E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

SandDiffusion: Image Generation with Depth Conditioning for Augmented Reality SandBoxes

Grado Universitario en
Ingeniería Informática

Trabajo Fin de Grado

Igor Volodimir Vons Vons

Asier Ruperto Marzo Pérez

Pamplona, 10/01/2024

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Table of Contents

1. Introduction

1.1. Objectives

This TFG emerges from a collaboration grant signed with UPNA during the academic exercise of 2022-2023, and which main focus was diving into the new AI technologies that started surging at the beginning of set academic year. Its main objective is exploring a real case implementation into already existing technologies, allowing for a newer, richer, and more expressive experience, enhancing its usage without needing to resort to complex algorithms which may be both extremely difficult to produce and maintain, and whose results are not at the level of the ones achieved with AI.

1.2. State of the Art

Recent developments in the field of artificial intelligence (AI) have seen widespread attention and expansion. Discussing AI in its entirety is challenging due to the diverse range of AI varieties. While image diffusers and transformer-based generative AIs have recently dominated the spotlight, other types, particularly those making significant contributions to specific fields, have often been overlooked.

Noteworthy examples include AlphaFold[1], a revolutionary model in predicting the 3D structures of proteins, and AlphaTensor[2], which enhances the efficiency of specific algorithms. Despite their impressive achievements in their respective domains, these models have garnered less attention from the general public due to their high specificity and technical nature.

The landscape began to shift with the introduction of DALL·E 2[3], an upgraded version of the previous DALL·E model. This upgrade caught the attention of a broader audience as it showcased the ability to generate images seemingly out of thin air. However, access to this technology was initially restricted. The momentum further increased with the release of Midjourney on July 12, 2022, offering more accessible results, and challenging the perceived impressiveness of previous models.

During the development of this project advancements were made, with the release of DALL·E 3, Midjourney v6, and SDXL amongst them.

A pivotal moment in the growing interest was the release of Stable Diffusion, a free-source model that allowed users greater control over content generation without reliance on third-party access. This democratization of access led to optimizations and adaptations, enabling the running of AIs on less powerful hardware, and empowering more individuals to experiment with and modify the technology according to their needs.

Simultaneously, the rise of transformers, particularly generative pretrained transformers (GPTs), gained immense popularity. GitHub Copilot, a technology aiding programmers by suggesting code completions, was an early example, followed by the widespread recognition of GPT-3 and its chat variation. Despite the public's fascination and even speculation about approaching sentience, it is crucial to delve into the technology behind these models to understand their limitations.

During this evolving landscape, multimodality has emerged as a notable technology capable of combining image and text generation, promising further advancements.

Shifting focus to Stable Diffusion, its history traces back to August 22, 2022, with its public release. While the model itself is recent, its development spanned several years, with predecessors like DALL-E and DALL-E 2. Stable Diffusion employs a diffusion technique, generating images based on text prompts describing the desired content. The model's open-source nature distinguishes it from similar models, fostering rapid development of tools and add-ons for precise control over output, marking the onset of heightened interest in the AI industry.

1.3. On Stable Diffusion

The history of Stable Diffusion starts in August, the 22nd of 2022, with its public release by Stability Ai[4].

Stable Diffusion is the name of an open-source AI model based on a diffusion technique to generate images out of text prompts, which would describe the contents desired. Although the model itself is recent (its v1 was released in early autumn, and the latest iteration, v2 at the end of 2022) the technology already has been several years in development, with models like DALL-E and DALL-E 2 were released previously. SD together with Midjourney marked the start of the current boom in interest towards the AI industry (lately focused on GPTs), showing the world that these models were able to generate far more consistent results that what anyone could have expected, and their final consequences are still yet to be fully comprehended.

The idea behind diffusion-based AI models consists in training of autoencoders (AIs trained to encode and then decode the images into the original ones), conditioned with various additional steps, like diffusing the images with Gaussian noise (thus the “diffusion-based” part) or cross attention methods which allow retaining more info from the learning process. Specifically, the interest on SD was brought by the fact that, unlike other similar models, this one is open source, which allowed for a quick and intense development of many tools and add-ons used for a much more precise control over the output. These will be presented along the rest of the document.

1.3.1. Work and Architecture

In the simplest form, Stable Diffusion is a text-to-image model. Give it a text prompt. It will return an AI image matching the text.

Stable Diffusion belongs to a class of deep learning models called diffusion models. They are generative models, meaning they are designed to generate new data similar to what they have seen in training. In the case of Stable Diffusion, the data are images.

The “Diffusion” part comes from the math behind its functioning looking very much like diffusion in physics. The concept is the following:

Let’s suppose there is a diffusion model trained with only two kinds of images: cats and dogs. In the Figure 1, the two peaks on the left represent the groups of cat and dog images.

1.3.1.1. Forward diffusion

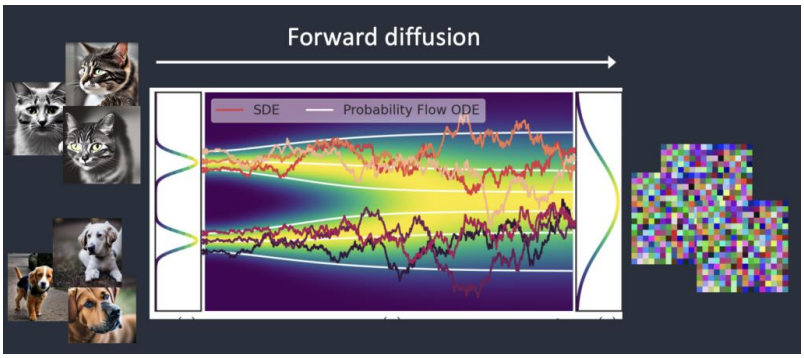

Figure 1 - Forward diffusion turns a photo into noise[5]

A forward diffusion process adds noise to a training image, gradually turning it into an uncharacteristic noise image. The forward process will turn any cat or dog image into a noise image. Eventually, it will be impossible to tell whether they were initially a dog or a cat.

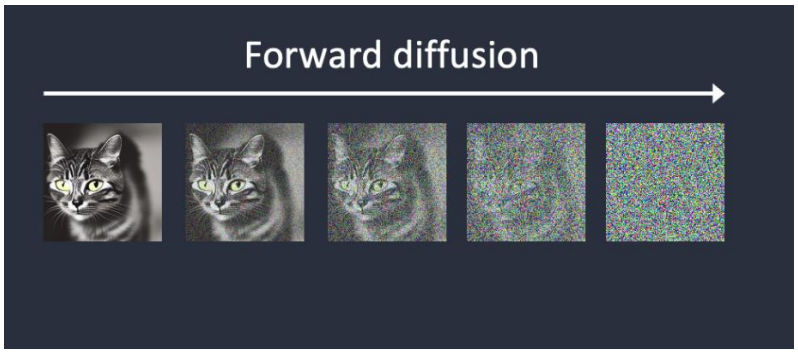

Figure 2 - Forward diffusion of a cat image.

1.3.1.2. Reverse diffusion

It aims at recovering the original image from the noise. Starting from a noisy, meaningless image, reverse diffusion recovers a cat or a dog image. This is the main idea.

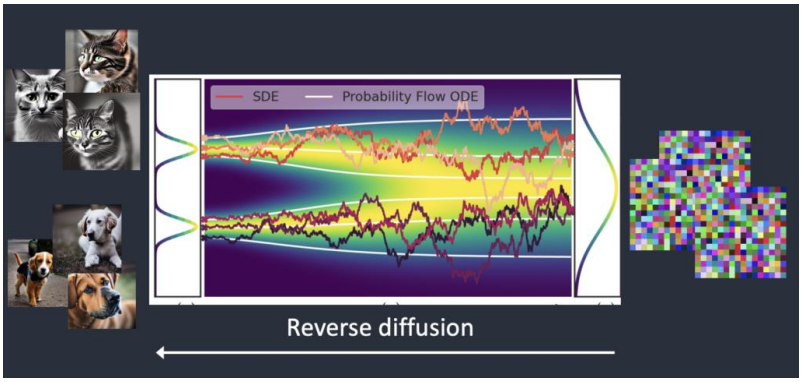

Figure 3 - The reverse diffusion process recovers an image.

Technically, every diffusion process has two parts: (1) drift and (2) random motion. The reverse diffusion drifts towards either cat or dog images but nothing in between. That's why the result can either be a cat or a dog.

1.3.1.3. Training

To reverse the diffusion, we need to know how much noise is added to an image. The answer is teaching a neural network model to predict the noise added. It is called the noise predictor in Stable Diffusion. It is a U-Net model[6]. The training goes as follows:

- Pick a training image, like a photo of a cat.
- Generate a random noise image.
- Corrupt the training image by adding this noisy image up to a certain number of steps.
- Teach the noise predictor to tell us how much noise was added. This is done by tuning its weights and showing it the correct answer.

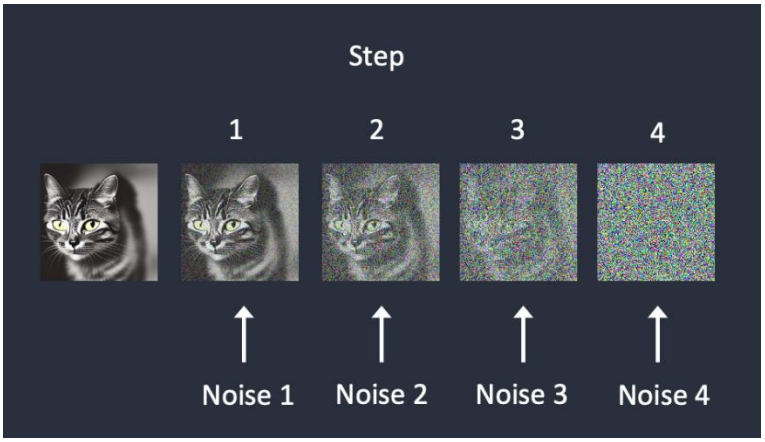

Figure 4 - Noise is sequentially added at each step. The noise predictor estimates the total noise added up to each step.

After training, we have a noise predictor capable of estimating the noise added to an image.

Once we have the noise predictor, we can use it by first generating a completely random image and ask the noise predictor to tell us the noise. We then subtract this estimated noise from the original image. Repeating this process a number of times we get an image of either a cat or a dog.

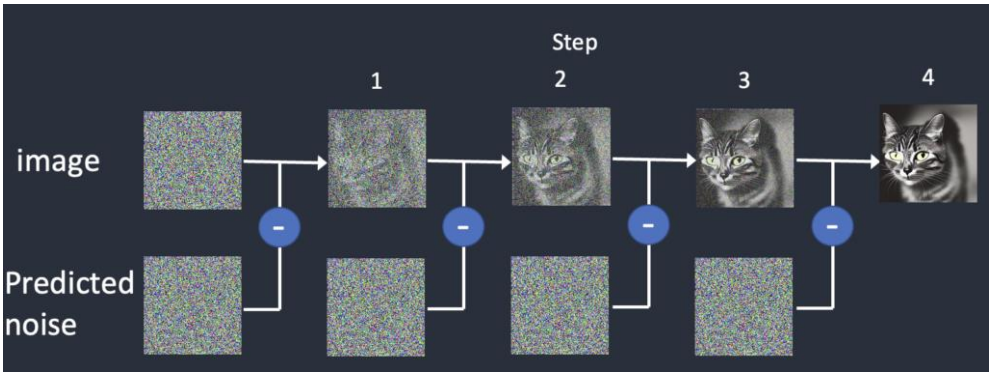

Figure 5 - Reverse diffusion works by subtracting the predicted noise from the image successively.

The way this noise to be subtracted is estimated depends on the sampler[7] one chooses, and it determines the speed and stability of the convergence to a result.

The image changes from noisy to clean progressively. The real reason is we try to get to an expected noise at each sampling step. This is called the noise schedule. We can see an example on Figure 6.

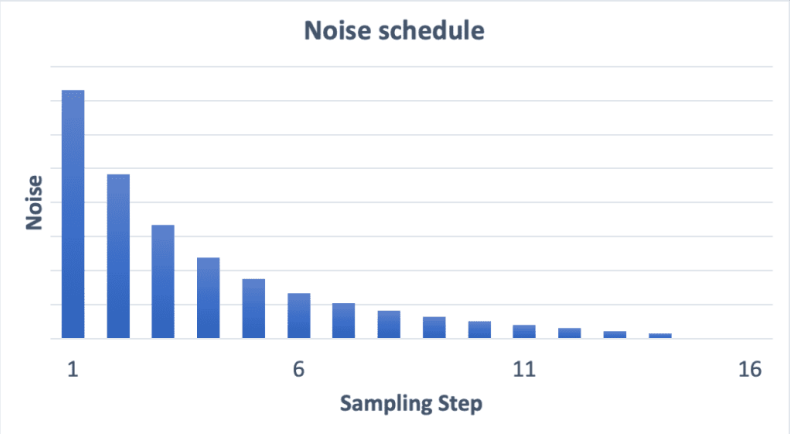

Figure 6 - A noise schedule for 15 sampling steps.

The noise schedule is something we define. We can choose to subtract the same amount of noise at each step. Or we can subtract more in the beginning, like above. The sampler subtracts just enough noise in each step to reach the expected noise in the next step. That's what you see in the step-by-step image.

1.3.1.4. Stable Diffusion model

It should be noted that the previous description is not how Stable Diffusion works. That diffusion process takes place in image space. It is computationally very, very slow. The image space is enormous. We have a 512×512px image with three colour channels (red, green, and blue), which makes for a 786,432-dimensional space.

Diffusion models like Google's Imagen[8][9] and Open AI's DALL-E are in pixel space. They have used some tricks to make the model faster but still not enough.

Stable Diffusion in contrast is a latent diffusion model. Instead of operating in the high-dimensional image space, it first compresses the image into the latent space. The latent space is 48 times smaller, so it reaps the benefit of crunching a lot fewer numbers. That's why it's a lot faster.

