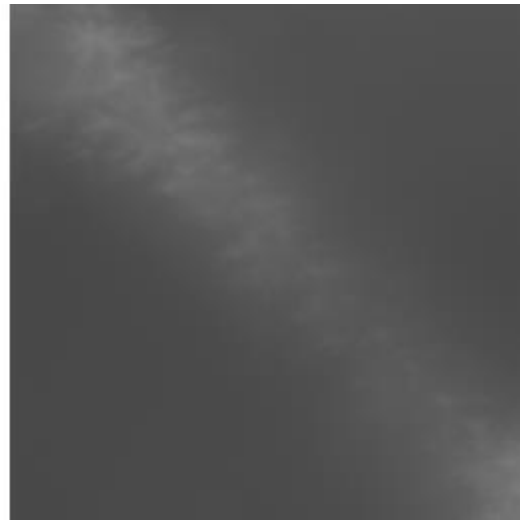
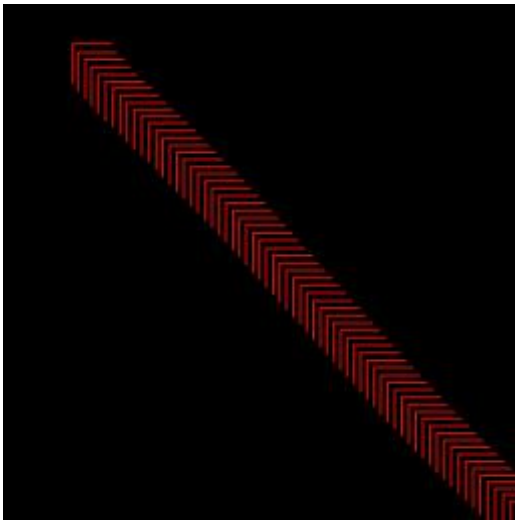


Figure A.71 - Input/Output testing pair. Right side is the input and left side is the output.

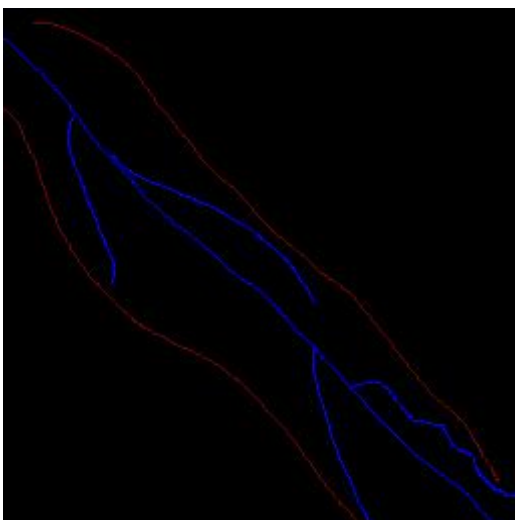


Figure A.72 - Input/Output testing pair. Right side is the input and left side is the output.

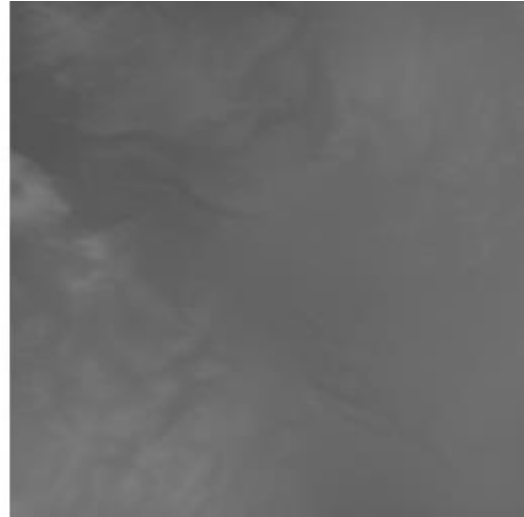
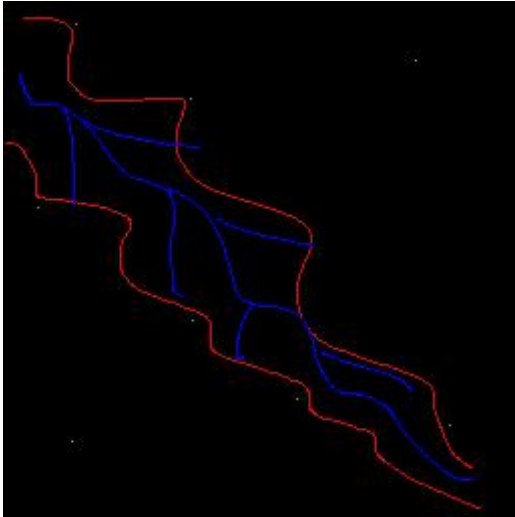


Figure A.73 - Input/Output testing pair. Right side is the input and left side is the output.

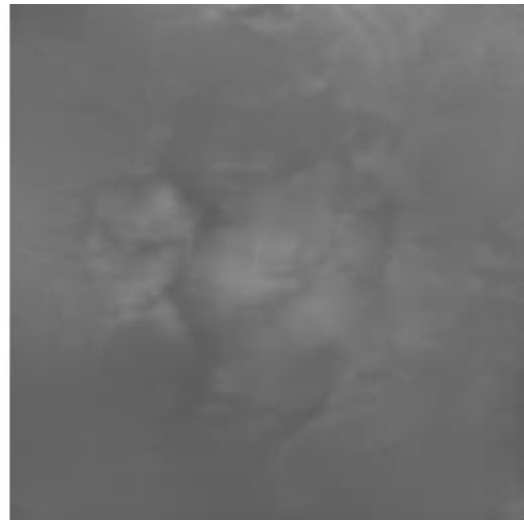
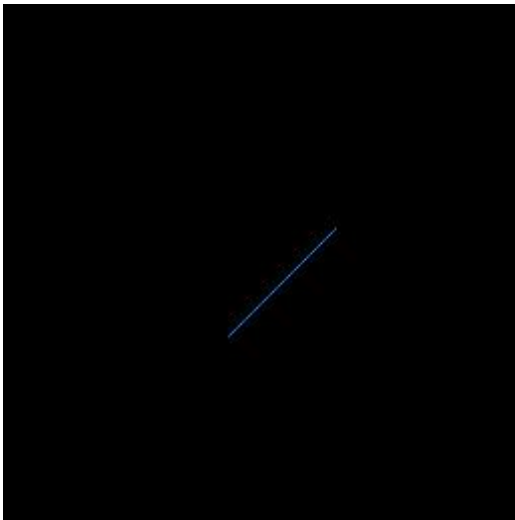


Figure A.74 - Input/Output testing pair. Right side is the input and left side is the output.

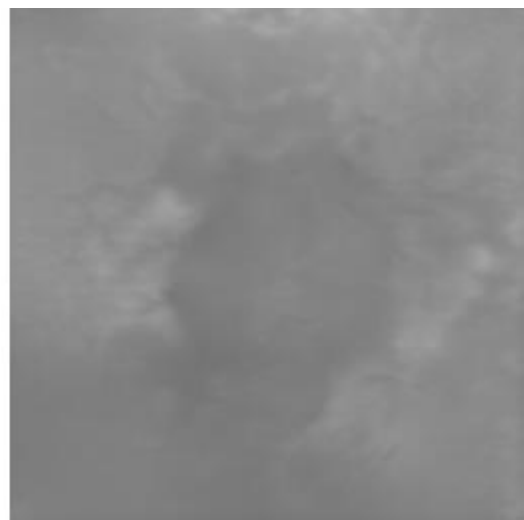
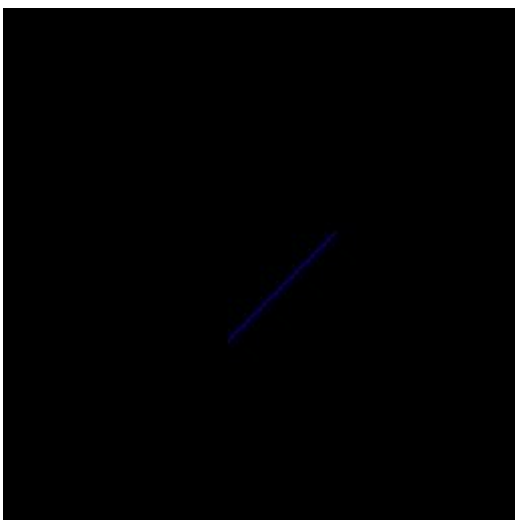


Figure A.75 - Input/Output testing pair. Right side is the input and left side is the output.

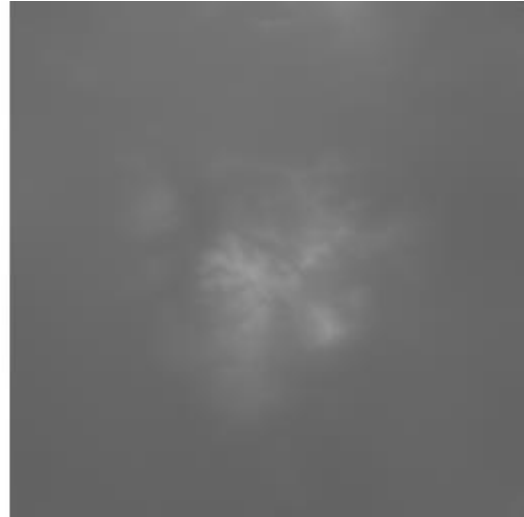
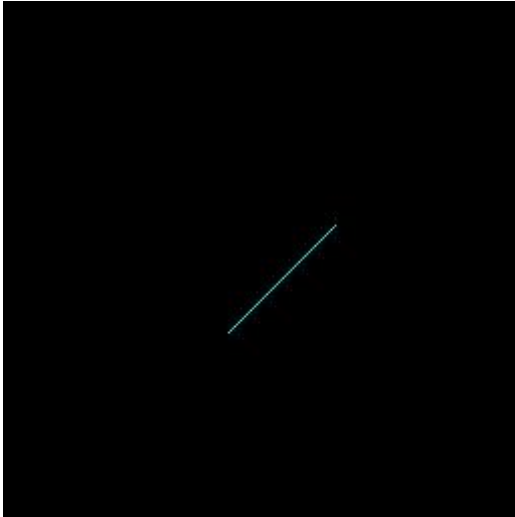


Figure A.76 - Input/Output testing pair. Right side is the input and left side is the output.

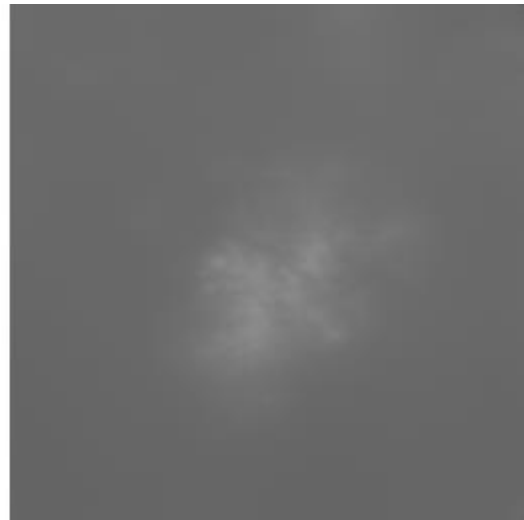
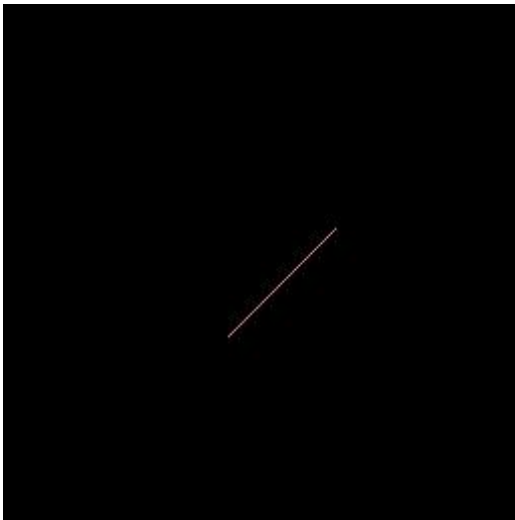


Figure A.77 - Input/Output testing pair. Right side is the input and left side is the output.