1.3.1.5. Variational Autoencoder

It is done using a technique called the variational autoencoder. The Variational Autoencoder (VAE) neural network has two parts: (1) an encoder and (2) a decoder. The encoder compresses an image to a lower dimensional representation in the latent space. The decoder restores the image from the latent space.

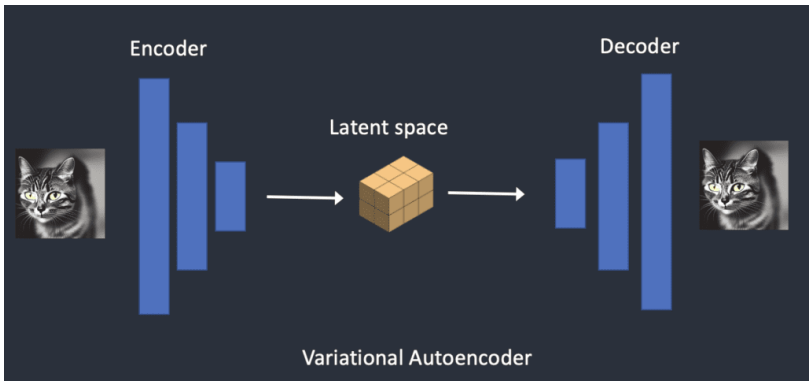

Figure 7 - Variational autoencoder transforms the image to and from the latent space.

The latent space of the Stable Diffusion model is 4x64x64, 48 times smaller than the image pixel space. All the forward and reverse diffusions named earlier are done ºin the latent space.

So, during training, instead of generating a noisy image, it generates a random tensor in latent space (latent noise). Instead of corrupting an image with noise, it corrupts the representation of the image in latent space with the latent noise. The reason for doing that is it is a lot faster since the latent space is smaller.

The image resolution is reflected in the size of the latent image tensor. The size of the latent image is 4x64x64 for 512×512 images only. It is 4x96x64 for a 768×512 portrait image. That's why it takes longer and more VRAM to generate a larger image.

The reason why the VAE can compress an image into a much smaller latent space without losing information is, unsurprisingly, natural images are not random. They have high regularity: A face follows a specific spatial relationship between the eyes, nose, cheek, and mouth. A dog has 4 legs and is a particular shape. This implies that high dimensionality of images is artefactual. Natural images can be readily compressed into the much smaller latent space without losing most information. This is called the manifold hypothesis[10] in machine learning.

Although the finer details are lost to this process we can recover them with an additional VAE decoder, responsible only for painting those.

Reverse diffusion in latent space

Here's how latent reverse diffusion in Stable Diffusion works.

1. A random latent space matrix is generated.
2. The noise predictor estimates the noise of the latent matrix.
3. The estimated noise is then subtracted from the latent matrix.
4. Steps 2 and 3 are repeated up to specific sampling steps.
5. The decoder of VAE converts the latent matrix to the final image.

1.3.2. Conditioning

Up to now, the process we described will only give you an image of a cat or a dog without any way to control it. This is where conditioning comes in. The purpose of conditioning is to steer the noise predictor so that the predicted noise will give us what we want after subtracting noise from the image.

There are many ways use stable diffusion with conditioning, like inpainting, image-to-image, etc. We will focus on explaining text-to-image, depth-to-image and LoRA.

1.3.2.1. Text conditioning (text-to-image)

On Figure 8 is an overview of how a text prompt is processed and fed into the noise predictor. Tokenizer first converts each word in the prompt to a number called a token. Each token is then converted to a 768-value embedding vector. The embeddings are then processed by the text transformer and are ready to be consumed by the noise predictor.

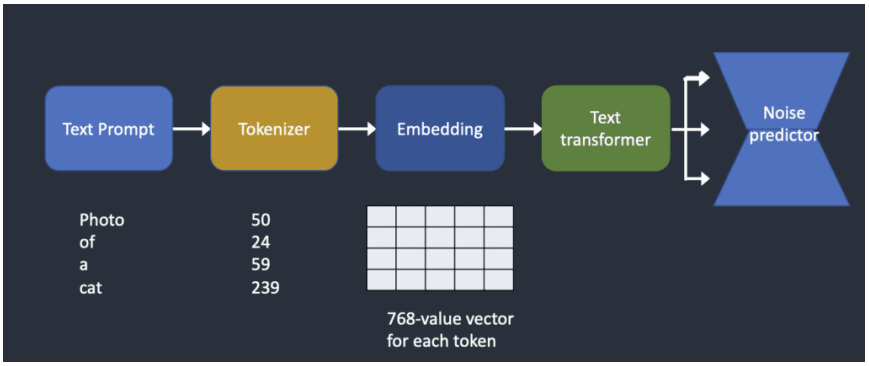

Figure 8 - How the text prompt is processed and fed into the noise predictor to steer image generation.

Now let's look closer into each part:

Tokenizer

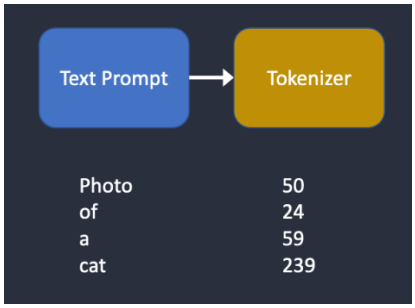

Figure 9 – Tokenizer

The text prompt is first tokenized by a CLIP tokenizer[11][12][13]. CLIP is a deep learning model developed by Open AI to produce text descriptions of any images. Stable Diffusion v1 uses CLIP's tokenizer.

A tokenizer can only tokenize words it has seen during training. For example, there are "dream" and "beach" in the CLIP model but not "dreambeach". Tokenizer would break up the word "dreambeach" into two tokens "dream" and "beach". So, one word does not always mean one token.

Embedding

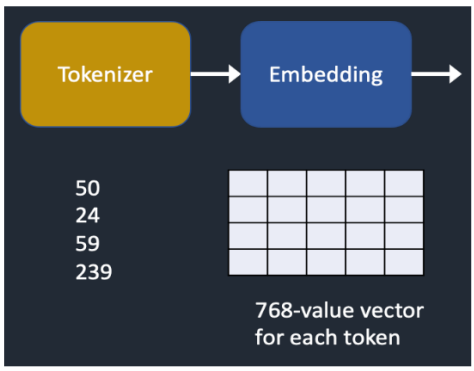

Figure 10 – Embedding.

Stable diffusion uses Open AI's ViT-L/14 CLIP model. An embedding is a 768-value vector. Each token has its own unique embedding vector. The embedding is fixed by the CLIP model, which is learned during training.

Embeddings allow representing words in an n-dimensional space, where depending on their sematic similarity the vectors are closer or further apart. We want to take advantage of this information. For example, the embeddings of man, gentleman, and guy are nearly identical because they can be used interchangeably. Monet, Manet, and Degas all painted in impressionist styles but in different ways. The names have close but not identical embeddings.

Feeding embeddings to noise predictor

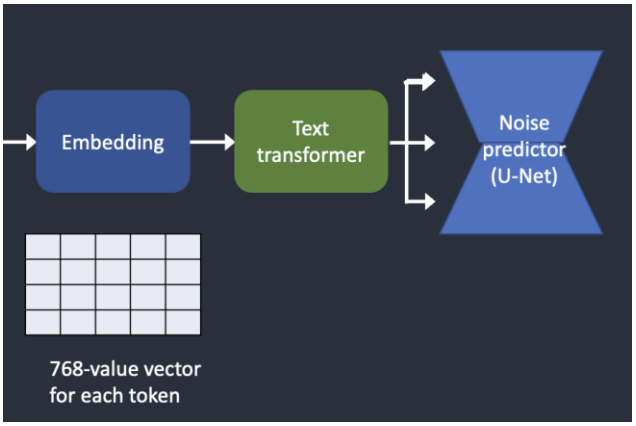

Figure 11 - From embeddings to the noise predictor.

The embedding needs to be further processed by the text transformer before feeding into the noise predictor. The transformer is like a universal adapter for conditioning. In this case, its input is text embedding vectors, but it could as well be something else like class labels, images, and depth maps. The transformer not only further processes the data but also provides a mechanism to include different conditioning modalities.

Cross-attention

The output of the text transformer is used multiple times by the noise predictor throughout the U-Net. The U-Net consumes it by a cross-attention mechanism. That's where the prompt meets the image.

Let's use the prompt "A man with blue eyes" as an example. Stable Diffusion pairs the two words "blue" and "eyes" together (self-attention within the prompt) so that it generates a man with blue eyes but not a man with a blue shirt. It then uses this information to steer the reverse diffuse towards images containing blue eyes. (cross-attention between the prompt and the image)

As a side note, hypernetwork, a technique to fine-tune Stable Diffusion models, hijacks the cross-attention network to insert styles. LoRA[14][15] models modify the weights of the cross-attention module to change styles. The fact that modifying this module alone can fine-tune a Stabe Diffusion model tells you how important this module is.

Stable Diffusion step-by-step

Step 1. Stable Diffusion generates[40] a random tensor in the latent space. You control this tensor by setting the seed of the random number generator. If you set the seed to a certain value, you will always get the same random tensor.

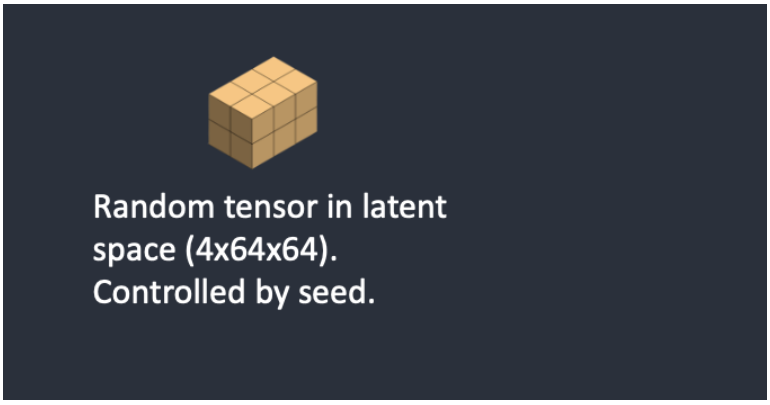

Figure 12 - A random tensor is generated in latent space.

Step 2. The noise predictor U-Net takes the latent noisy image and text prompt as input and predicts the noise, also in latent space (a 4x64x64 tensor).

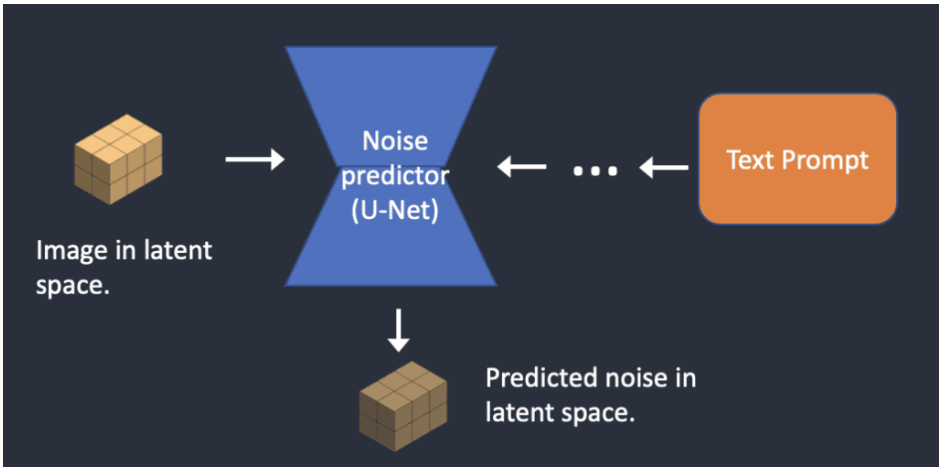

Figure 13 – U-net takes the compressed image and text and gives a predicted noise.

Step 3. Subtract the latent noise from the latent image. This becomes your new latent image.

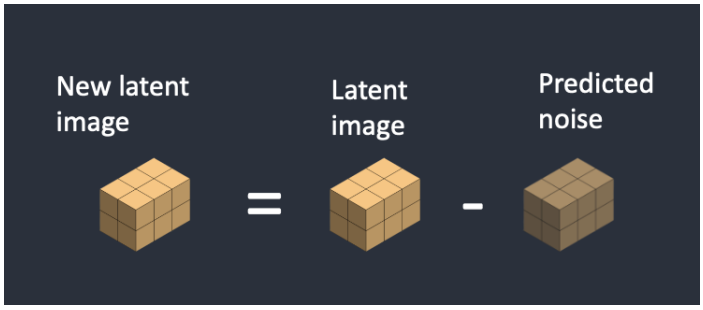

Figure 14 – Predicted noise subtraction.

Steps 2 and 3 are repeated for a certain number of sampling steps, for example, 20 times.

Step 4. Finally, the decoder of VAE converts the latent image back to pixel space. This is the image you get after running Stable Diffusion.

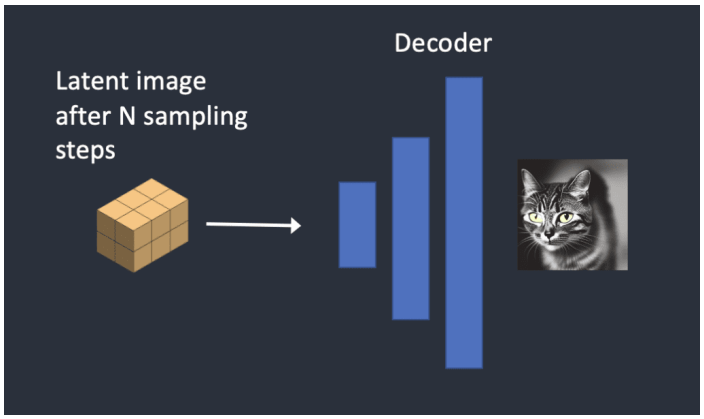

Figure 15 – Image recovered from latent space.

1.3.2.2. Depth conditioning (depth-to-image)

To use the depth to mage conditioning we will use ControlNet[16][17]. ControlNet conditions the noise predictor with detected outlines, human poses, etc, and achieves excellent controls over image generations.

ControlNet refers to a group of neural networks refined using Stable Diffusion, which empowers precise artistic and structural control in generating images. It improves default Stable Diffusion models by incorporating task-specific conditions.

ControlNet provides us control over prompts through task-specific adjustments. For this to be effective, ControlNet has undergone training to govern a substantial image diffusion model. This enables it to grasp task-specific adjustments from both the prompt and an input image.

ControlNet, functioning as a complete neural network structure, takes charge of substantial image diffusion models, like Stable Diffusion, to grasp task-specific input conditions. ControlNet achieves this by replicating the weights of a major diffusion model into both a “trainable copy” and a “locked copy.” The locked copy conserves the learned network prowess from vast image data, while the trainable copy gets trained on task-specific datasets to master conditional control.

This process connects trainable and locked neural network segments using an exceptional convolution layer called “zero convolution.” In this layer, convolution weights progressively evolve from zeros to optimal settings through a learned approach. This strategy maintains the refined weights, ensuring strong performance across various dataset scales. Importantly, because zero convolution doesn’t introduce extra noise to deep features, the training speed matches that of fine-tuning a diffusion model. This contrasts with the lengthier process of training entirely new layers from scratch.

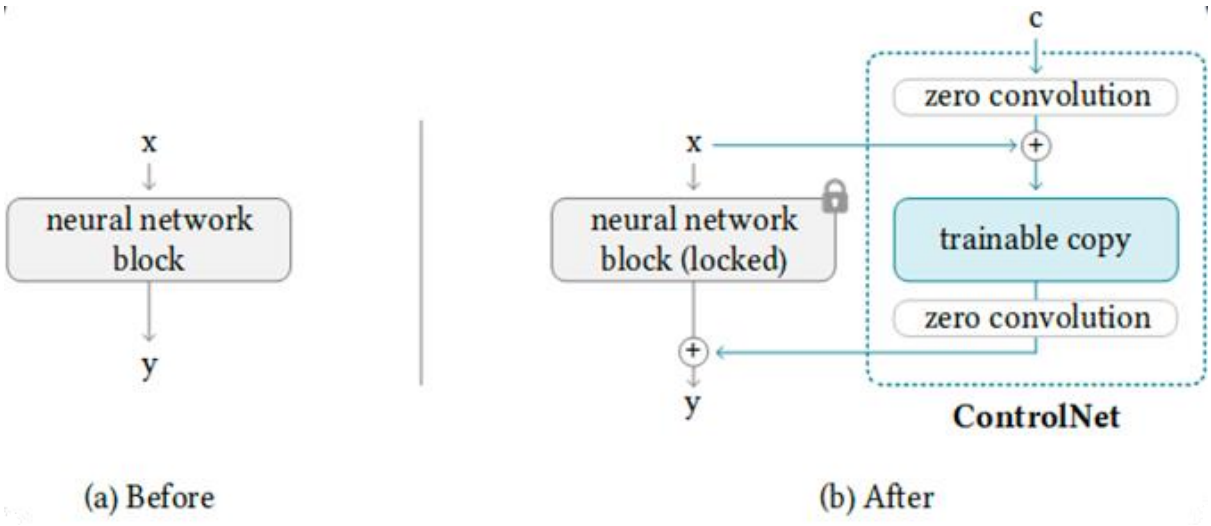

Figure 16 – a) Network structure without ControlNet b) Network structure with ControlNet

The Stable Diffusion Block Before and After the ControlNet Connections

Some of the ControlNet models come with a preprocessor. This is an additional model responsible to turn a normal a normal image into another that can be used as input to the corresponding model. For example, in the case of depth conditioning, we may use MiDaS[18] to infer the depth map for a simple RGB image, and then use that map to generate a new one.

Currently the main control net models are:

- Canny edge

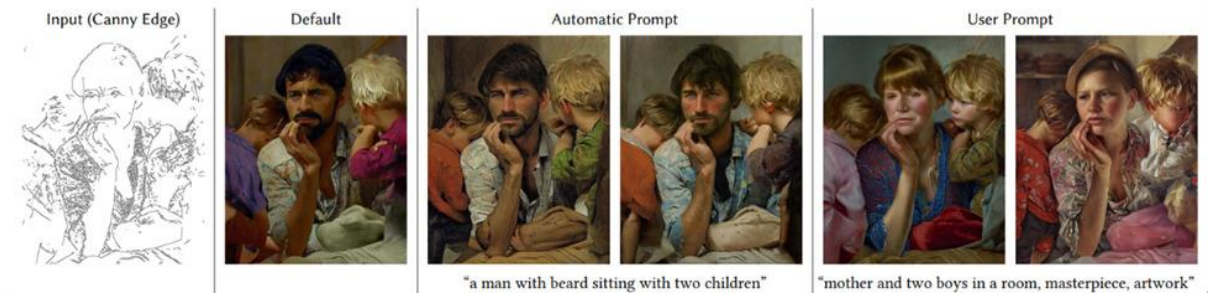

Figure 17 – Examples generated with canny edge conditioning.

- Line art

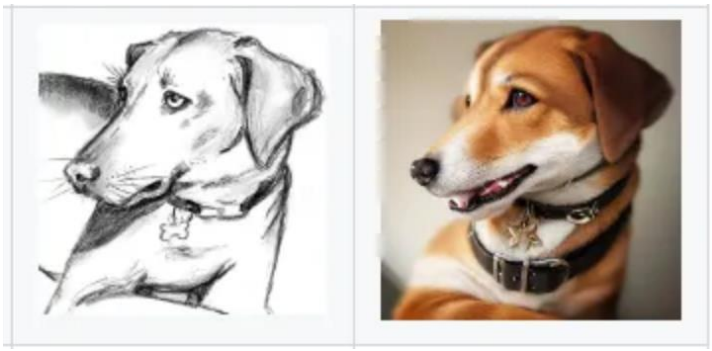

Figure 18 - Example generated with line art conditioning.

- Scribble

Figure 19 - Examples generated with scribble conditioning.

- Hough line

Figure 20 – Examples generated with Hough line conditioning.

- Semantic segmentation

Figure 21 - Examples generated with segmentation map conditioning.

- Normal map

Figure 22 – Examples generated with normal map conditioning.

- Depth map

Figure 23 – Original image.

Figure 24 – Depth map extracted with MiDaS preprocessor.

Figure 25 – Example generated with depth map conditioning.

- Open pose

Figure 26 – Examples generated with open pose conditioning.

1.3.2.3. LoRA

LoRA (Low-Rank Adaptation) is a training technique for fine-tuning Stable Diffusion models. It is an alternative to techniques such as Dreambooth[19] and textual inversion[20]. LoRA offers a good trade-off between file size and training power. Dreambooth is powerful but results in large model files (2-7 GBs). Textual inversions are tiny (about 100 KBs), but you can't do as much with them.

LoRA sits in between. Its file size is much more manageable (2 – 200 MBs), and the training power is decent.

Stable Diffusion models quickly fill one's local storage. Because of the large size, it is hard to maintain a collection within a personal computer. LoRA is an excellent solution to the storage problem.

Like textual inversion, you cannot use a LoRA model alone. It must be used with a model checkpoint file. LoRA modifies styles by applying small changes to the accompanying model file.

LoRA is a great way to customize AI art models without filling up local storage.

LoRA applies small changes to the most critical part of Stable Diffusion models: The cross-attention layers. It is the part of the model where the image and the prompt meet. Researchers found it sufficient to fine-tune this part of the model to achieve good training. The cross-attention layers are the yellow parts in the Stable Diffusion model architecture on Figure 27.

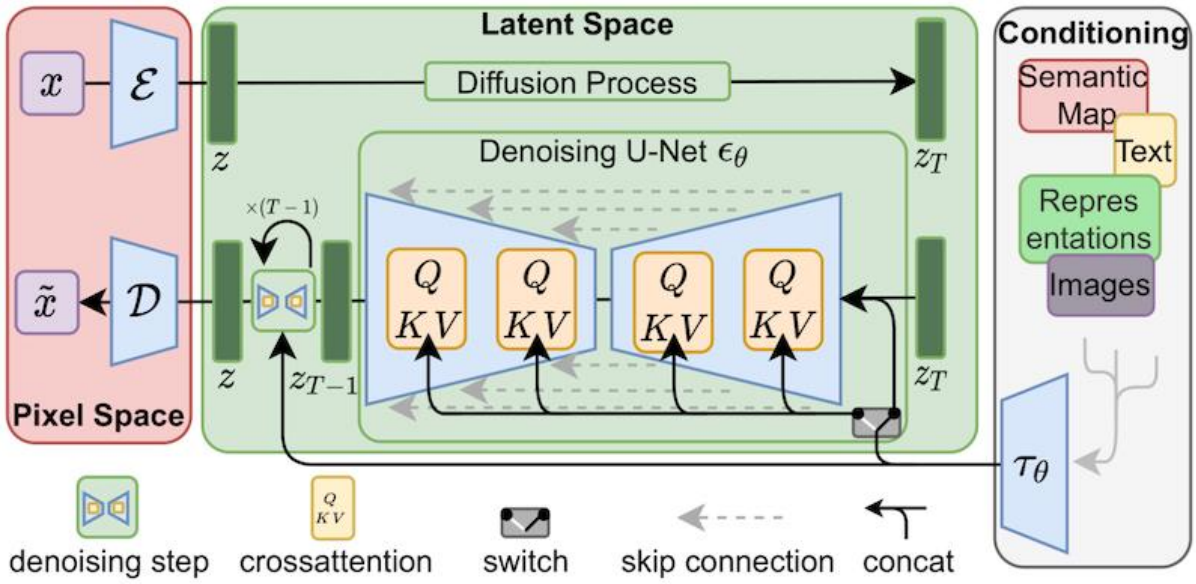

Figure 27 - LORA fine-tunes the cross-attention layers (the QKV parts of the U-Net noise predictor).

The weights of a cross-attention layer are arranged in matrices. A LoRA model fine-tunes a model by adding its weights to these matrices.

The trick for LoRA model files being smaller even when they need to store the same number of weights, is breaking a matrix into two smaller (low-rank) matrices. It can store a lot fewer numbers by doing this. Let's illustrate this with the following example.

Let's say the model has a matrix with 1,000 rows and 2,000 columns. That's 2,000,000 numbers (1,000 x 2,000) to store in the model file. LoRA breaks down the matrix into a 1,000-by-2 matrix and a 2-by-2,000 matrix. That's only 6,000 numbers (1,000 x 2 + 2 x 2,000), 333 times less. That's why LoRA files are a lot smaller.

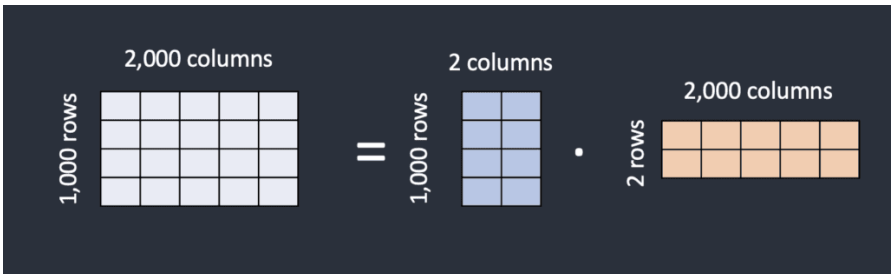

Figure 28 - LoRA decomposes a big matrix into two small, low-rank matrices.

In the example on Figure 28, the rank of the matrices is 2. It is much lower than the original dimensions, so they are called low-rank matrices. The rank can be as low as 1.

So far, researchers found doing that in cross-attention layers did not affect the power of fine-tuning much.

1.3.2.4. CFG value

Classifier guidance

Classifier guidance[21] is a way to incorporate image labels in diffusion models. You can use a label to guide the diffusion process. For example, the label "cat" steers the reverse diffusion process to generate photos of cats.

The classifier guidance scale is a parameter for controlling how closely should the diffusion process follow the label.

On the Figure 29 we can see an example. Suppose there are 3 groups of images with labels "cat", "dog", and "human". If the diffusion is unguided, the model will draw samples from each group's total population, but sometimes it may draw images that could fit two labels, e.g. a boy petting a dog.

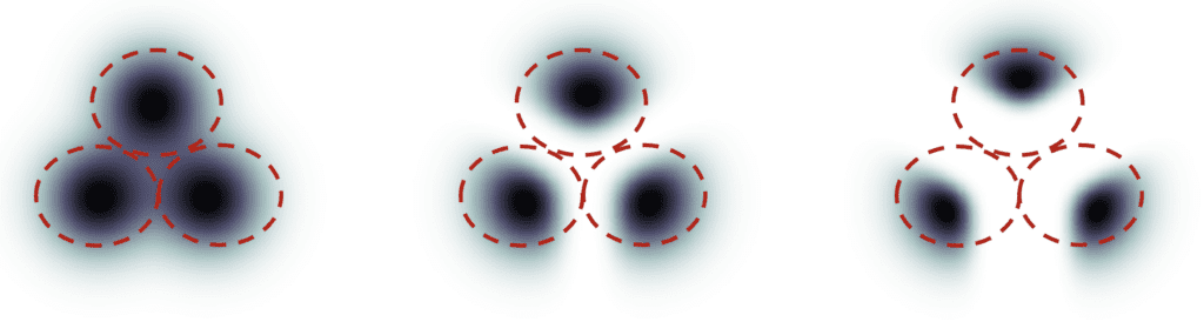

Figure 29 - Classifier guidance. Left: unguided. Middle: small guidance scale. Right: large guidance scale.

With high classifier guidance, the images produced by the diffusion model would be biased toward the extreme or unambiguous examples. If you ask the model for a cat, it will return an image that is unambiguously a cat and nothing else.

The classifier guidance scale controls how closely the guidance is followed. In the figure above, the sampling on the right has a higher classifier guidance scale than the one in the middle. In practice, this scale value is simply the multiplier to the drift term toward the data with that label.