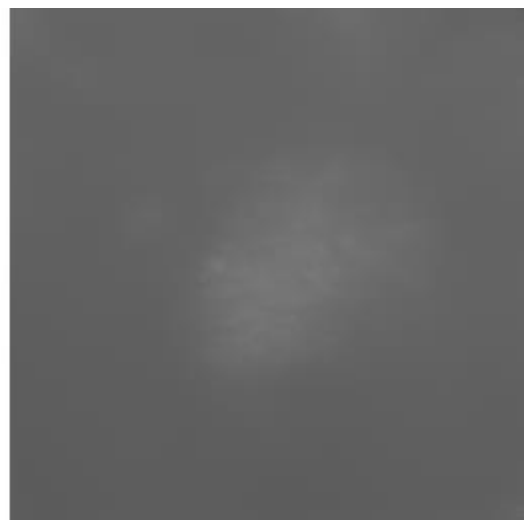
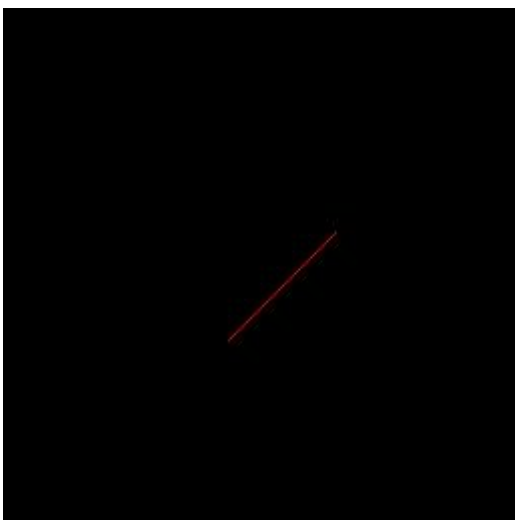


Figure A.77 - Input/Output testing pair. Right side is the input and left side is the output.

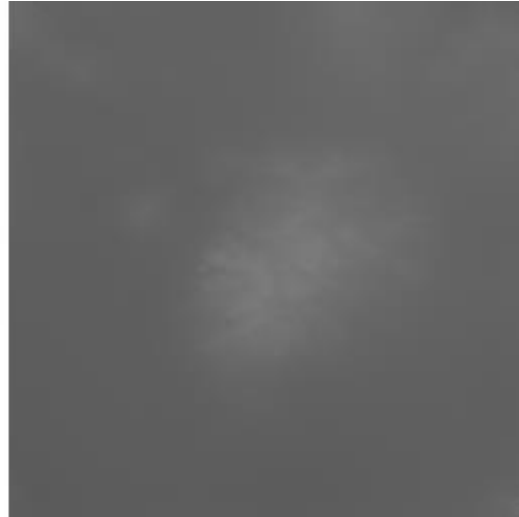
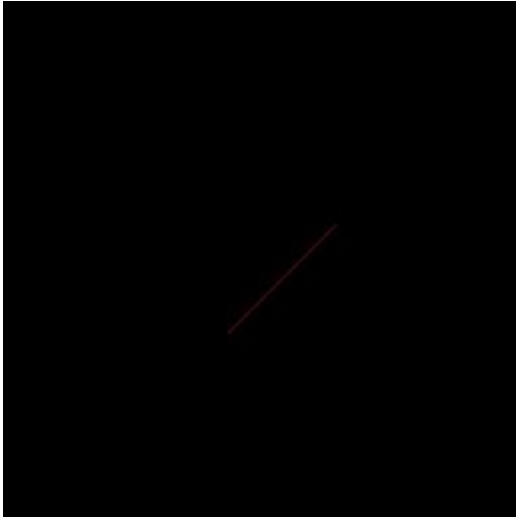


Figure A.78 - Input/Output testing pair. Right side is the input and left side is the output.

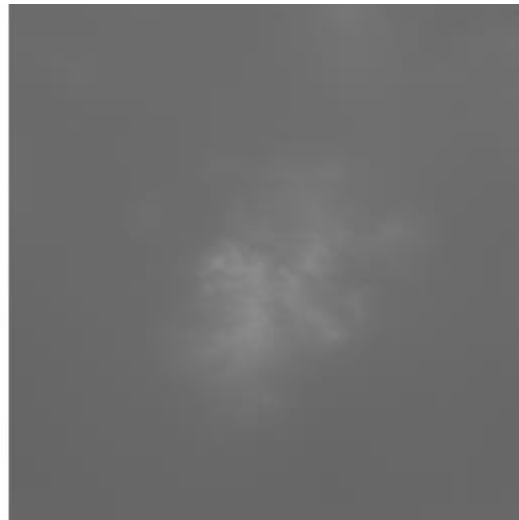
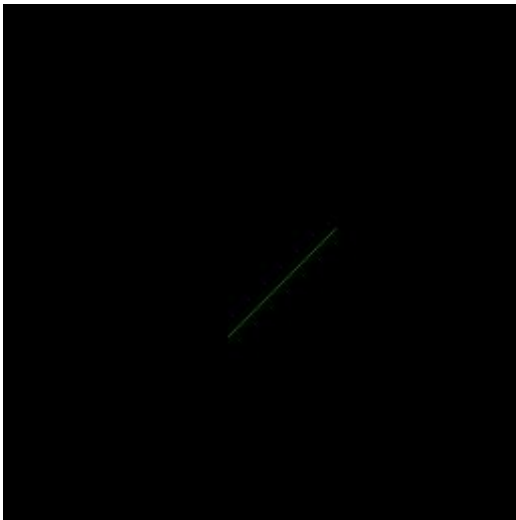


Figure A.79 - Input/Output testing pair. Right side is the input and left side is the output.

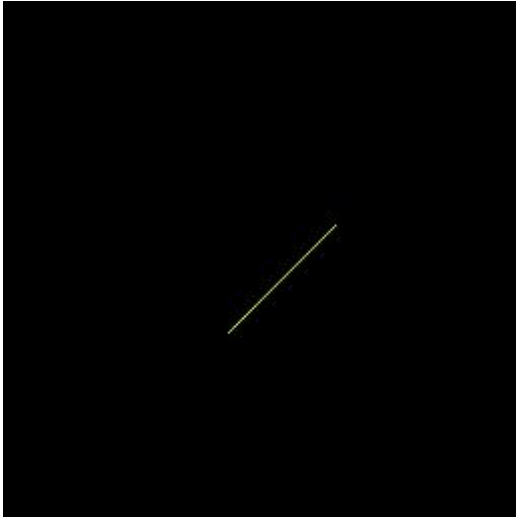


Figure A.80 - Input/Output testing pair. Right side is the input and left side is the output. Corrupted output. Yellow did not work as a new color. Retraining necessary.

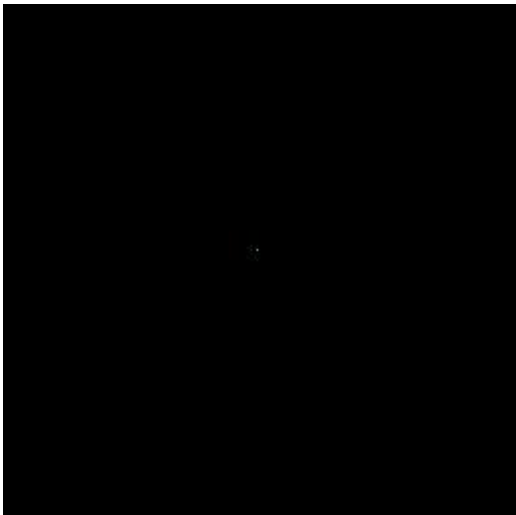


Figure A.81 - Input/Output testing pair. Right side is the input and left side is the output.

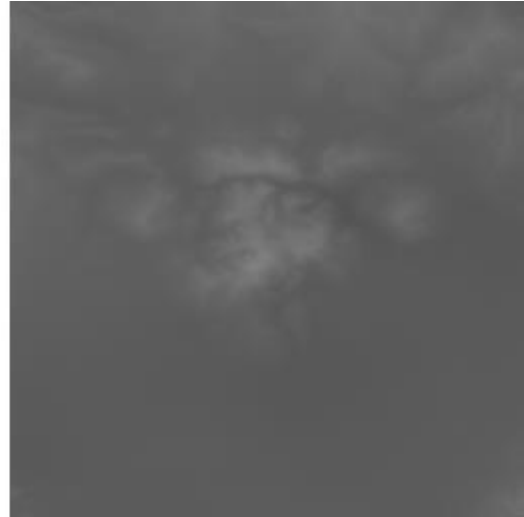
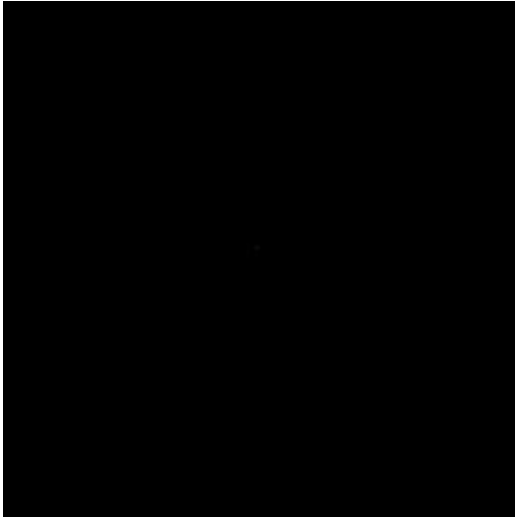


Figure A.82 - Input/Output testing pair. Right side is the input and left side is the output.

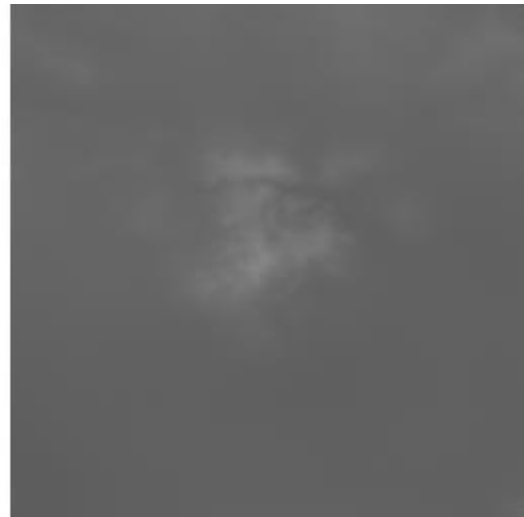
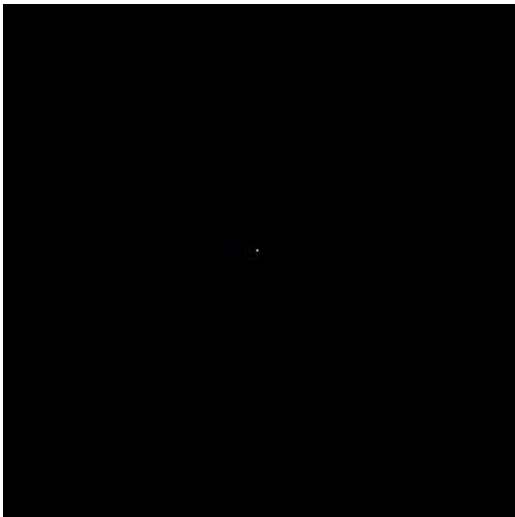


Figure A.83 - Input/Output testing pair. Right side is the input and left side is the output.