Classifier-free guidance

Although classifier guidance achieved record-breaking performance, it needs an extra model to provide that guidance. This has presented some difficulties in training.

Classifier-free guidance[22], in its authors' terms, is a way to achieve "classifier guidance without a classifier". Instead of using class labels and a separate model for guidance, they proposed to use image captions and train a conditional diffusion model, exactly like the one we discussed in text-to-image.

They put the classifier part as conditioning of the noise predictor U-Net, achieving the so-called "classifier-free" (i.e., without a separate image classifier) guidance in image generation. The text prompt provides this guidance in text-to-image.

Classifier-free guidance scale

The classifier-free guidance scale (CFG scale) is a value that controls how much the text prompt steers the diffusion process. The AI image generation is unconditioned (i.e. the prompt is ignored) when the CFG scale is set to 0. A higher CFG scale steers the diffusion towards the prompt.

1.3.3. Models

Stable Diffusion v1 vs Stable Diffusion v2

Stable Diffusion v2[23][27] uses OpenClip for text embedding. Stable Diffusion v1 uses Open AI's CLIP ViT-L/14 for text embedding. The reasons for this change are:

- OpenClip is up to five times larger. A larger text encoder model improves image quality.
- Although Open AI's CLIP models are open source, the models were trained with proprietary data. Switching to the OpenClip model gives researchers more transparency in studying and optimizing the model. It is better for long-term development.
- The v2 models come in two flavours.
    - The 512 version generates 512×512 images.
    - The 768 version generates 768×768 image.
- Training data difference
- Stable Diffusion v1.5 is trained with:
    - 237k steps at resolution 256×256 on laion2B-en[24] dataset.
    - 194k steps at resolution 512×512 on laion-high-resolution**Error! Reference source not found.**[25].
    - 225k steps at 512×512 on "laion-aesthetics v2 5+" [26], with a 10% dropping in text conditioning.
- Stable Diffusion v2 is trained with:
    - 550k steps at the resolution 256x256 on a subset of LAION-5B[28] filtered for explicit pornographic material, using the LAION-NSFW classifier[29] with punsafe=0.1 and an aesthetic score[30] >= 4.5.
    - 850k steps at the resolution 512x512 on the same dataset on images with resolution >= 512x512.
    - 150k steps using a v-objective**Error! Reference source not found.**[31] on the same dataset.
    - Resumed for another 140k steps on 768x768 images.

Outcome difference

Users generally find it harder to use Stable Diffusion v2 to control styles and generate celebrities. Although Stability AI did not explicitly filter out artist and celebrity names, their effects are much weaker in v2. This is likely due to the difference in training data. Open AI's proprietary data may have more artwork and celebrity photos. Their data is probably highly filtered so that everything and everyone looks fine and pretty.

The v2 and v2.1 models are not popular. People use the fine-tuned v1.5 and SDXL models exclusively.

SDXL model

The SDXL[32][33] model is the official upgrade to the v1 and v2 models. The model is released as open-source software.

It is a much larger model. In the AI world, we can expect it to be better. The total number of parameters of the SDXL model is 6.6 billion, compared with 0.98 billion for the v1.5 model.

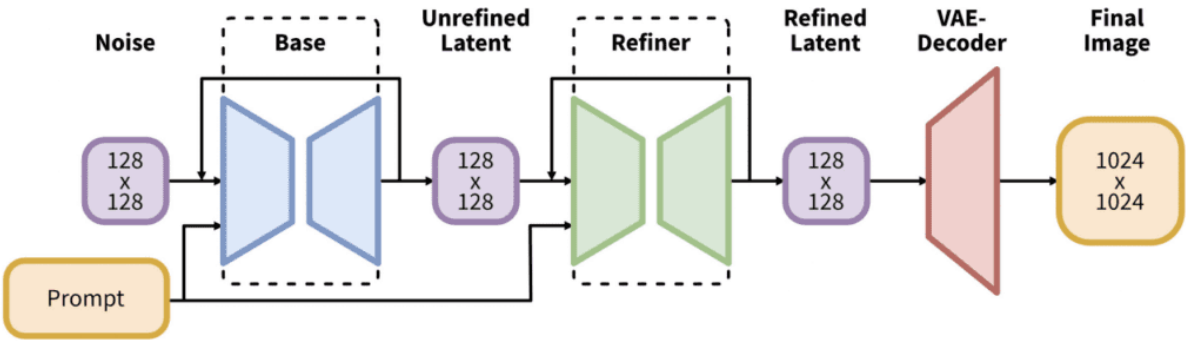

Figure 30 - The SDXL pipeline consists of a base model and a refiner model.

The SDXL model is, in practice, two models. You run the base model, followed by the refiner model. The base model sets the global composition. The refiner model adds finer details. You can run the base model alone without the refiner.

The changes in the SDXL base model are:

- The text encoder combines the largest OpenClip model (ViT-G/14) and OpenAI's proprietary CLIP ViT-L. It is a smart choice because it makes SDXL easy to prompt while remaining the powerful and trainable OpenClip.
- New image size conditioning that aims to use training images smaller than 256×256. This significantly increases the training data by not discarding 39% of the images.
- The U-Net is three times larger than v1.5.
- The default image size is 1024×1024. This is 4 times larger than the v1.5 model's 512×512. (See image sizes to use with the SDXL model)

1.3.4. Ethical, moral, and legal issues

Stable Diffusion, like other AI models, needs a lot of data for training. Its first version used images without copyright permissions. Though this might be okay for research, its commercial popularity drew attention from copyright owners and lawyers.

Three artists filed a class-action lawsuit[34] against Stability AI, the commercial partner of Stable Diffusion, in January 2023. The next month, the image giant Getty filed another lawsuit[35] against it.

The legal and moral issue is complicated. It questions whether a computer creating new works by imitating artist styles counts as copyright infringement.

1.4. On sandbox

Figure 31 – UC Davis' logo

The Augmented Reality Sandbox project[43], led by UC Davis' W.M. Keck Center for Active Visualization in the Earth Sciences, is an NSF-funded initiative in informal science education for freshwater lake and watershed science. The project focuses on developing 3D visualization applications for teaching earth science concepts, primarily utilizing a combination of real sand, a Microsoft Kinect 3D camera, simulation software, and a data projector. The resulting Augmented Reality (AR) sandbox allows users to shape real sand to create topography models, which are then augmented in real time with elevation colour maps, topographic contour lines, and simulated water.

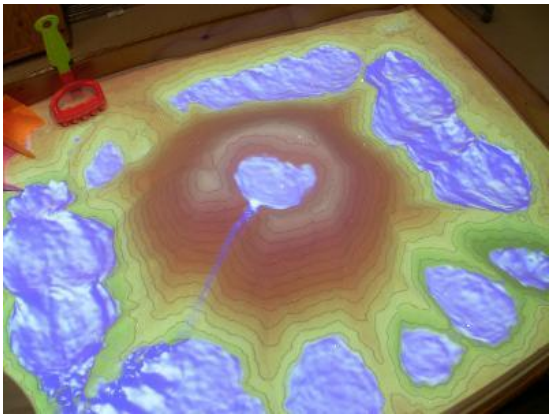

Figure 32 - Sandbox table when turned on, showing a mountain with a crater lake, surrounded by several lower lakes.

Figure 33 - Sandbox unit when turned off. The Kinect 3D camera and the digital projector are suspended above the sandbox proper from the pole attached to the back.

Inspired by Czech researchers and an earlier project called Project Mimicry, the goal is to create a self-contained, hands-on exhibit for science museums. The AR Sandbox software, based on the Vrui VR development toolkit and Kinect 3D video processing framework, is available for download under the GNU General Public License. The system employs GLSL shaders to render the topography surface and simulate water flow based on shallow water equations. The project has gained traction,

with a growing user community building their own versions globally. Notably, the AR Sandbox has been showcased at various events, including the White House Water Summit and the USA Science and Engineering Festival. Publications and a user support forum contribute to the project's ongoing development and dissemination.

2. Project Implementation

The final implementation of the project is divided in the following parts:

2.1. Stable Diffusion Backend

Our initial option was programming our own implementation of the code to work with Stable Diffusion based on the different APIs found in the diffusers' repository at Hugging Face[Error! Reference source not found.][37]. We did achieve to make some simple generations solely with SD models, but when it came to modifying the pipeline to include control-Net or LoRA in it, we had problems.

That is because the Diffusers library works more as a repository for all the different developments related with diffusers. This means that you end up with a section dedicated to Stable Diffusion, another to ControlNet, another to Wuerstchen, etc. At the end, each tool really works mostly by itself. There are some specific community-created pipeline examples, but none integrating the three core elements we were interested in.

Instead, we opted to use an already built tool that would satisfy all the requirements. It is Automatic1111's Stable Diffusion web-UI[38][39]. It can be found on Automatic1111's GitHub repository and is both highly integrated with tools like ControlNet and LoRA, aside from giving easy access to other parameters for the generations. Thanks to the extensions the open-source community has created to be easily integrated with the web-UI, you get easy access to all the conditioning parameter of the SD models.

Once you have cloned the repository to a local storage you can run the *webui-user.bat* file. It will automatically install possible required dependencies and, in recent versions, also download the *v1-5-pruned-emaonly.safetensors*, which is the file containing the weights of Stable Diffusion v1.5.

Once all is installed, the web-UI launches a local server, running on https://127.0.0.1:8760, to which you can connect through a web explorer and work manually with its interface. The parts of the interface we would be interested in are the ones showed on Figure 34, Figure 35 and Figure 36.

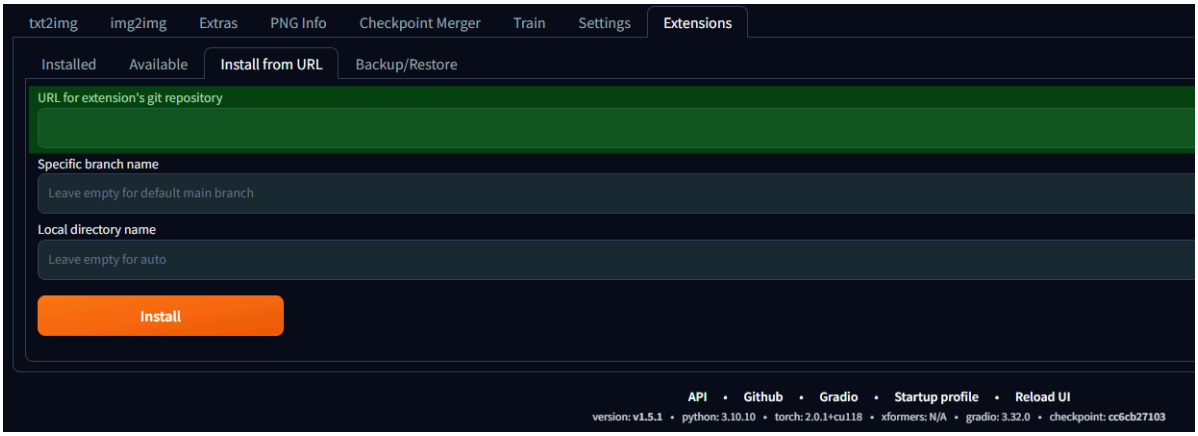

Figure 34 – Extensions installation tab in the Automatic1111's web-UI.

Here we can see a snapshot of the window used to instal new extensions. Most of them are available on GitHub repositories, so they can be directly installed from there, although it also leaves the possibility of using local directories, which in term means that you can create of modify your own extensions.

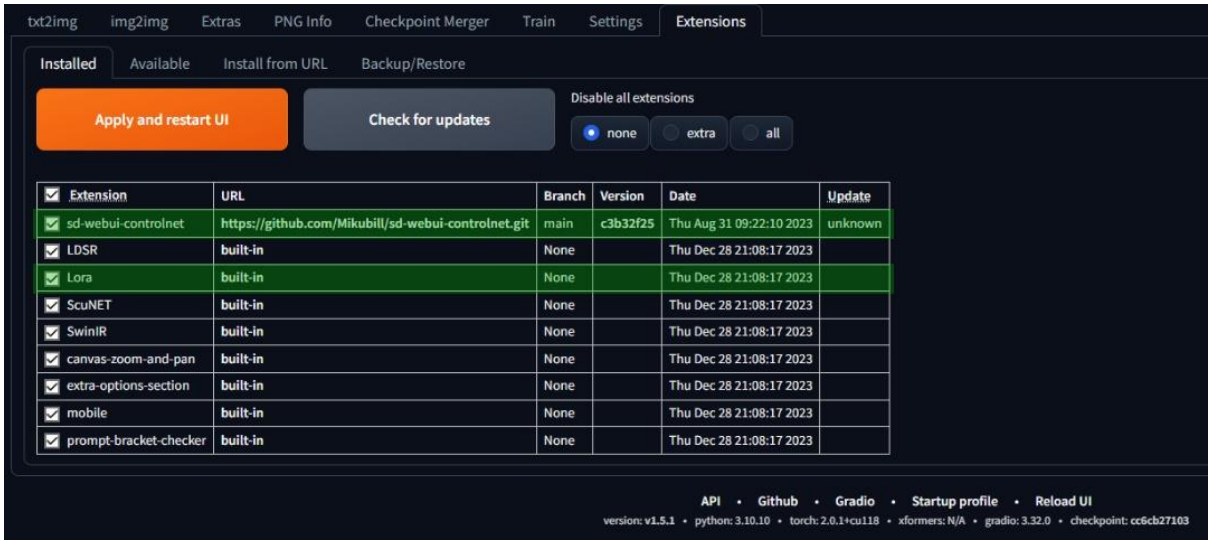

Figure 35 – Installed extensions tab in the Automatic1111's web-UI.

Once the extensions are installed, they are listed in this other tab. Here we can see that we have currently installed a ControlNet extension from Mikubill's GitHub repository, and we got a built-in extension to work with LoRAs. One important think to note is the version field. Some of the extension may present slightly different behaviours depending on it.

On the general overview of the web-UI we can see the following sections of interest:

1. Stable Diffusion model selector, useful when using multiple models, like custom made Dreambooth retraining or simply other Stable Diffusion versions.
2. Positive and negative prompt input fields
3. LoRA selector button, where one can add the trigger tokens for the different LoRA files stored locally.
4. Parameter control panel, where we can adjust the parameters, Stable Diffusion uses to generate images.
5. ControlNet conditioning image input field. In the version of the web-UI displayed there is even the option of using several different images for the generation or even several different ControlNet models.
6. Generation output field, where the created images are displayed, as well as the possible images used for the process.
7. Generation information field, where the values of some of the parameters are displayed, as well as some data on the generation itself, like the used VRAM or time.
8. ControlNet model selector. Currently the depth model for Stable Diffusion v1.5 is selected but we can see a list of the locally downloaded alternatives.

Other fields of interest are:

1. The ControlNet preprocessor selector, for the models that have it.
2. The ControlNet weight, which tell how much influence the conditioning image should have over the final result.

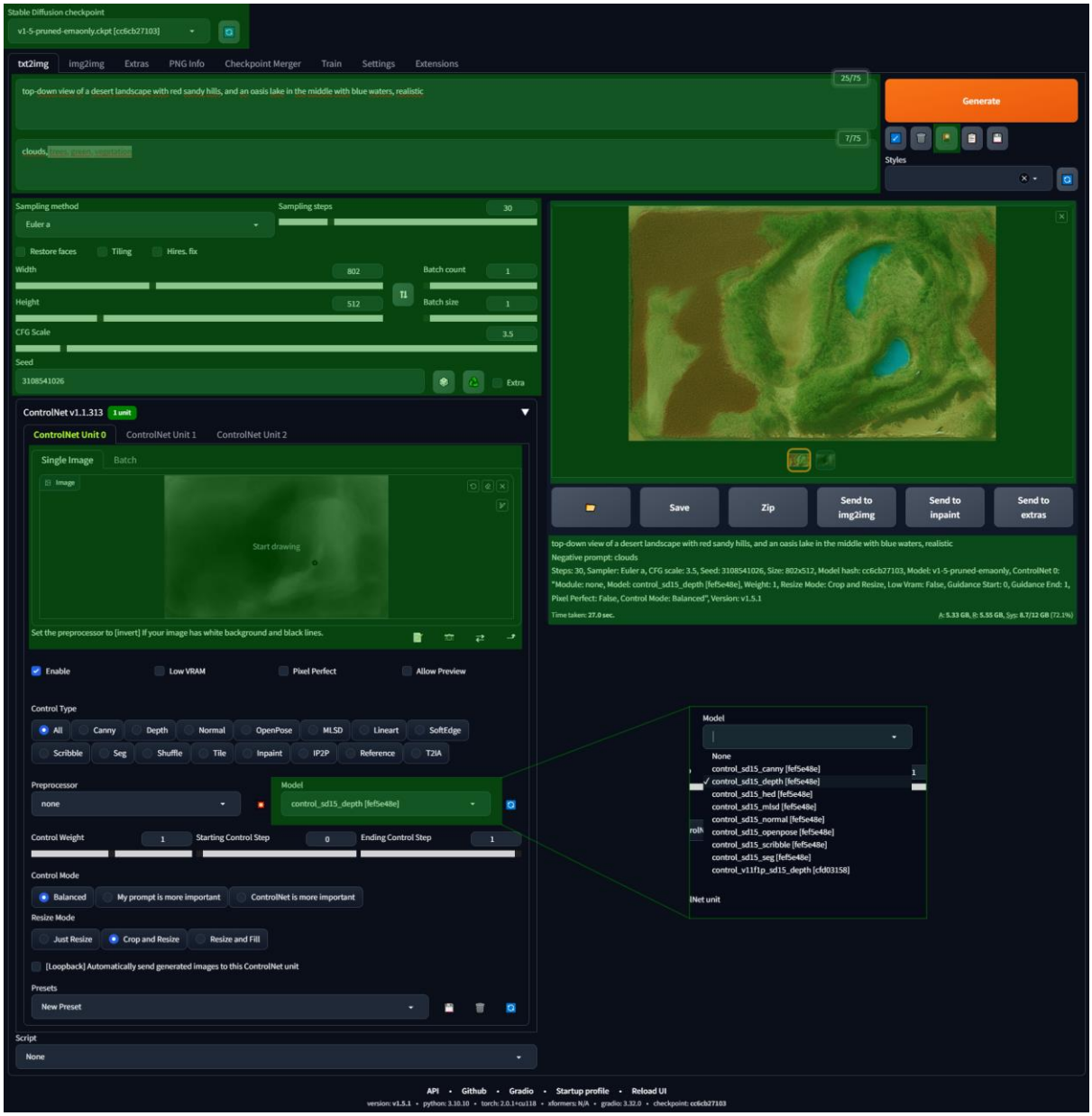

Figure 36 – Main view in the Automatic1111's web-UI once ControlNet extension is stalled.

In our case, we required a way to interact with the web-UI automatically. An initial idea we had to integrate it into our pipeline was using Selenium python library. This is a web driver-based automatization python library which lets the user emulate sequential human interactions with a web browser. Ideally a script in python that used the library would be launched whenever a new image is required. This script would open a new browser and perform the necessary actions to obtain such image. The process would like something like the following:

1. Open a new browser window on the https://127.0.0.1:8760 URL.

2. Input the prompt, negative prompt, and adjust the values of the different sliders.
3. Activate the ControlNet extension.
4. Load the conditioning image by clicking on the field and interacting with the floating window.
5. Select the ControlNet configuration and model.
6. Click on the “Generate” button.
7. Wait until the image is fully generated.
8. Return control to the code that started the script.

From there the code would read the image from the output folder and perform any necessary actions.

Later we discovered that by launching the *webui-user.bat* file with the “–api” flag, it would allow for other programs to directly interact with it. These interactions would be made through the HTTP POST request method to the https://127.0.0.1:8760/sdapi/v1/txt2img URL. In this petition, a JSON is sent with all the required data to start a new generation.


```
{
      "enable_hr": false,
      "denoising_strength": 0,
      "firstphase_width": 0,
      "firstphase_height": 0,
      "hr_scale": 2,
      "hr_upscaler": "string",
      "hr_second_pass_steps": 0,
      "hr_resize_x": 0,
      "hr_resize_y": 0,
      "hr_sampler_name": "string",
      "hr_prompt": "string",
      "hr_negative_prompt": "string",
      "prompt": "string",
      "styles": [
            "string"
      ],
      "seed": -1,
      "subseed": -1,
      "subseed_strength": 0,
      "seed_resize_from_h": -1,
      "seed_resize_from_w": -1,
      "sd_model_hash": "none",
      "sampler_name": "string",
      "batch_size": 1,
      "n_iter": 1,
      "steps": 50,
      "cfg_scale": 7,
      "width": 512,
      "height": 512,
      "restore_faces": false,
      "tiling": false,
      "do_not_save_samples": false,
      "do_not_save_grid": false,
      "negative_prompt": "string",
      "eta": 0,
      "s_min_uncond": 0,
      "s_churn": 0,
      "s_tmax": 0,
```

```
        "s_tmin": 0,
        "s_noise": 1,
        "override_settings": {},
        "override_settings_restore_afterwards": true,
        "script_args": [],
        "sampler_index": "Euler",
        "script_name": "string",
        "send_images": true,
        "save_images": false,
        "alwayson_scripts": {
              "controlnet": {
                          "args": [
                                {
                                      "input_image": "base64",
                                      "mask":"string",
                                      "module": "none",
                                      "model": "none",
                                      "weight":1,
                                      "resize_mode":0,
                                      "low_vram":false,
                                      "processor_res":64,
                                      "threshold_a":64,
                                      "threshold_b":64,
                                      "guidance_start": 0.0,
                                      "guidance_end": 1.0,
                                      "control_mode":0,
                                      "pixel_perfect":false,
                                }
                          ]
              }
        }
}
```

Code 1 – JSON structure to be sent in the POST petition for generating an image.

As we can see there are many parameters to work with, but we are most interested in toying with the following ones:

- steps: The “steps” parameter establishes the number of times the denoising process is applied on the image. The more steps the sharper the details, but the more, the less impactful each of them is on the final result. Thus, there is a threshold one can find where adding more steps will increase the generation time without making noticeable difference on the image.
- cfg_scale: The CFG scale (Classifier-free guidance scale) is a parameter that determines how closely should the generation process follow the input prompt. Depending on the contents of the image there is a value around which the best results are achieved. Move further away from it and the quality becomes lower. In practice, it is also reflected on the saturation of the image. The lower the value, the greyer the image is, and go too high and the colours become excessively vibrant.
- width and height: Indicates the size of the image to be generated. These values are relevant to influence the speed of the image generated. Ideally with Stable Diffusion v1.5, which is the version picked for the project, the image size should be of 512x512px because that is the one the model was designed for. If we make slightly larger images, the execution time drops significantly. An approach we took here was downscaling the conditioning images so that

their shorter side matched the 512px. It still reduced the speed, but worked faster than if we were to use full scale images.

- prompt and negative_prompt: They are both used to tell the model, what should it generate. The first is used to describe the contents one would like the image to have and the negative those to be avoided. Their syntaxis and structure depend on the model used, as some have been retrained on custom datasets, using for example Dreambooth, and the images in those were described using tags instead of plain descriptions.
- seed: Allows for replicability of results and studying the influence of the different parameters on the final result. It is set to -1 when we want it to be random. When used this way, we can recover the actual seed used by accessing the https://127.0.0.1:8760/sdapi/v1/png-info URL.
- sd_model_hash: A value associated to the Stable Diffusion *.safetensors* file used for the generation of the image.
- sampler_name or sampler_index: Both these values can be used to indicate the sampler we want for the generation. The sampler determines how the predicted noise is generated, which is used in the denoising steps. There is a list of values to choose from and depending on which of them we choose, the final result, the convergence speed or stability, etc. will change.

Then we have the ControlNet-specific parameters, which are stored inside the alwayson_scripts->controlnet->args list. Here the values we work with are:

- model: The name of the ControlNet model to be used to condition the image generation process.
- input_image: The image the ControlNet model will use to condition the generation process. Given that it is sent through a HTTP POST petition, we first have to encode it into a format that can be sent.

As for the LoRA models, those are activated with the corresponding tokens, there is no external parameters used for that.

2.1.1. Finding the best parameters

For determining the values for the parameters to be used as a basis we executed a series of tests with a fixed seed, prompts and conditioning image.

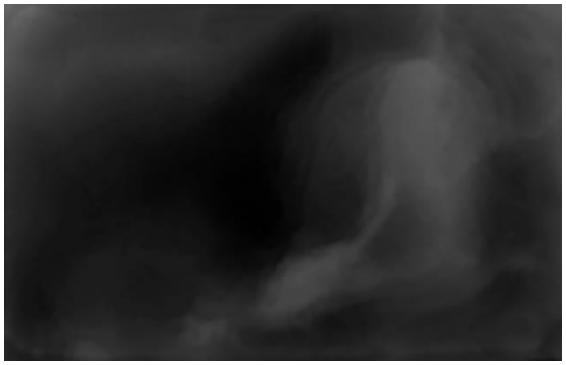

Figure 37 – Depth image used for parameters testing.

Next is a matrix showing the results depending on the CFG-scale and the number of steps. The values of the parameters used are the following:

- prompt: bird's eye view of a landscape with green hills, a few trees, and a lake in the middle with crystal clear blue water, realistic
- negative_prompt: clouds
- sd_model_hash: cc6cb27103 (Stable Diffusion v1.5)
- sampler_name: Euler a
- seed: 3108541024

Here the negative prompt being “clouds” helps the model stir away from satellite-like images, where it is common to have patches covered by cloud formations. We, on the other hand, are more interested in more close-up images.

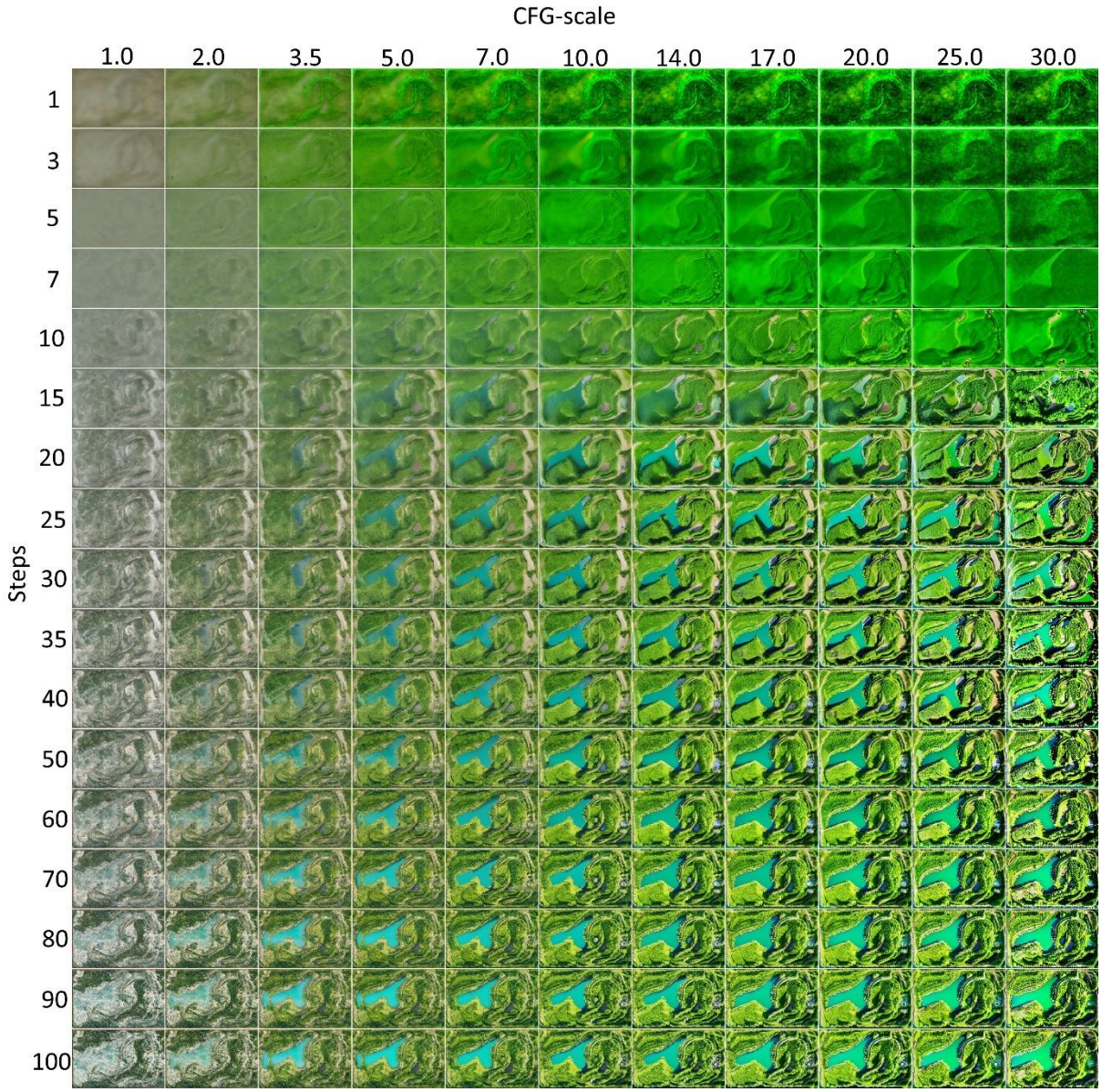

Figure 38 – Grid of images generation depending on CFG-scale [1-30] and number of steps [1-100].