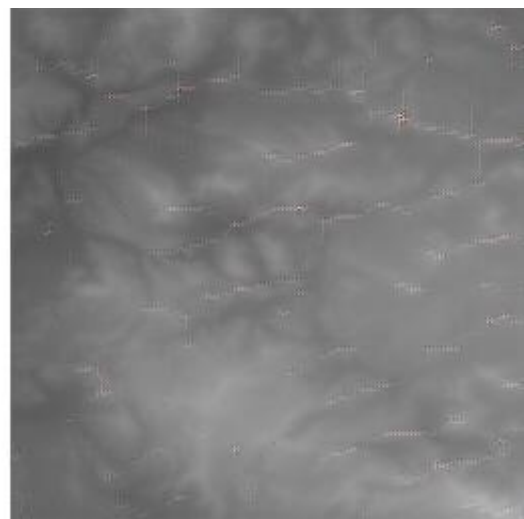
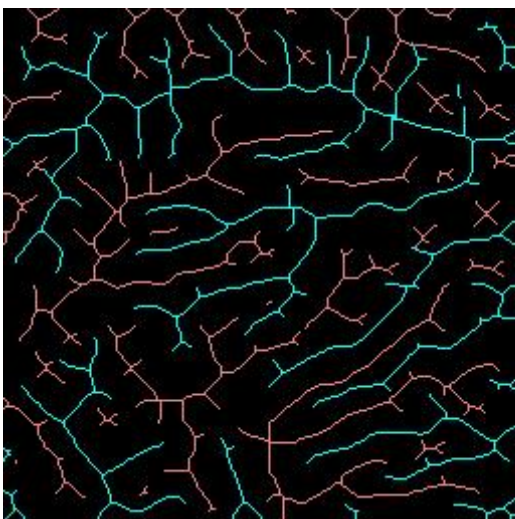


Figure A.84 - Input/Output testing pair. Right side is the input and left side is the output.

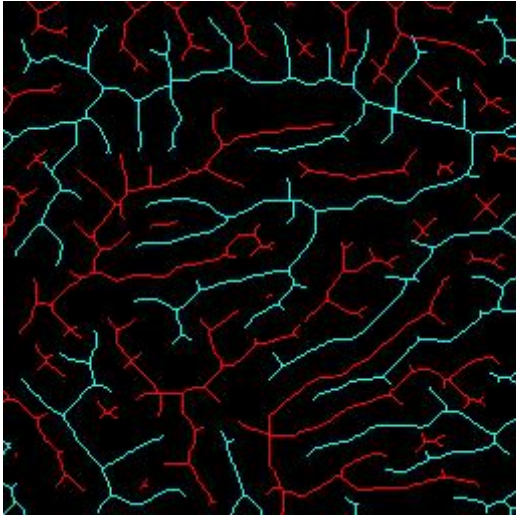


Figure A.85 - Input/Output testing pair. Right side is the input and left side is the output.

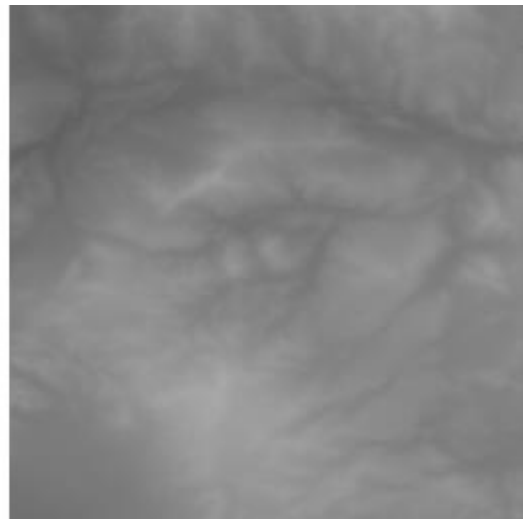
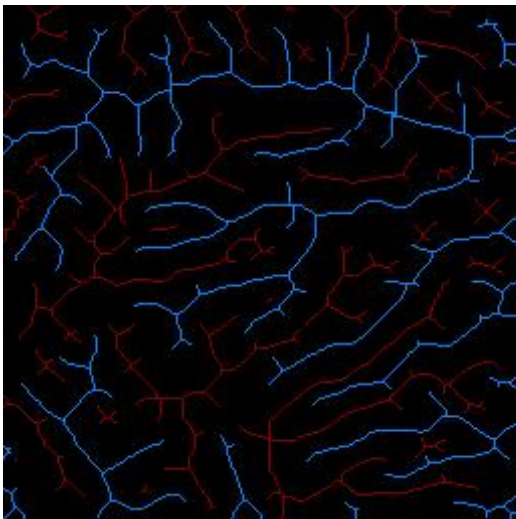


Figure A.86 - Input/Output testing pair. Right side is the input and left side is the output.

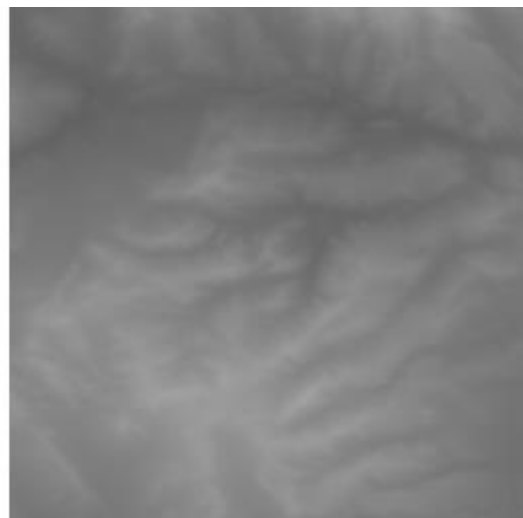
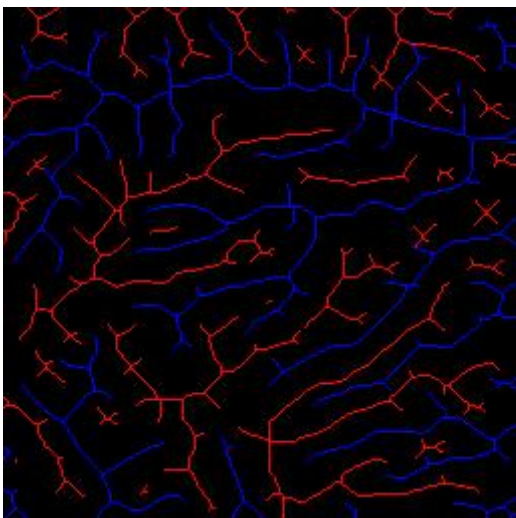


Figure A.87 - Input/Output testing pair. Right side is the input and left side is the output.

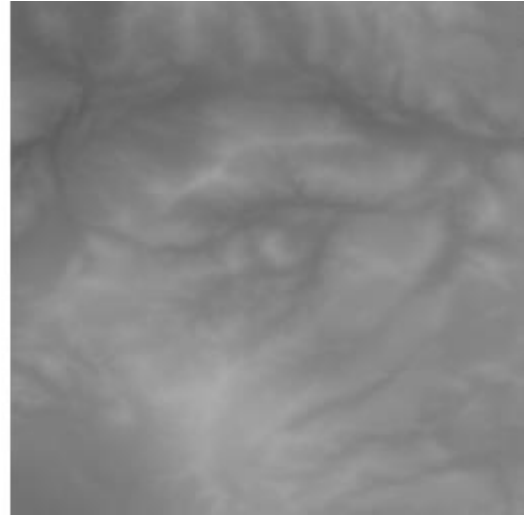
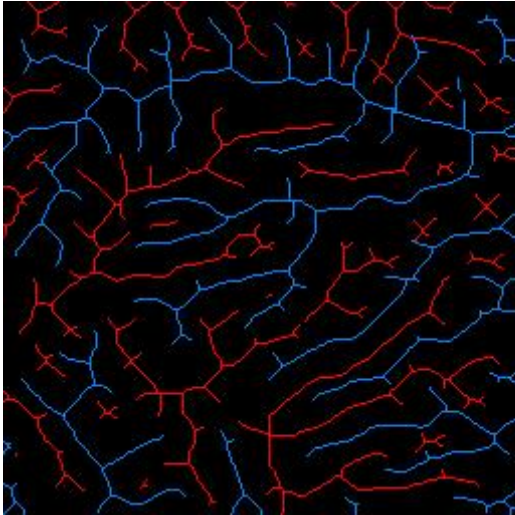


Figure A.88 - Input/Output testing pair. Right side is the input and left side is the output.

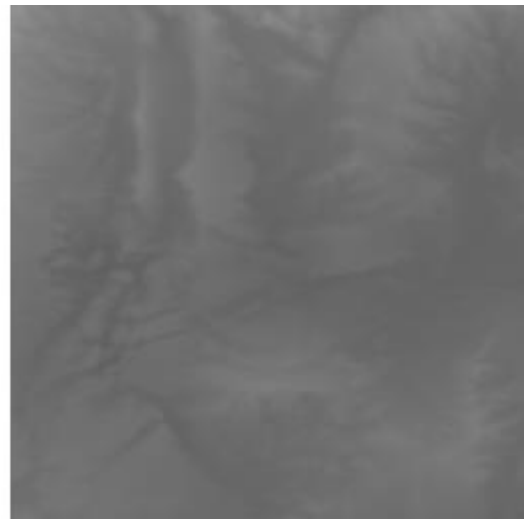
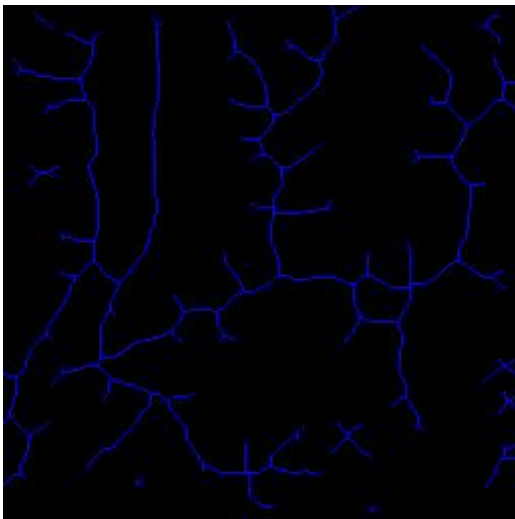


Figure A.89 - Input/Output testing pair. Right side is the input and left side is the output.

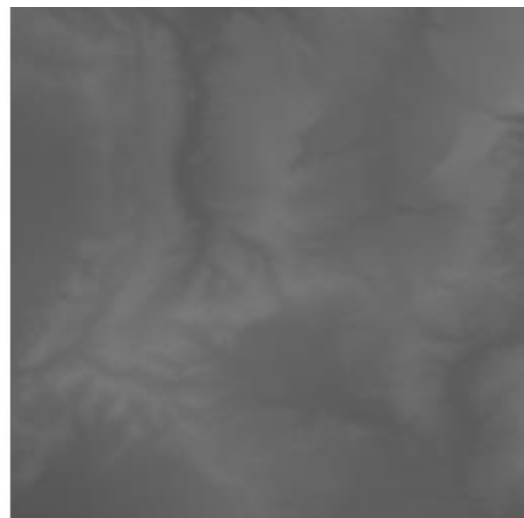
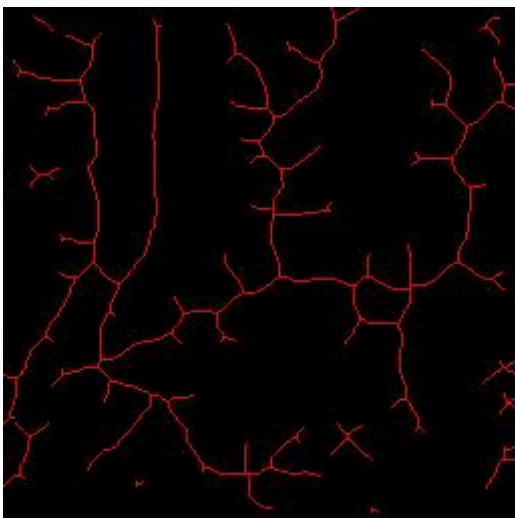


Figure A.90 - Input/Output testing pair. Right side is the input and left side is the output.

Annex B Terrain Comparison Test Pairs

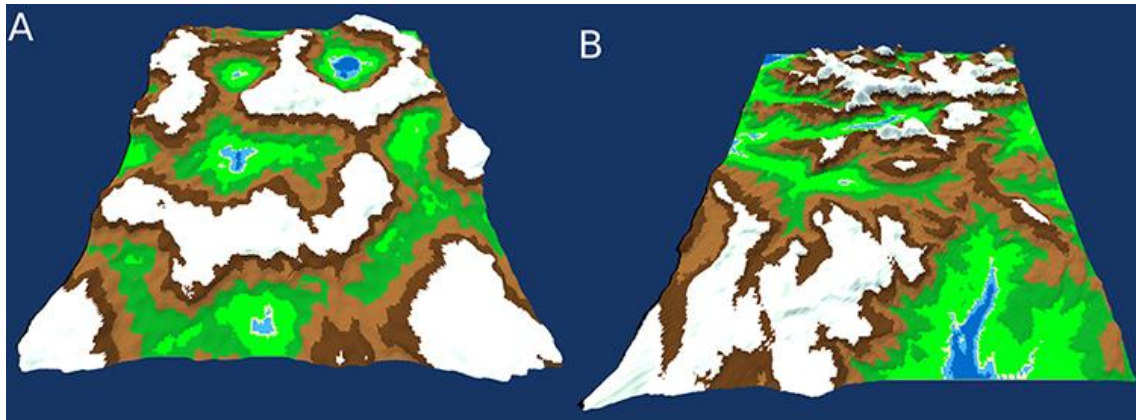


Figure B.1 - Terrain Comparison Pair

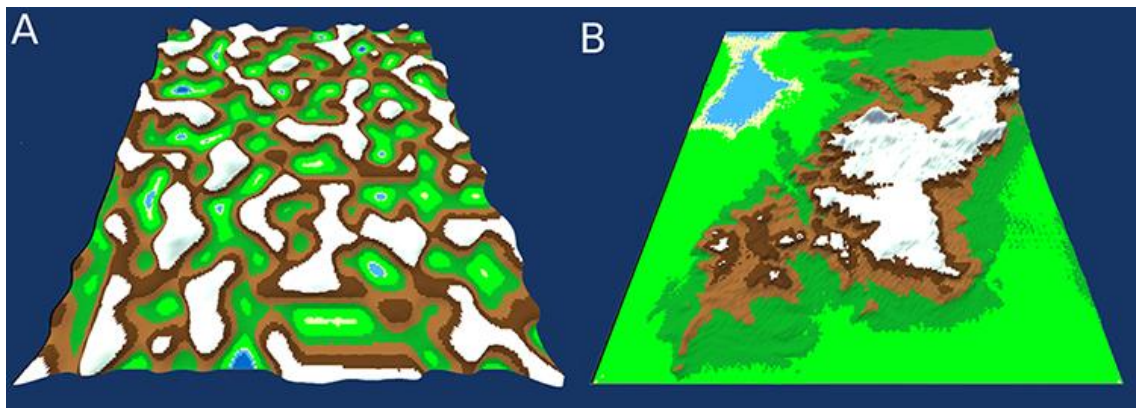


Figure B.2 - Terrain Comparison Pair

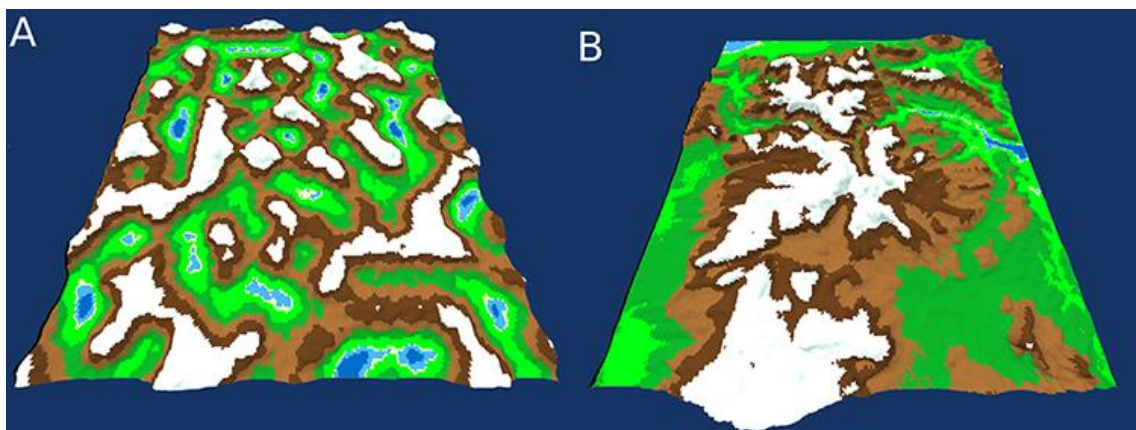


Figure B.3 - Terrain Comparison Pair

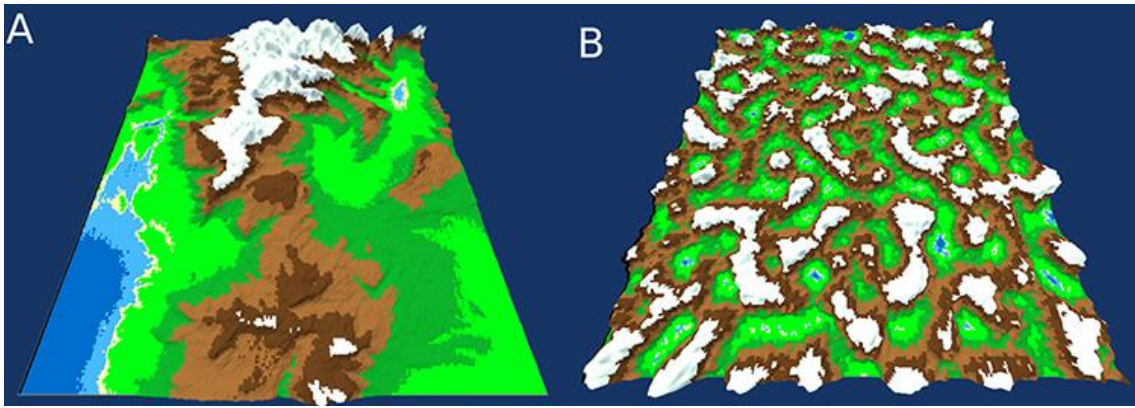


Figure B.4 - Terrain Comparison Pair

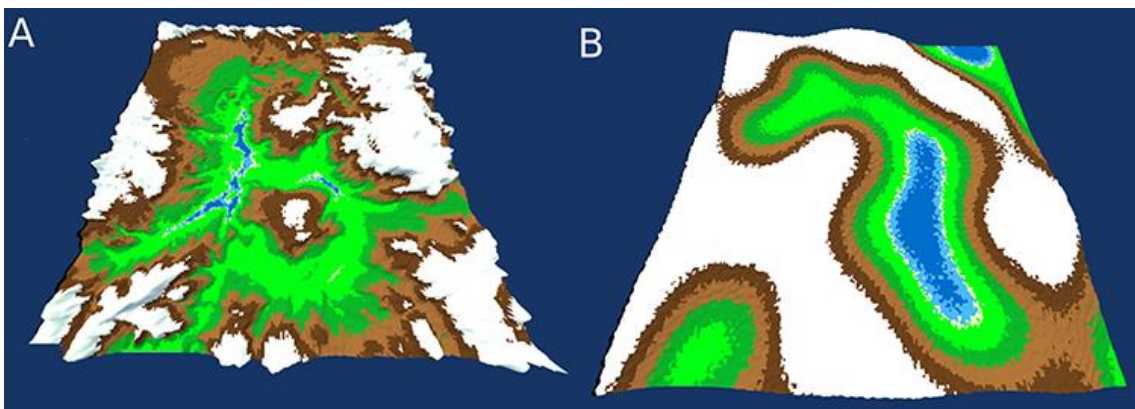


Figure B.5 - Terrain Comparison Pair

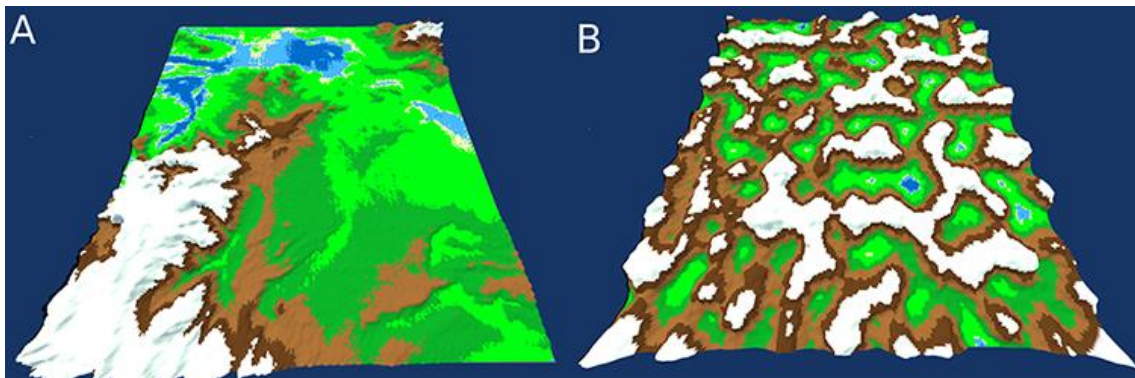


Figure B.6 - Terrain Comparison Pair

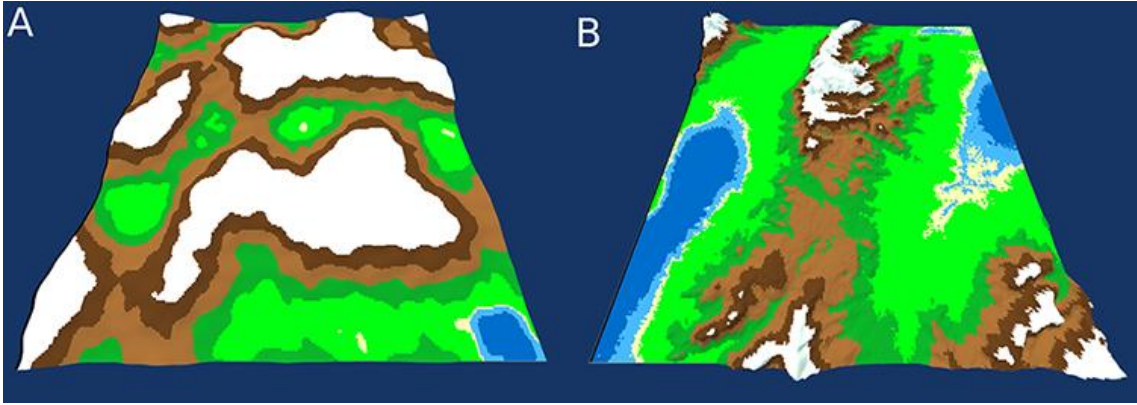


Figure B.7 - Terrain Comparison Pair

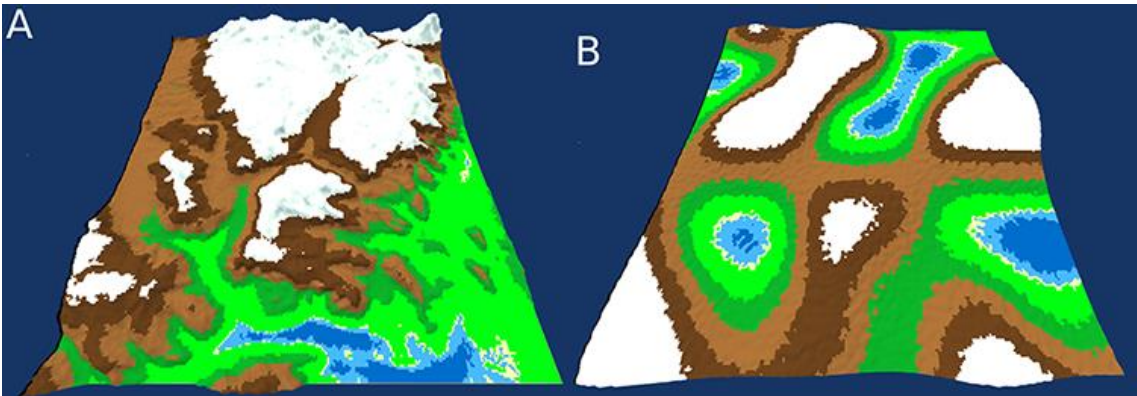


Figure B.8 - Terrain Comparison Pair

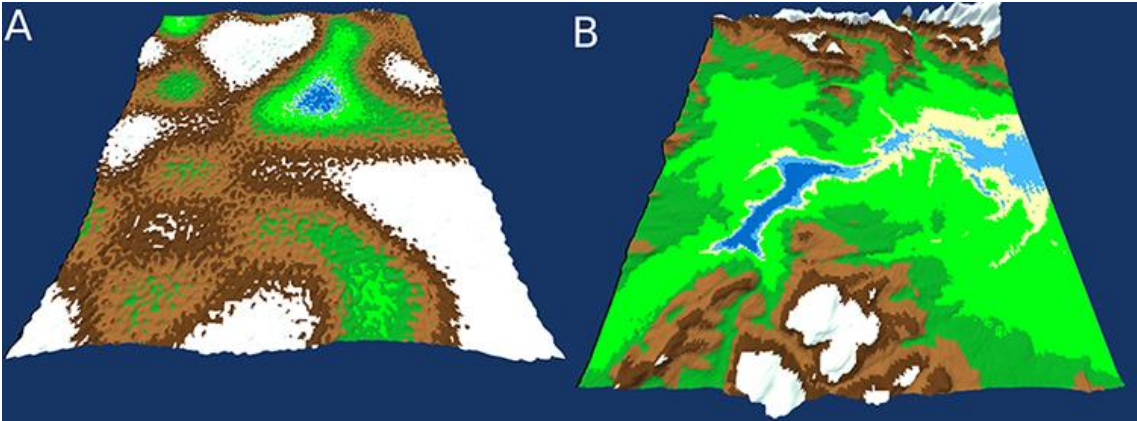


Figure B.9 - Terrain Comparison Pair

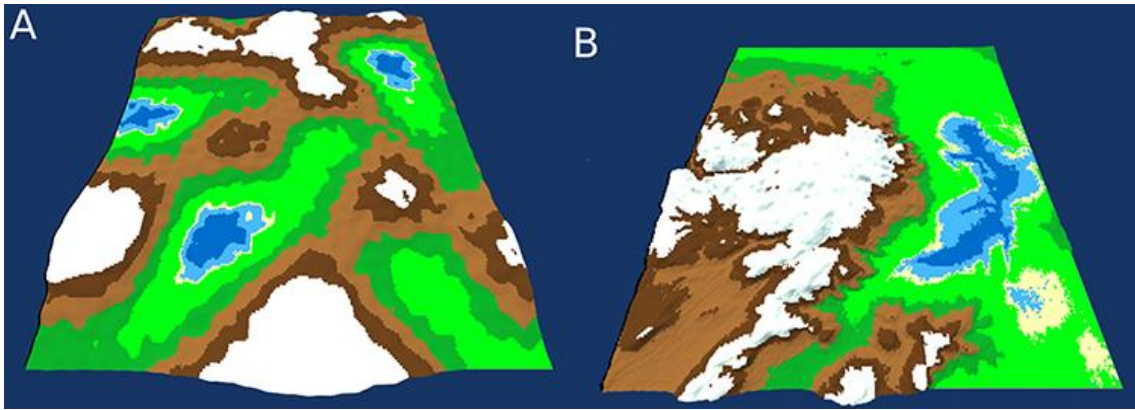


Figure B.10 - Terrain Comparison Pair

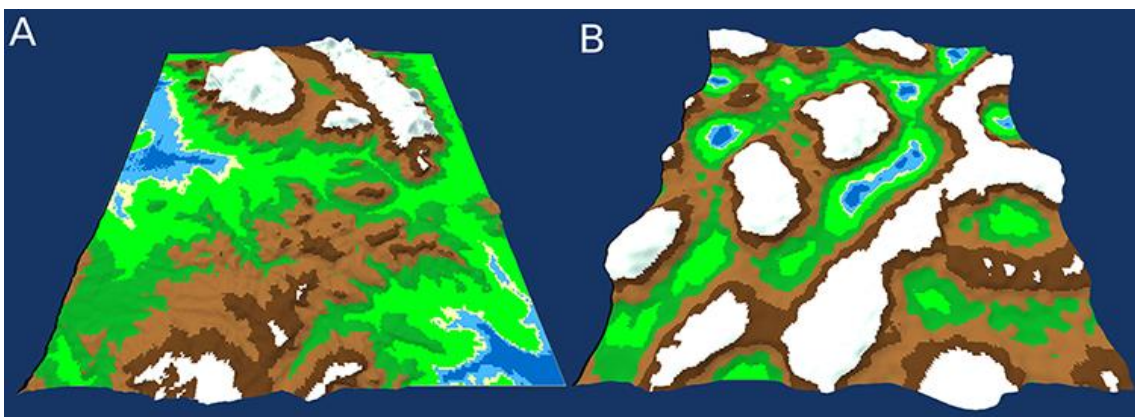


Figure B.11 - Terrain Comparison Pair

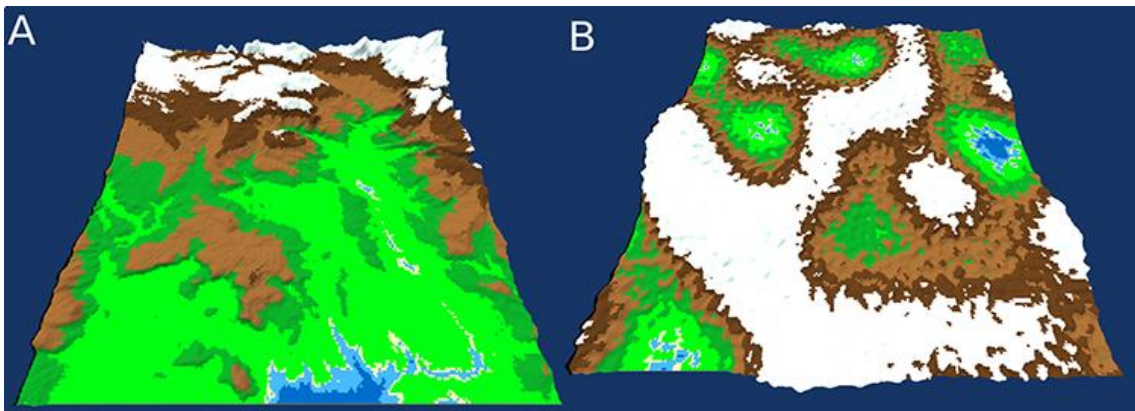


Figure B.12 - Terrain Comparison Pair

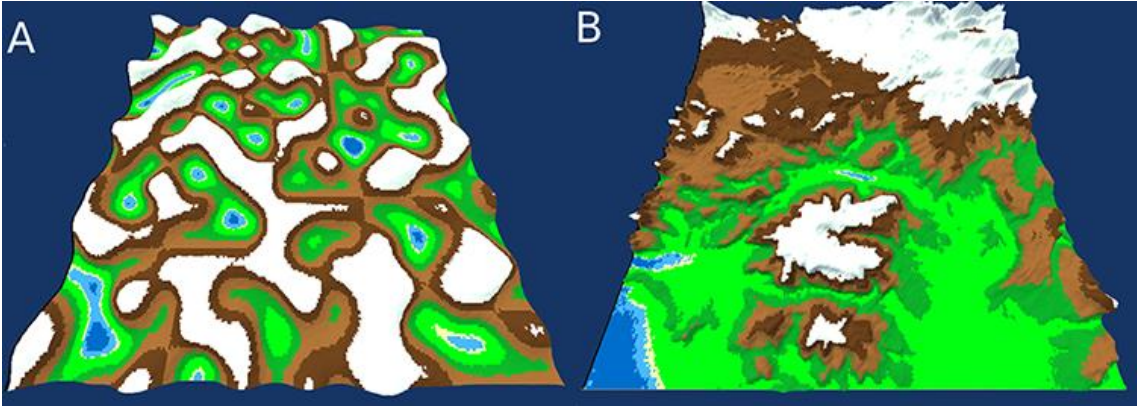


Figure B.13 - Terrain Comparison Pair

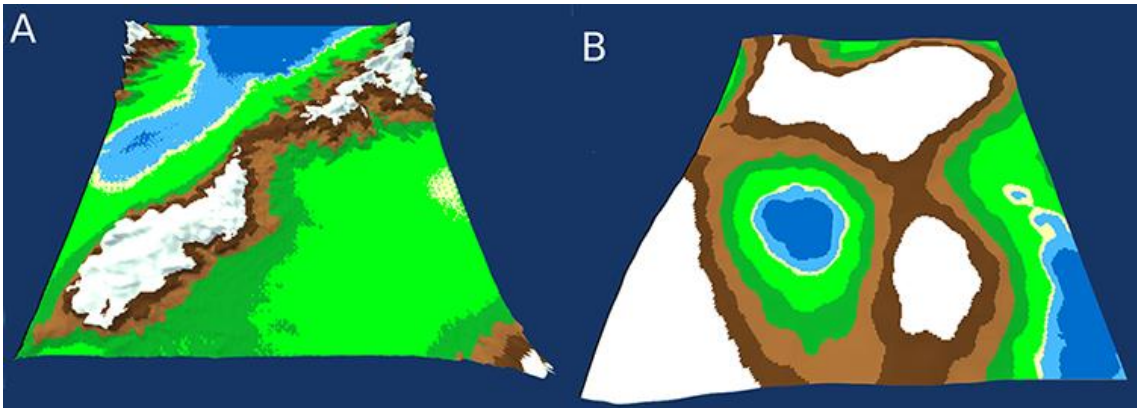


Figure B.14 - Terrain Comparison Pair

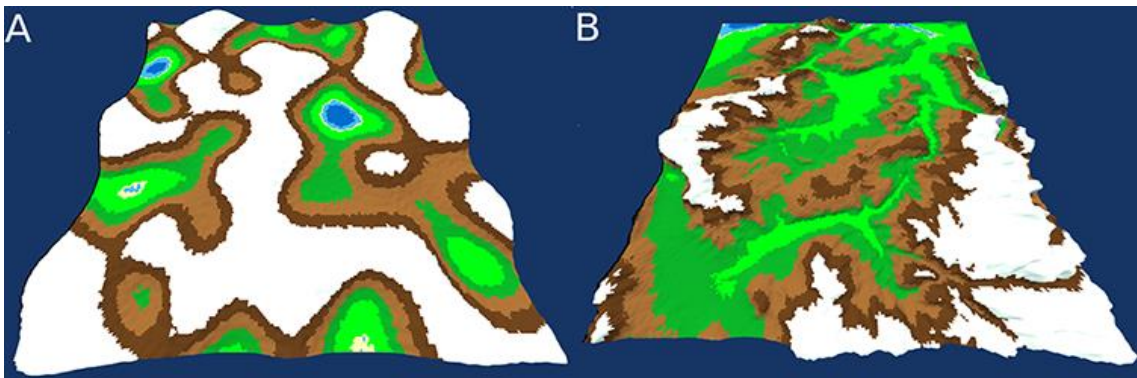


Figure B.15 - Terrain Comparison Pair

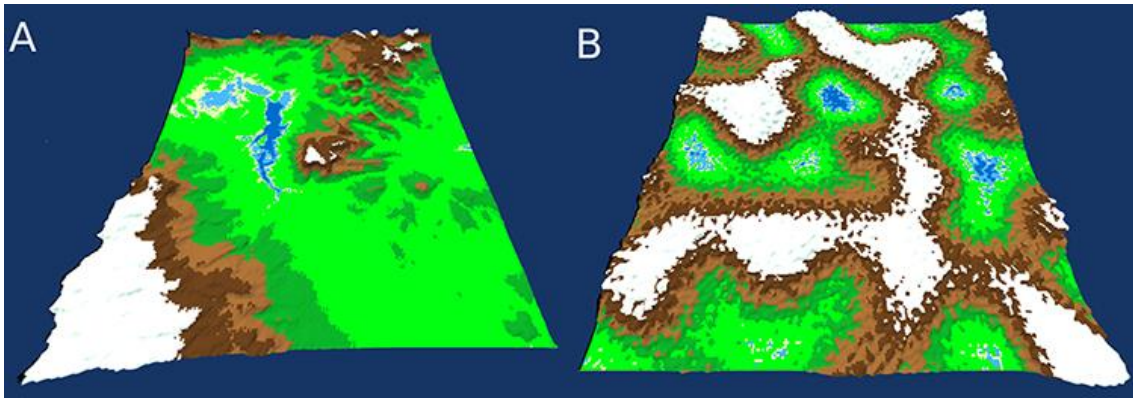


Figure B.16 - Terrain Comparison Pair

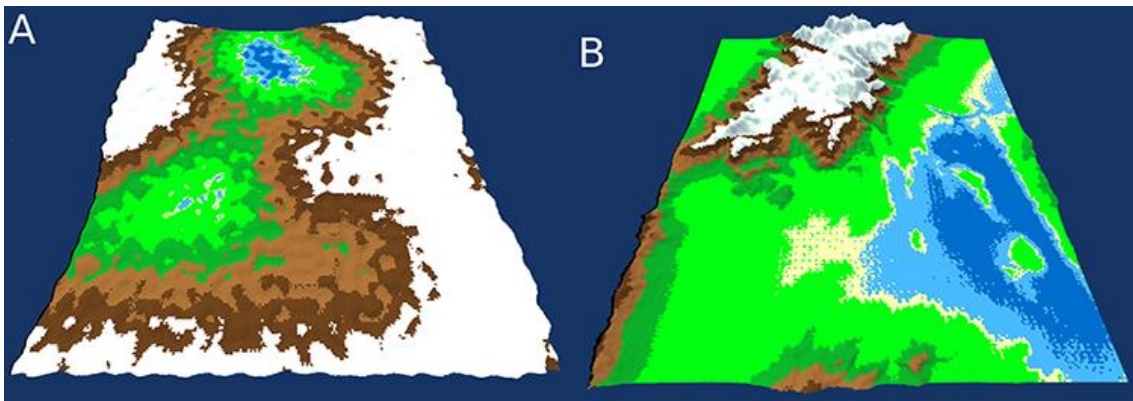


Figure B.17 - Terrain Comparison Pair

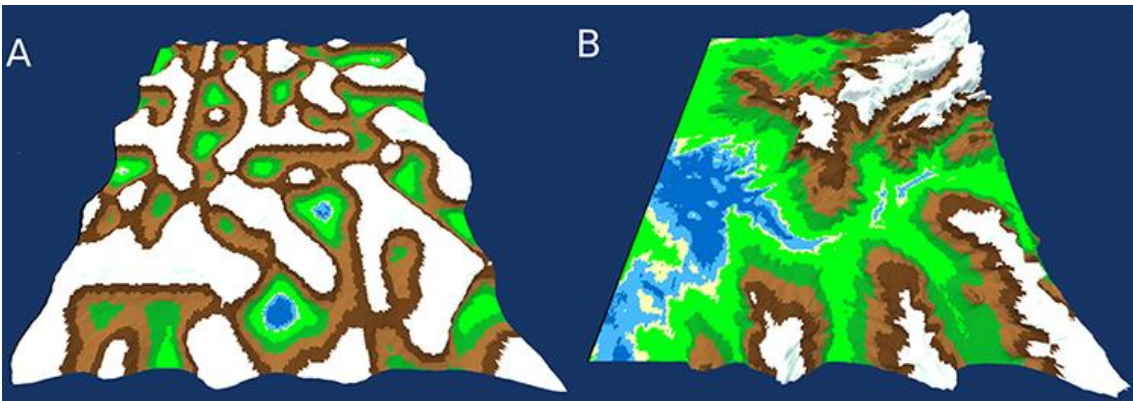


Figure B.18 - Terrain Comparison Pair

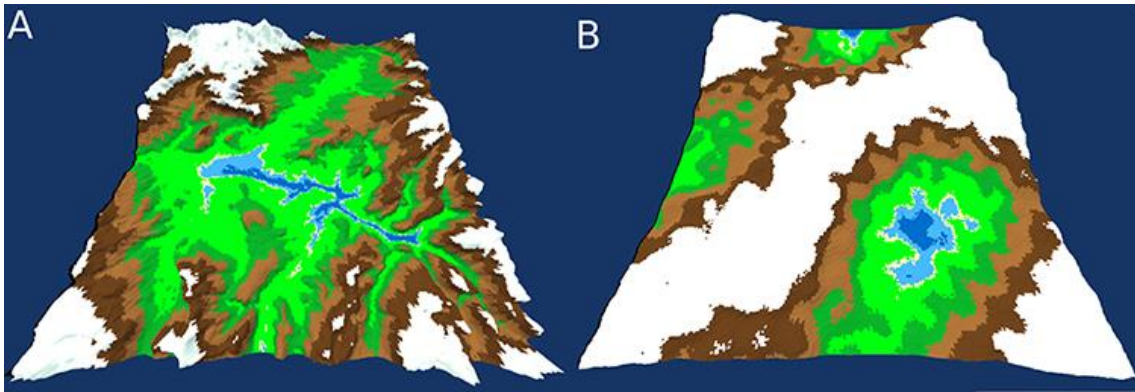


Figure B.19 - Terrain Comparison Pair

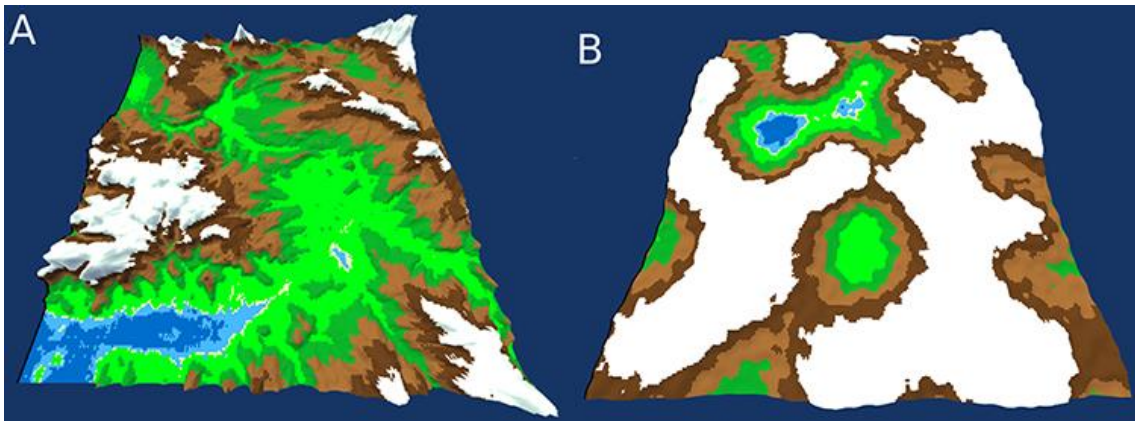


Figure B.20 - Terrain Comparison Pair

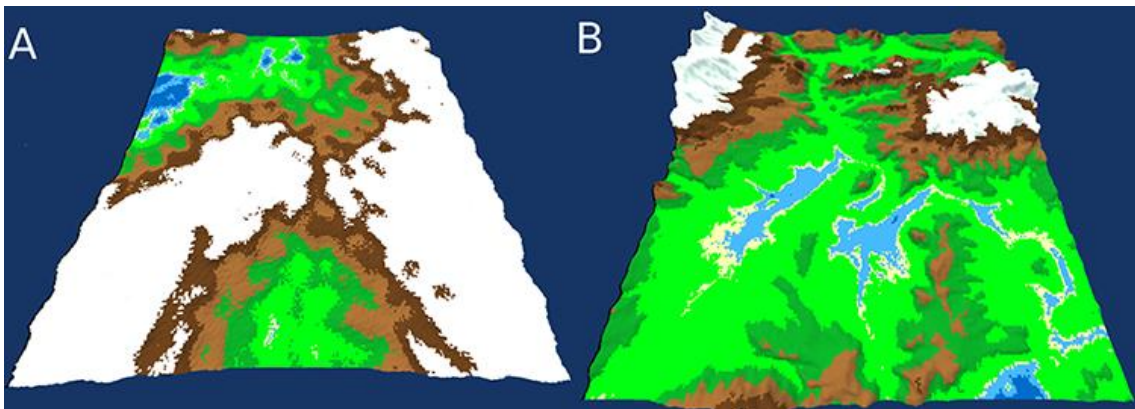


Figure B.21 - Terrain Comparison Pair

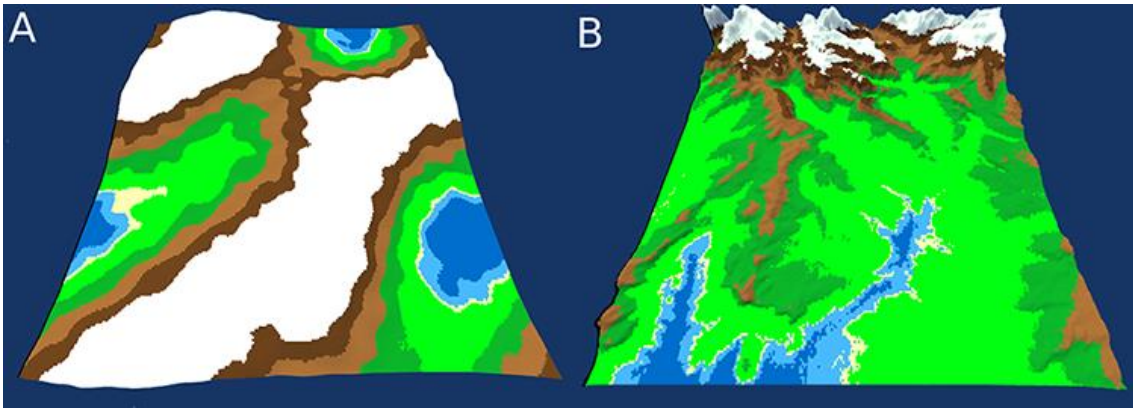


Figure B.22 - Terrain Comparison Pair

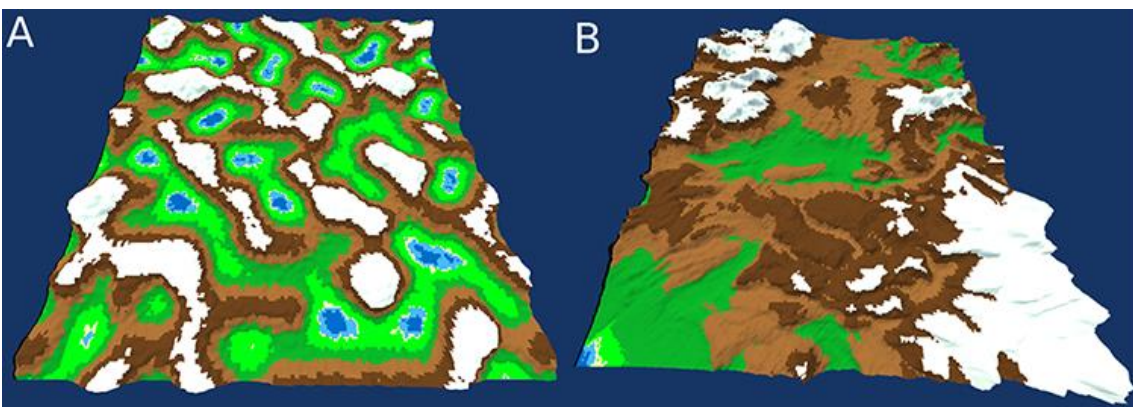


Figure B.23 - Terrain Comparison Pair

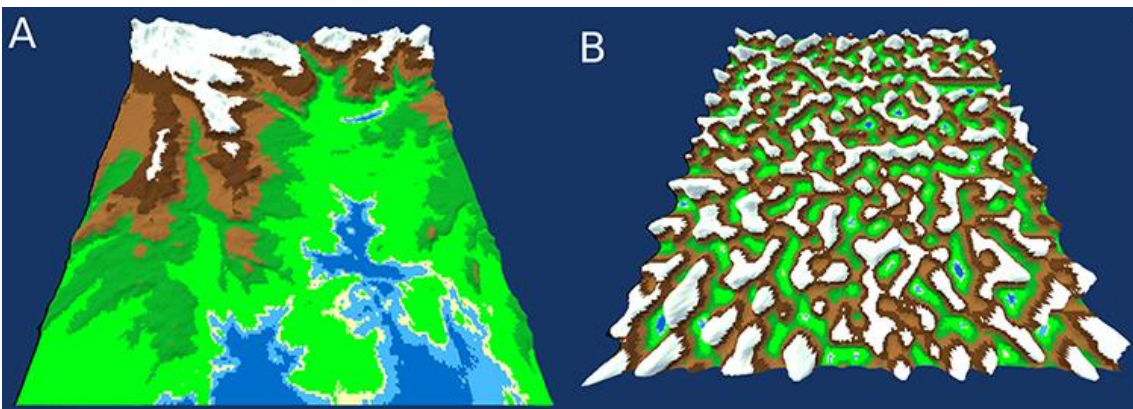


Figure B.24 - Terrain Comparison Pair

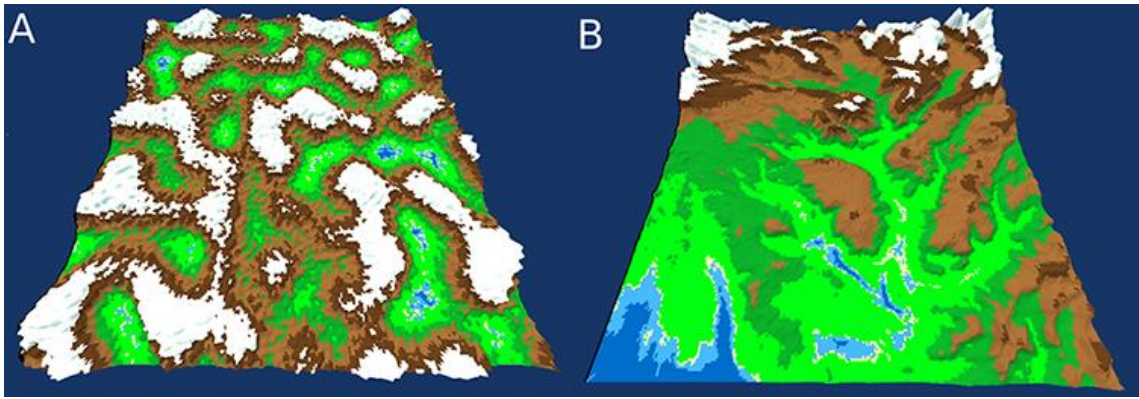


Figure B.25 - Terrain Comparison Pair

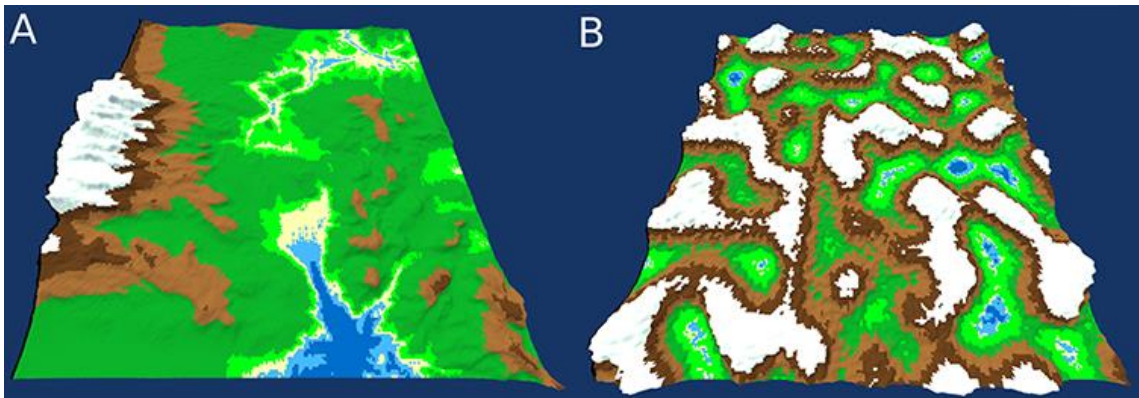


Figure B.26 - Terrain Comparison Pair