We observe that any value under 15 for number of steps gives an output a bit distant from what we intended in the prompt. We can see a green landscape, but there is no lake anywhere to be seen. We can take a closer look at some of the results on the Figure 39-Figure 42. Thus, we will use at least a value of 15, which is where we begin to see better results.

Figure 39 – CFG-scale 3.5 Nº steps 10

Figure 40 – CFG-scale 7 Nº steps 10

Figure 41 – CFG-scale 17 Nº steps 10

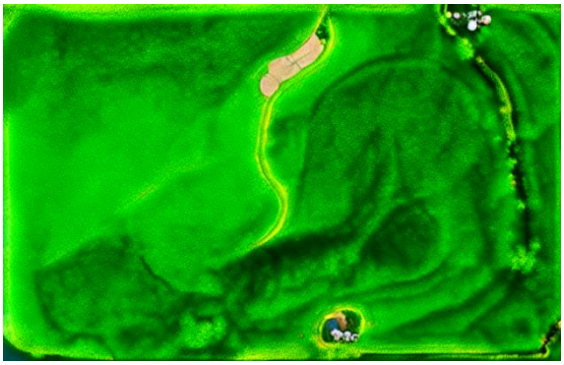

Figure 42 – CFG-scale 30 Nº steps 10

Next, we discuss the CFG-scale that should be used. The higher the CFG-scale, the more saturated and vibrant the colours look, and the more steps we need to use to compensate for that saturation. At a value of 17, we already see a bit plastic-like colours that don't really suit the "realistic" landscape view we were aiming for.

Figure 43 - CFG-scale 17 Nº steps 15

Figure 44 - CFG-scale 17 Nº steps 35

Figure 45 - CFG-scale 17 Nº steps 70

Figure 46 - CFG-scale 17 Nº steps 100

If we go for higher values still, this contrast is exaggerated even further. The result can be observed in the FIGURES with 30 CGF-scale. We still can counteract its high saturation by increasing the step's number. The colours are still very vibrant, but not as intense as when less steps were used. Another side effect to be noted, is the appearance of sharper shadows on the landscape.

Figure 47 - CFG-scale 30 Nº steps 15

Figure 49 - CFG-scale 30 Nº steps 35

Figure 48 - CFG-scale 30 Nº steps 70

Figure 50 - CFG-scale 30 Nº steps 100

Particularly for landscapes the results we were most fond of where those under 7 CFG-scale. These are the ones that gave us the best result. We can see what could be rocky formations amongst the vegetation, which also has a much more believable colours. The same applies to the water which looks more natural that on the images with higher CFG-scale.

Figure 51 – CFG-scale 3.5 Nº steps 15

Figure 53 – CFG-scale 7 Nº steps 15

Figure 52 – CFG-scale 3.5 Nº steps 50

Figure 54 – CFG-scale 7 Nº steps 50

Another point worth noticing is that if we use a CFG-scale of 1 or 2 we get really bleached out results. They do get better when we greatly increase the number of steps but still look stranger compared to those got with a bigger value. They have a mosaic-like pattern, which shouldn't bee too noticeabble once projected, but still that is not exactly what we are aimimg for.

Figure 55 - CFG-scale 1 Nº steps 15

Figure 56 - CFG-scale 2 Nº steps 15

Figure 57 - CFG-scale 1 Nº steps 70

Figure 58 - CFG-scale 2 Nº steps 70

From the images of the previous grid on the FIGURE we consider the best area of interest to be >15 iterations and between 2 and 7 for the CFG-scale. With that we make a new one, where we consider smaller steps for the CFG-scale to narrow it further yet. We can see the result on FIGURE. There we observe that for the value 5 and higher, the lake becomes a bit too plain, and here even when increasing the steps, the result doesn't seem to become better. Rather the opposite.

Figure 59 – CFG-scale 5 Nº steps 35

Figure 61 – CFG-scale 7 Nº steps 35

Figure 60 – CFG-scale 5 Nº steps 80

Figure 62 – CFG-scale 7 Nº steps 80

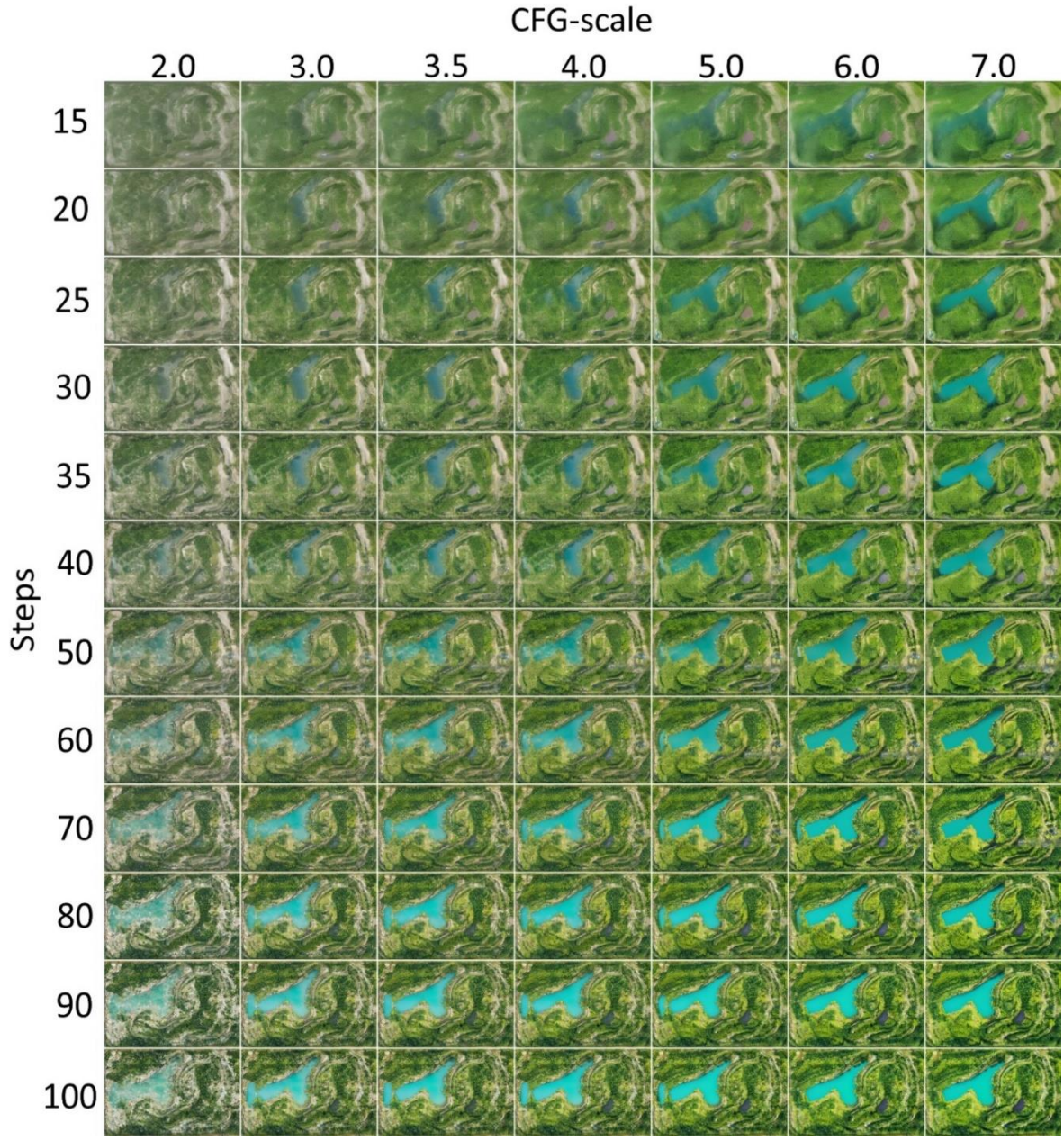

Figure 63 - Grid of images generation depending on CFG-scale [2-7] and number of steps [15-100].

Consequently, we will opt to stick with the zone of most interest on the grid would be of between 3 and 4 for the CFG-scale. As for the number of steps we need to balance between quality and speed. On the TABLE, we can see that anything over 35 steps will take more than half a minute to be generated. As a result, we will focus on the section of the grid with steps in [30, 35] and CFG-scale [3, 4].

	Steps											
	15	20	25	30	35	40	50	60	70	80	90	100
time (s)	14	18	22	26	31	36	45	53	62	70	79	87

Table 1 – Table of generation times in seconds depending on the number of steps.

Figure 64 – CFG-scale 3 Nº steps 30

Figure 67 – CFG-scale 3.5 Nº steps 30

Figure 65 – CFG-scale 4 Nº steps 30

Figure 68 – CFG-scale 3 Nº steps 35

Figure 66 – CFG-scale 3.5 Nº steps 35

Figure 69 – CFG-scale 4 Nº steps 35

On Figure 64-Figure 69 we can see that there is no great variation so, realistically we could go with any of those configurations. We decided to pick a CFG-scale of 3.5 because the water colour of the image was not as bright as in with a value of 4 but neither as pale as with a value of 3. The changes are subtle enough for it to be more of a personal preference rather than a purely objective decision.

As for the number of steps, given that there is no especially noticeable improvement between the results, we decided to go with 30 steps which also cuts off a few seconds off the generation.

To reduce the execution duration, we thought about adding an additional step to the process. It would consist in reshaping the images to fit a 512x512px shape, and we did achieve to get the time drop by half, but because of the distortion, que quality of the image was significantly worse, so this line of investigation was aborted.

Figure 70 – Image generated with Figure 72

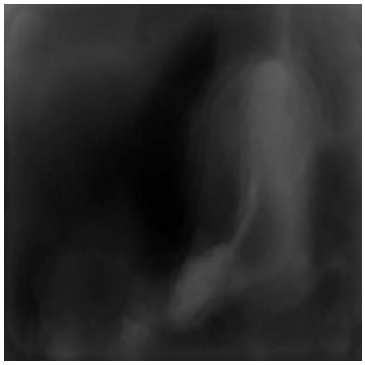

Figure 72 – Rescaled depth map

Figure 71 – Rescaled generated image

Figure 73 - Image generated without rescaling

2.1.2. Choosing the SD model

The Stable Diffusion models we tried were:

- *v1-5-pruned-emaonly.safetensors*: The base Stable Diffusion v1.5 model. It was the one that was released first and the one around which the open-source community has been most active.
- *512-base-ema.safetensors* and *768-v-ema.safetensors*: The Stable Diffusion v2.0 models. In this update, StabilityAI decided to create a model that is able to work with 786x786px images, aside from the one that allows doing so with 512x512px. Nonetheless, the qualitative jump with respect to the 1.5 version is not especially noticeable.
- *satelliteImageGenerator_mapsatimageeuV10.safetensors*: This is a Dreambooth-finetuned Stable Diffusion v1.5 version. It was downloaded because the results published online looked promising, but it didn't really fit our problem, as the depth maps, we were using don't really look like satellite imagery.

During the development of this project, SDXL was also released, which would allow us to mage images of 1024x1024 in size, and most recently SDXL Turbo was published. This last one allows generating images in a single inference step, which is an astonishing achievement. Still none of those were tested out because of lack of time.

So, in the end the configuration used as a basis for later comparison of results.

2.2. On ControlNet

Control-Net plays a crucial part in our setup as it is the part responsible for depth conditioning the image generation process. In our case, we are using the depth model without preprocessor because the depth map will be obtained directly from the physical environment with the Kinekt depth camera.

To use ControlNet as part of the Automatic1111's web-UI, we must install the corresponding extension. It can be found on Mikubill's GitHub repository under the name sd-webui-controlnet[42]. Here the version, or commit id, we download is pretty relevant because the API specifics change. Whilst working on this project we had to face this very problem, as after updating the web-UI at some point, our API code stopped working correctly.