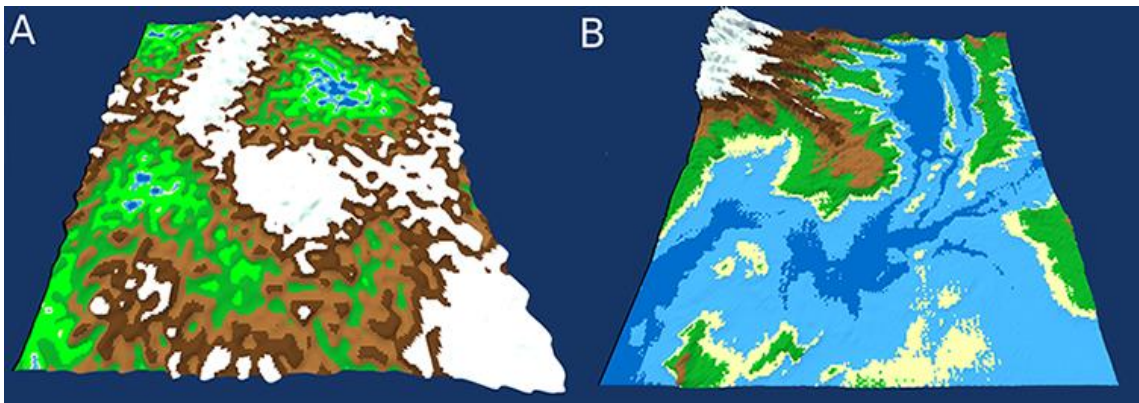


Figure B.27 - Terrain Comparison Pair

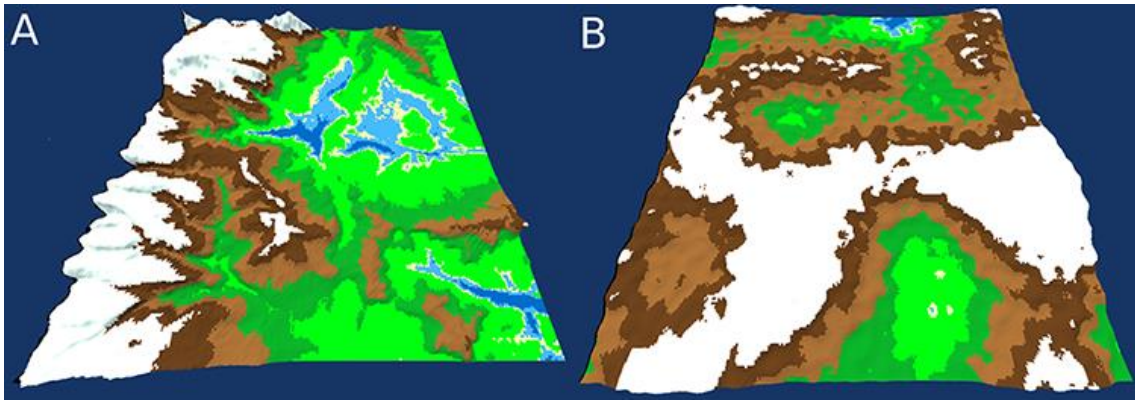


Figure B.28 - Terrain Comparison Pair

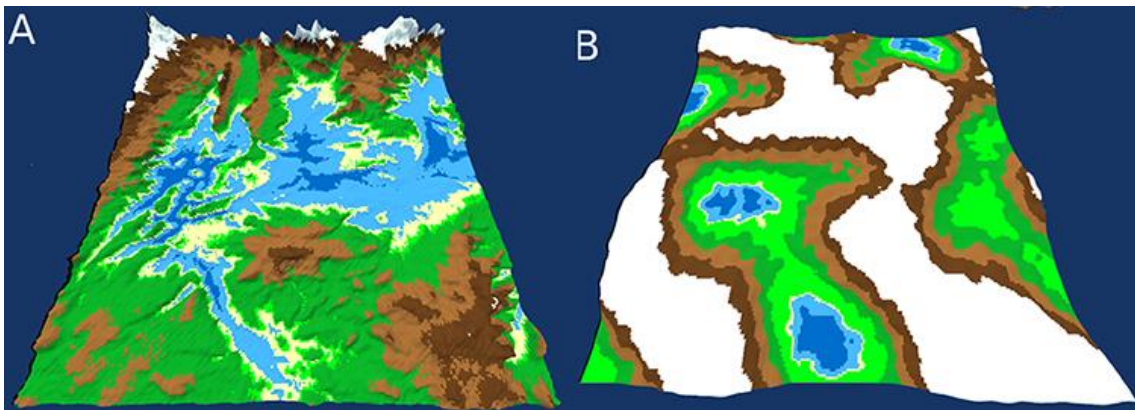


Figure B.29 - Terrain Comparison Pair

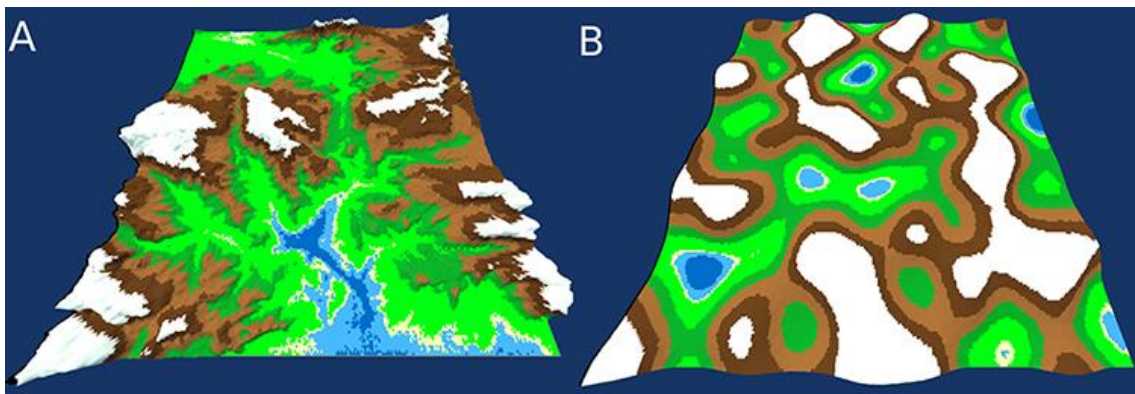


Figure B.30 - Terrain Comparison Pair

Annex C Usability Test Guide



Instituto Universitário de Lisboa

Departamento de Ciências e Tecnologias da Informação

Master's in computer engineering

Second App Intuitiveness Test

Rodrigo de Matos Pires Tavares de Almeida

Supervisor

PhD. Pedro Figueiredo Santana

ISCTE-IUL

October 2019

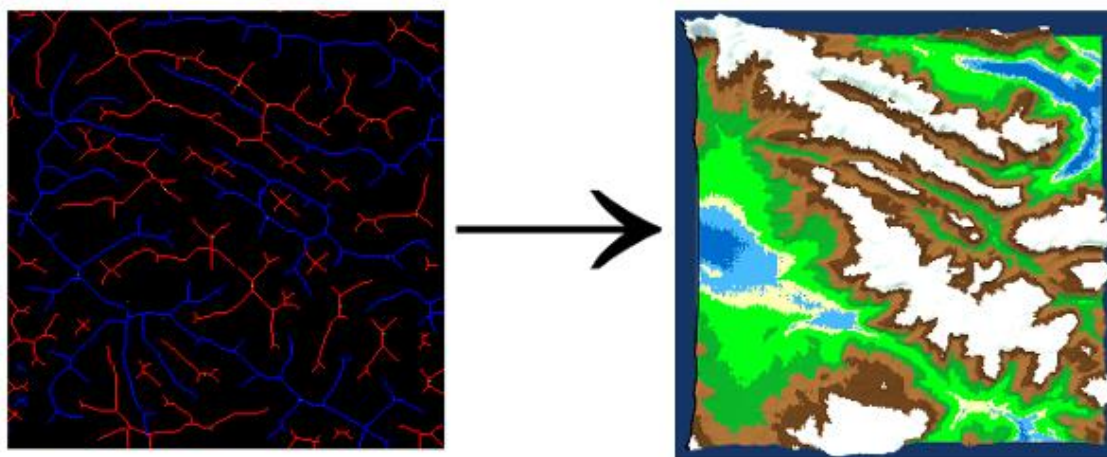
Fase 0

Introdução à aplicação.

Nesta fase inicial o utilizador recebe uma breve explicação sobre o objetivo e utilização da aplicação.

“Esta aplicação tem como objectivo permitir ao utilizador gerar terrenos 3D de forma automática. Ao contrário de outros métodos alternativos, este permite que o utilizador influencie o formato dos terrenos gerados pelo algoritmo através de linhas e pontos com cores específicas desenhados pelo próprio.”

mostrar ao utilizador a imagem presente em baixo



“Este é o género de input que é possível fazer e à direita é o tipo de resultados que se pode esperar deste método. A área de desenho vai ser do teu lado direito e o terreno vai ser gerado do lado esquerdo.”

Fase 1

Ações Atómicas.

Nesta fase o objectivo é entender qual o grau de compreensão que o utilizador tem da interface e das suas acções atómicas.

Para isso o utilizador é colocado num cenário que emula uma sessão real e são simuladas acções para testar a intuição da interface.

“Nesta primeira fase vou apenas testar a intuição da interface. Para isso vamos imaginar que te encontras a meio de uma sessão real e que as perguntas que te faço são acções que tu queres fazer na aplicação”

1. Botões de Desenho

- a) “Imagina que queres desenhar uma cordilheira de montanhas, em que botão carregarias?”.
- b) “Agora imagina que querias desenhar uma depressão no teu terreno, que botão escolherias?”.
- c) “E para desenhar uma elevação singular?”.
- d) “Imagina que te enganaste e queres apagar parte do que fizeste, em que botão escolherias?”.

2. Botões de Geração

- a) “Imagina agora que te sentes satisfeito com o desenho que fizeste e queres que o que esta na área de desenho seja utilizado para gerar o teu terreno, onde carregarias?”
- b) “Agora gostarias de ver na área de visualização o teu terreno, em que botão carregarias?”

- c) “Agora que já tens o teu terreno no ecrã, gostarias de atualizar o teu histórico de imagens de maneira a que o desenho que criaste anteriormente fique no histórico para futuramente o poderes reutilizar, onde carregarias?”
- d) “Imagina agora que estavas a desenhar o teu próximo terreno, mas a meio decidiste que não era nada disto que querias e decides limpar por completo a tua área de desenho, onde carregarias?”
- e) “Finalmente desenhaste algo do teu agrado e achas que o terreno que tens devia ser guardado. Em que botão vais carregar para exportar o teu terreno para fora da aplicação?”

3. Compreensão e Navegação do Log

- a) “Imagina agora que iniciaste uma nova sessão e decides que gostarias de alterar o terreno que te mostrei no início do teste para criar um terreno ligeiramente diferente. Que ações tomarias?”

4. Controlos de Visualização

- a) “Apercebeste-te de um pormenor no teu terreno e decides fazer zoom nessa parte do terreno. Como o farias?”
- b) “Agora queres ver a mesma zona, mas de outra perspetiva, que farias?”

certificar que o utilizador utiliza todos os controlos de navegação

- c) “Finalmente queres voltar a ver o terreno visto de cima, em que tecla pressionavas?”

Fase 2

Leitura do Tutorial.

Nesta fase os utilizadores irão ler na totalidade o tutorial presente na aplicação de forma a ficarem a conhecer na totalidade todas as funcionalidades da aplicação.

Fase 3

Interacção não supervisionada.

O utilizador é informado que depois da leitura na integra do tutorial tem 5-10 minutos de interacção não supervisionada com a aplicação para proporcionar uma familiarização o mais orgânica possível.

Fase 4

Interacção de Alto Nível.

Nesta fase final o utilizador será desafiado a fazer acções completas que tentam emular o fluxo de tarefas que se seriam realizadas caso a sessão fosse real.