Initially, what the extension did, was creating a side application, so that if you wanted to use conditioning for the generation, you had to send the petition to the https://127.0.0.1:8760/controlnet/txt2img URL instead of the standard https://127.0.0.1:8760/sdapi/v1/txt2img, used for generating images using just Stable Diffusion. The JSON structure was different too, with the ControlNet parameters being at the same level as the rest.

```
{
        "prompt":
        "negative_prompt":
        "controlnet_input_image": [],
        "controlnet_mask": [],
        "controlnet_module":
        "controlnet_model":
        "controlnet_weight":
        "controlnet_resize_mode": "Scale to Fit (Inner Fit)",
        "controlnet_lowvram": true,
        "controlnet_processor_res": 512,
        "controlnet_threshold_a": 64,
        "controlnet_threshold_b": 64,
        "controlnet_guidance": 1,
        "controlnet_guessmode": true, "enable_hr": false,
        "denoising_strength": 0.5, "hr_scale": 1.5,
        "hr_upscale": "Latent",
        "seed": -1,
        "subseed": -1,
        "subseed_strength":
        "sampler_index":
        "batch_size": 1,
        "n_iter": 1,
        "steps": 20,
        "cfg_scale": 7,
        "width": 512,
        […]
}
```

Code 2 – Alternative JSON section for the ControlNet parameters.

In later versions this behaviour was patched so that instead of sending the petitions to a different URL, all petitions are sent to the same one, and then, depending on the parameters the JSON carries, certain functionalities would or would not be involved in the process. If there was a ControlNet

entry inside the alwayson_scripts, then the ControlNet module would be used with the parameters present there.

To avoid possible future problems, the local version was decided not to be updated, so that no more time than what was strictly needed would be spent on figuring out what behaviours were changed in the patch and have to continuously adapt to them.

2.3. LoRA model training

To train a custom LoRA file, first what we have to install another web-UI. This time around it is Kohya's GUI which can be found on bmaltai's GitHub repository. It is a GUI implementation for the Kohya-SS Python library, which was created for finetuning stable diffusion model. It allows finetune them not only by creating LoRA files but using Dreambooth too. The GUI also includes some other additional utilities to make the training process easier, like for example a direct access to BLIP to pre-fabricate the description files for the dataset.

Once the repository is downloaded, the first time we use the tool we must run the *setup.bat* file which is responsible of installing all the environment requirements and files. After that first execution, we can launch the UI using the *gui.bat* file.

Once inside the tool, on Figure 74, we access the LoRA->Training section (1). There, on the source model tab, we choose the base Stable Diffusion model which we are going to finetune (2). Given that what we get from the LoRA creation process are smaller files, those cannot be used by themselves for generating images, and require a base Stable Diffusion model. If we were to use a LoRA file, we should also have access to the original Stable Diffusion file it was trained on.

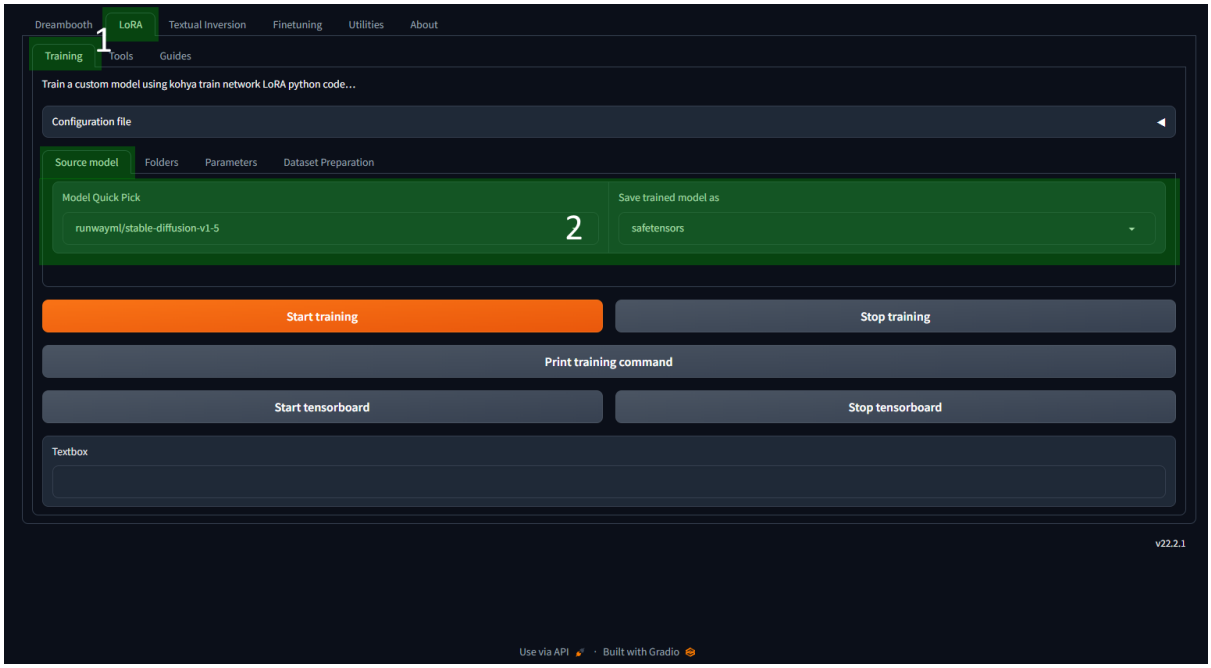

Figure 74 – LoRA training source model section inside Kohya's GUI

After that, we must configure the folder structure for the training. For that, on Figure 75, we have the Dataset Preparation (2) tab inside the LoRA->Training (1) section. There we get access to a Dreambooth/LoRA Folder preparation (3) tool. Within there, the data we have to introduce is the following:

4. Instance prompt: This is where the activation tokens for our LoRA model should go. Ideally, we want to use tokens that already make sense for the Stable Diffusion model we are finetuning, because that makes it easier to understand the concept we are trying to train. In our case, that would be a top-down view of a landscape. These images must have good variations between them, ideally only overlapping on the concept you are trying to train itself. We went with about 40 images of big size and good quality. The words we decided to use as activation token were “bird’s eye shot”.
5. Class prompt: This field should contain the broader class were the concept we are trying would fall into. In our case, it would be “landscape”.
6. Training images: Here we put the path to the directory where our training dataset is stored. This tool will take them and store in the new one that it will create for all the training process.
7. Repeats: A number depicting how many times should each image be used during the training process. From what we could find, it seems that the more you increase the number the better the result, although it also takes more time for the training to complete. In our case, we went with 20, which seemed like a good balance point.
8. Regularization images: A path to a second dataset of images, these ones depicting what you put in the class prompt field. These allow the model to differentiate between the base concept and the specific one you are trying to train. For example, if you want to train a LoRA of a character you would use a regularization dataset with people, so that the model is not overtrained. In our case we used a regularization dataset of landscapes, with 512x512px images depicting them from different angles. Ideally, these are supposed to be into the hundreds in size, but we stuck with around 50.
9. Destination training directory: a path to the directory where you want the folder structure to be created.
10. Prepare training data and Copy into Folders Tab buttons: The first one creates the directory structure with the data you filled in. The second one copies the required information into the Folders tab shown on FIGURE.

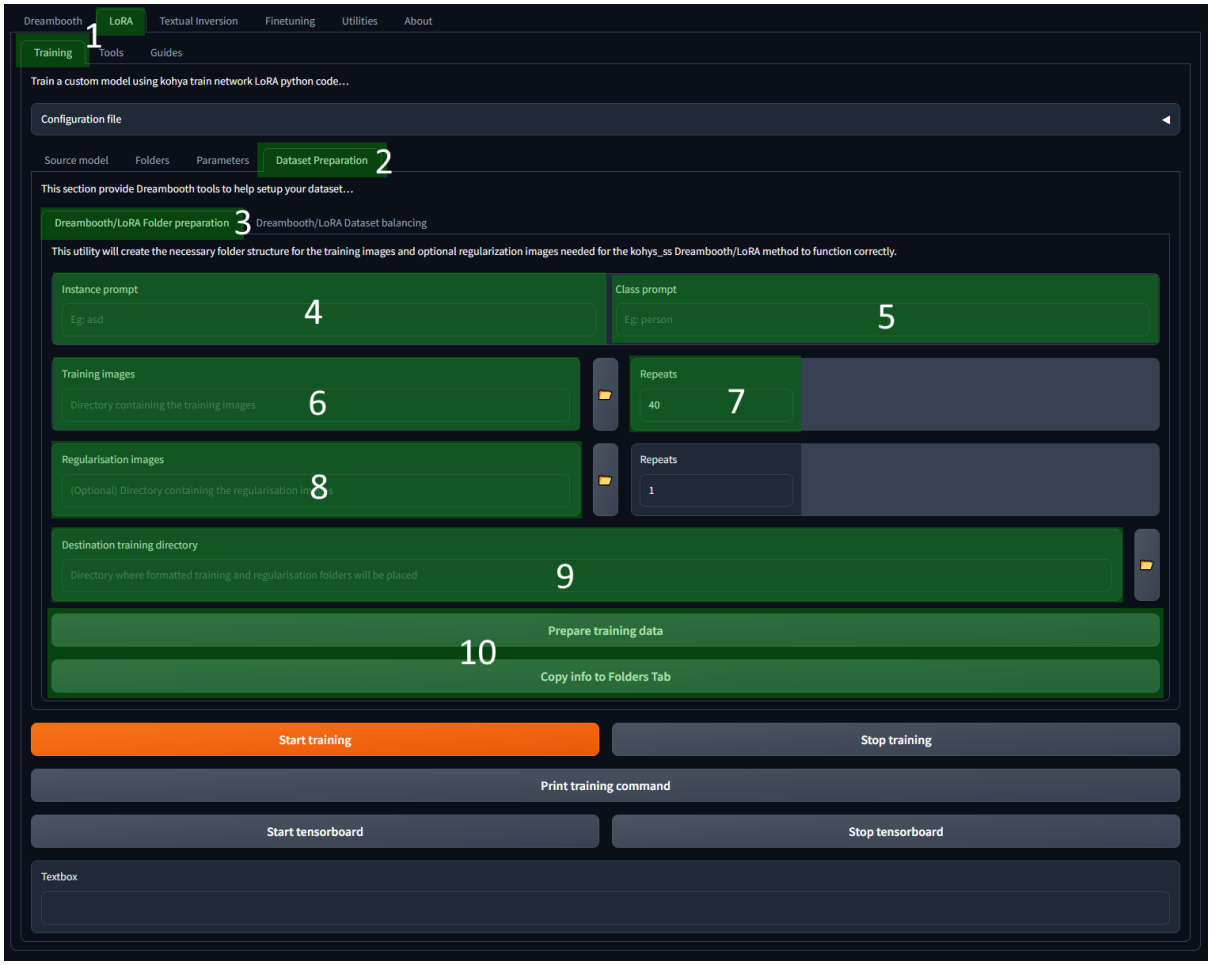

Figure 75 – Dataset preparation section inside Kohya's GUI

In case we have correctly created the folder structure ourselves, we can also manually fill the fields in the Folders tab (2) on Figure 76. Here we must input the training dataset director into the Image folder field (3), the path to the regularization dataset into the Regularization folder field (4) and the folder where we want the generated LoRA files to be stored into the Output folder field (5). There is also a Logging folder filed (6), where we can put the directory where togging files of the training process would be stored if we want to have access to them later. Finally, we have the Model output name and Training comment (7). These are used to set a name for the LoRA files we will make and, in case we want to do so, store some useful information about it, like for example, which was the activation token used to train the model.