“Nesta fase do teste vou descrever um resultado final, ou seja, um terreno, e deverás, por qualquer método que aches apropriado, reproduzir esse resultado final, utilizando qualquer funcionalidade, cor ou método.

Deverás seguir todo o fluxo descrito no tutorial desde o desenho até à actualização do histórico.”

Durante esta fase, irei recolher informação referente ao uso por parte do utilizador das funcionalidades implementadas: Uso ou não das cores extra, uso ou não da borracha, do Log e da funcionalidade de copiar e alterar inputs previamente guardados.

1. “Desenha uma cordilheira de montanhas que atravessa o terreno diagonalmente da esquerda para a direita.”
2. “Desenha um vale com montanhas que o circundam.”
3. “Desenha uma elevação singular no meio do terreno, sem mais nada a volta”
4. “Coloca a câmara de modo a ficar a olhar de perto para o pico mais alto do terreno que esta no ecrã.”
5. “Decidiste que queres adicionar algo mais ao terreno que criaste ainda agora.”
6. “Depois de adicionados os detalhes desejados, achas que o teu terreno poderia ser mais alto e decides aumentar a sua altura.”
7. “Finalmente achas que o resultado está do teu agrado e decides exportar o terreno para fora da aplicação.”

Annex D Second Usability Test Guide



Instituto Universitário de Lisboa

Departamento de Ciências e Tecnologias da Informação

Masters in Computer Engineering

Second App Intuitiveness Test

Rodrigo de Matos Pires Tavares de Almeida

Supervisor

PhD. Pedro Figueiredo Santana

ISCTE-IUL

October 2019

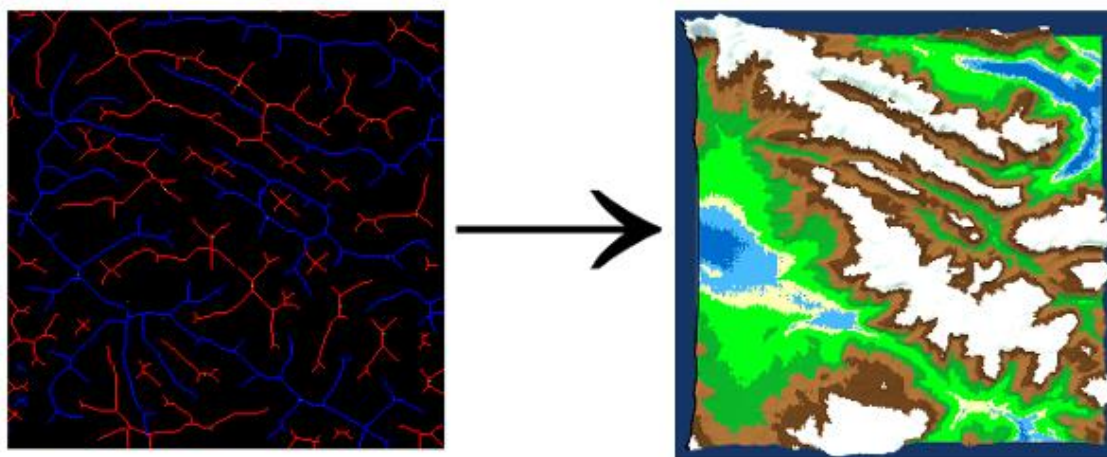
Fase 0

Introdução à aplicação.

Nesta fase inicial o utilizador recebe uma breve explicação sobre o objetivo e utilização da aplicação.

“Esta aplicação tem como objectivo permitir ao utilizador gerar terrenos 3D de forma automática. Ao contrário de outros métodos alternativos, este permite que o utilizador influencie o formato dos terrenos gerados pelo algoritmo através de linhas e pontos com cores específicas desenhados pelo próprio.”

mostrar ao utilizador a imagem presente em baixo



“Este é o género de input que é possível fazer e à direita é o tipo de resultados que se pode esperar deste método. A área de desenho vai ser do teu lado direito e o terreno vai ser gerado do lado esquerdo.”

Fase 1

Ações Atómicas.

Nesta fase o objectivo é entender qual o grau de compreensão que o utilizador tem da interface e das suas acções atómicas.

Para isso o utilizador é colocado num cenário que emula uma sessão real e são simuladas acções para testar a intuição da interface.

“Nesta primeira fase vou apenas testar a intuição da interface. Para isso vamos imaginar que te encontras a meio de uma sessão real e que as perguntas que te faço são acções que tu queres fazer na aplicação”

1. Botões de Desenho

- a) “Imagina que queres desenhar uma cordilheira de montanhas, em que botão carregarias?”.
- b) “Agora imagina que querias desenhar uma depressão no teu terreno, que botão escolherias?”.
- c) “E para desenhar uma elevação singular?”.
- d) “Imagina que te enganaste e queres apagar parte do que fizeste, em que botão escolherias?”.

2. Botões de Geração

- a) “Imagina agora que te sentes satisfeito com o desenho que fizeste e queres que o que esta na área de desenho seja utilizado para gerar o teu terreno, onde carregarias?”
- b) “Imagina agora que estavas a desenhar o teu próximo terreno, mas a meio decidiste que não era nada disto que querias e decides limpar por completo a tua área de desenho, onde carregarias?”

- c) “Finalmente desenhaste algo do teu agrado e achas que o terreno que tens devia ser guardado. Em que botão vais carregar para exportar o teu terreno para fora da aplicação?”

3. Compreensão e Navegação do Log

- a) “Imagina agora que iniciaste uma nova sessão e decides que gostarias de alterar o terreno que te mostrei no início do teste para criar um terreno ligeiramente diferente. Que ações tomarias?”

4. Controlos de Visualização

- a) “Apercebeste-te de um pormenor no teu terreno e decides fazer zoom nessa parte do terreno. Como o farias?”
- b) “Agora queres ver a mesma zona, mas de outra perspetiva, que farias?”

certificar que o utilizador utiliza todos os controlos de navegação

- c) “Finalmente queres voltar a ver o terreno visto de cima, em que tecla pressionavas?”