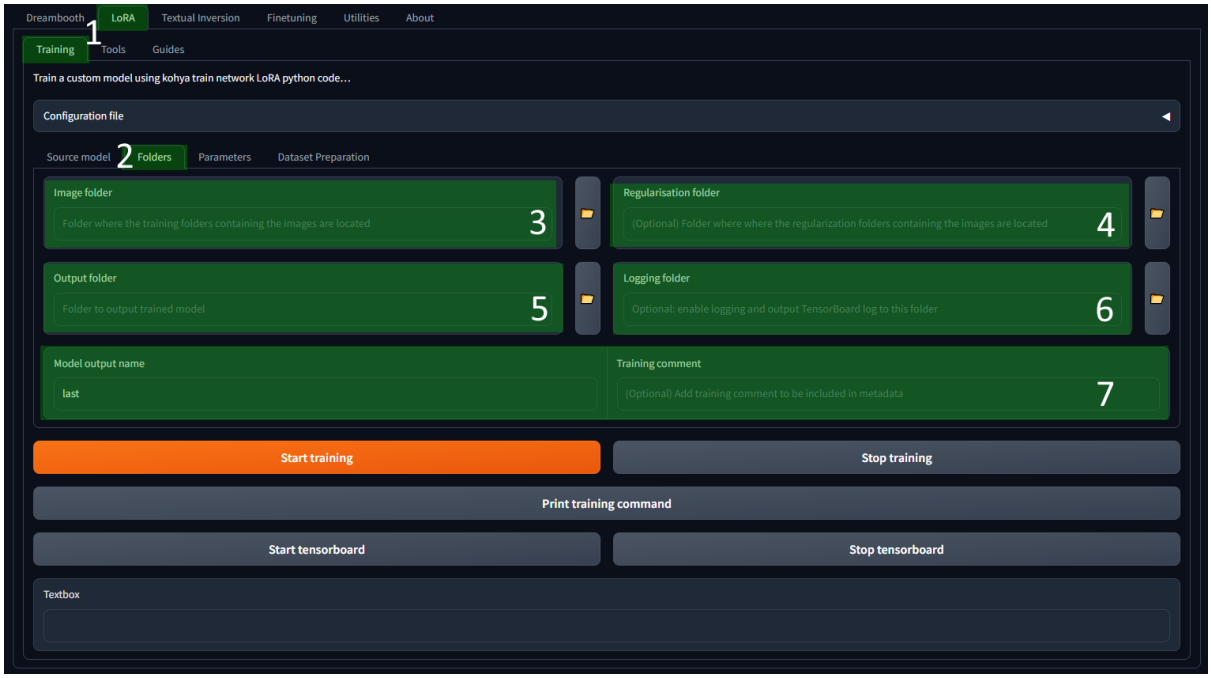

Figure 76 – Folders section inside Kohya's GUI

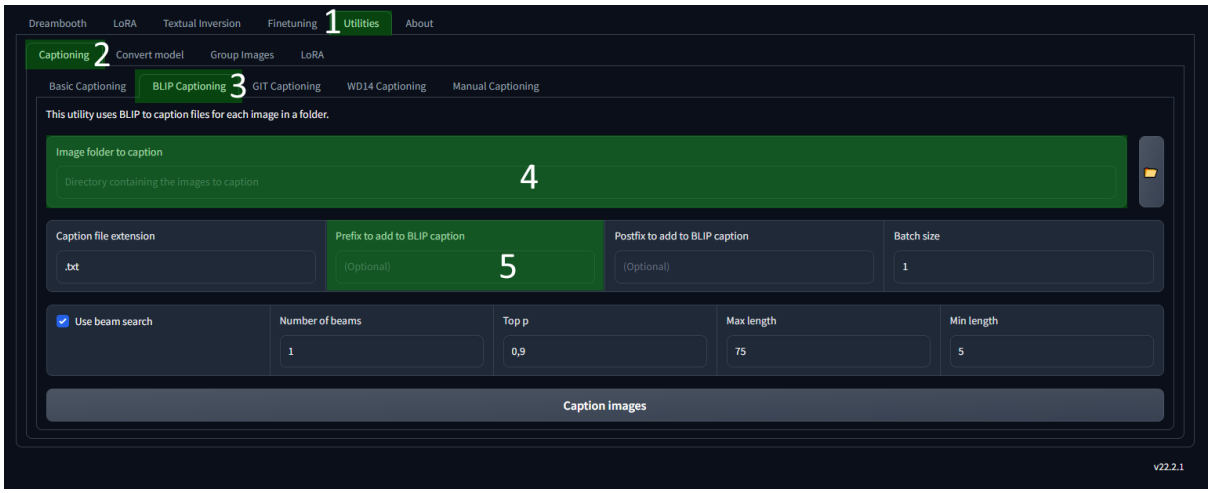

Figure 77 – BLIP captioning section inside Kohya's GUI

Now the dataset is mostly prepared, but we still need to make an additional step. This is, for each image of the training dataset, make a *.txt* file containing a description of what is being depicted on the corresponding picture. The GUI offers a way to quicken this process by using BLIP (Bootstrapping Language-Image Pre-training), which scans all the images and generates a description file for each of them. You then only need to go over those descriptions and adjust them to need, instead of building them from the ground up. This tool can be found in the Utilities section (1) on Figure 77. There we have a Captioning tab (2) and inside BLIP captioning (3). Here not much has to be done. We put the path to the training dataset into Image folder to caption filed (4) and fill the Prefix to add to BLIP caption (5), where we put the same token we will be using for the LoRA model. Sometimes instead of a prefix we want a postfix, as when styles are trained, so that the description reads "a [sth] in the style [sth]" instead of "in the style [sth], a [sth]".

Once the dataset is prepared, we have set the values for all the relevant parameters[36] for the training process. There are many more that those we are going to discuss here, but these are the once that make the most difference. We can find them in the LoRA->Training section (1) inside the Parameters tab (2) that we can see on Figure 78. The ones listed next are from the Basic section (3):

4. Train batch size: shows how many of the dataset images are used for the training at once. The more, the faster process is, but also more VRAM is required. Here we chose 5.
5. Epoch: number of iterations during training. It basically creates a new LoRA model in each iteration and allows us to determine how much should we iterate. In our case it was 10.
6. Save each N epochs: allows you to save the intermediate LoRA files created during the training process. This gives you more versions from a single training process you can compare the results of and choose the one suits you the best. We decided to save every 1.
7. Mixed precision and Save precision: Here we either choose between pf16 for a pre-3000 series NVidia graphics card and bf16 for a 3000 series or higher. This refers to a set of optimization strategies to increase efficiency and reduce memory usage for large model training and inference when using LoRA. We have a Nvidia GeForce RTX 3060, so it is bf16.
8. Cache latent and Cache latent to disk: Allows for the compressed images into latents to be saved in temporary files in main memory to save up VRAM. We checked both.
9. LR scheduler: allows changing the learning rate during the training process. We decided to keep it constant.
10. Optimizer: is a setting for how to update the neural net weights during training. We chose Adafactor which adjusts the learning rate according to the progress of learning while incorporating Adam's method. As additional parameters, these were added to prevent LR scaling “scale_parameter=False relative_step=False warmup_init=False”.
11. LR (learning rate): Here, it seems that a common value is 0.0009 and 0.0012, which determines the rate of changing of the weights in the neural network. Keeping it low presents the network from overtraining, but also slows the learning process down.
12. Max resolution: Allows changing the resolution of the trained output image of the model, so that for example instead of having to train a 1024x1024 model, if that was your base model, you can make from it a 764x764 model, which reduces the training load, although the quality also drops a bit. As out base is 512x512, there is no sense in making it smaller.
13. Enable buckets: Allows for training with uncropped images. It sorts images of the same size into buckets to then train on each of them separately. I you have not standardized the training dataset into equal shape images, is important to keep this field checked.
14. Network rank: Determines the dimension of the final file. The bigger is the file, the more VRAM is required, but also the better the results are. We set it to 128.

In Advanced settings (15), in order to save on VRAM, we can enable Gradient checkpointing. This allows for gradual updates of the weights of the neural nets for each image instead of performing a simultaneous one.

Figure 78 – Parameters section inside Kohya's GUI

Finally, we click the Start training button. This process in our setup took around 16 GB of VRAM and between one and two hours to complete. As a result, we got 10 LoRA files that we can use to compare the results.

2.3.1. Compare LoRA

Once we got the LoRA files we tried comparing the results they gave us with what would be those from the base Stable Diffusion model.

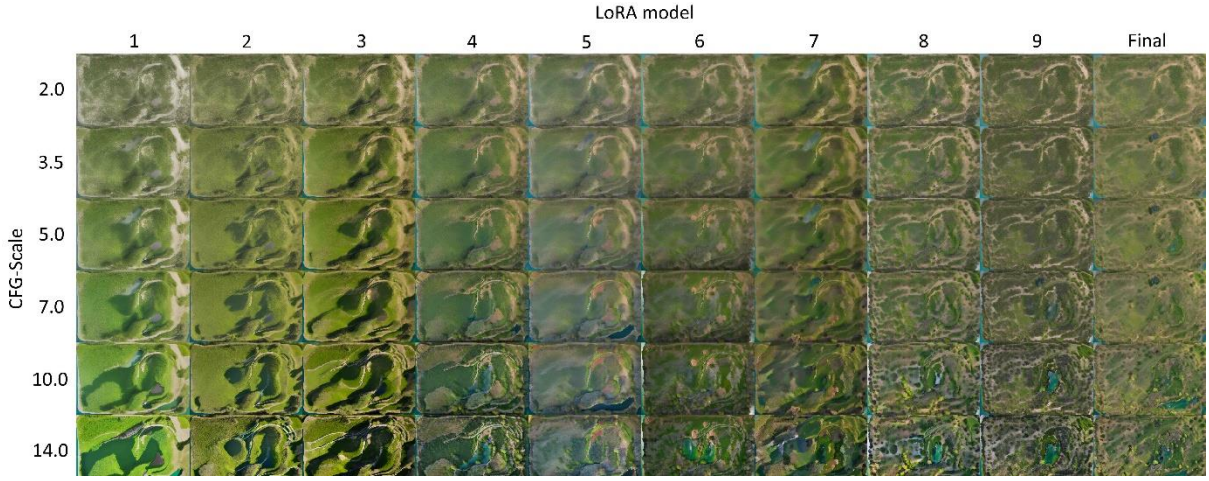

Figure 79 – Grid of images generated with the LoRA models and 30 steps

Figure 80 – Images generated with base Stable Diffusion v1.5 and 30 steps

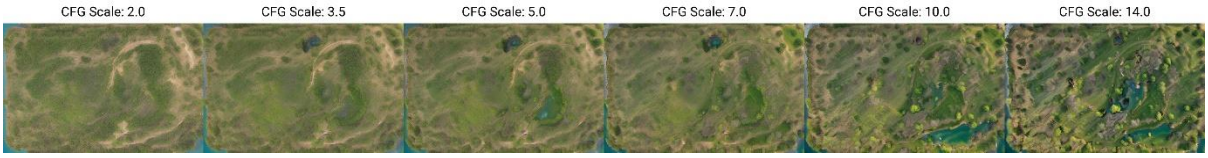

Figure 81 – Images generated with the Final LoRA model and 30 steps

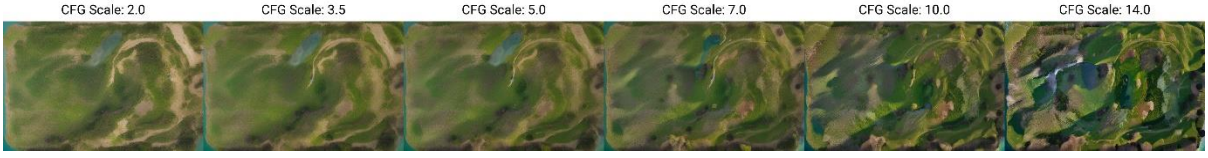

Figure 82 – Images generated with the 7th LoRA model and 30 steps

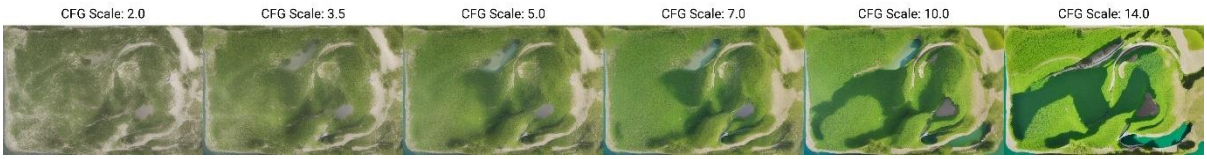

Figure 83 – Images generated with the 1st LoRA model and 30 steps

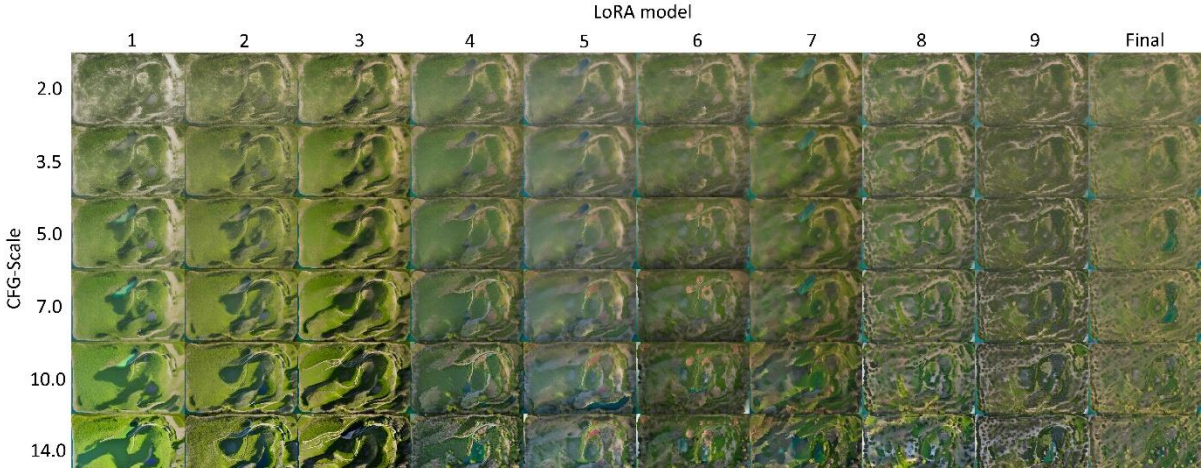

Figure 84 – Grid of images generated with the LoRA models and 40 steps

Figure 85 – Images generated with base Stable Diffusion v1.5 and 40 steps

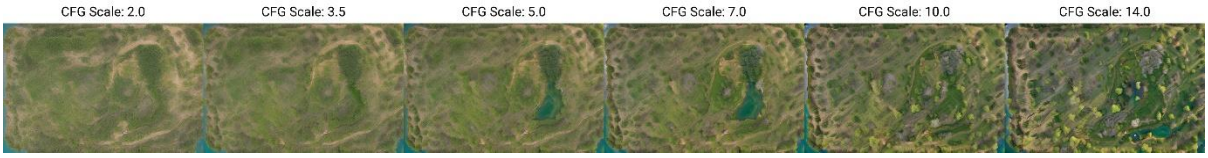

Figure 86 – Images generated with the Final LoRA model and 40 steps

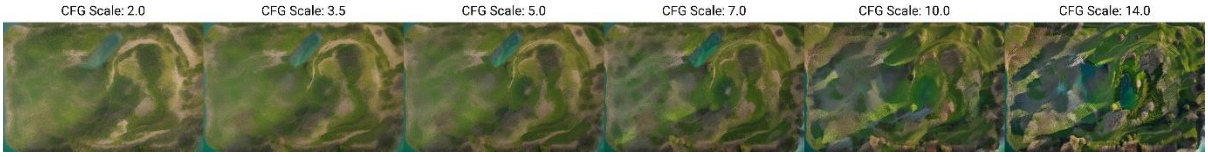

Figure 87 – Images generated with the 7th LoRA model and 40 steps

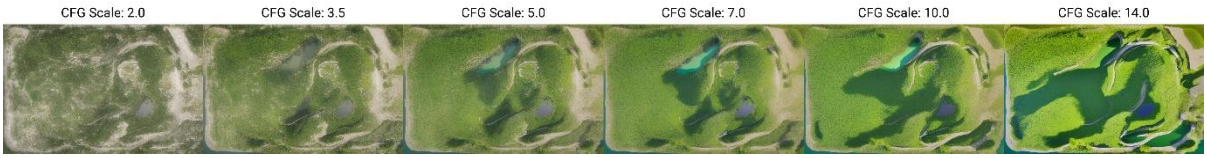

Figure 88 – Images generated with the 1st LoRA model and 40 steps

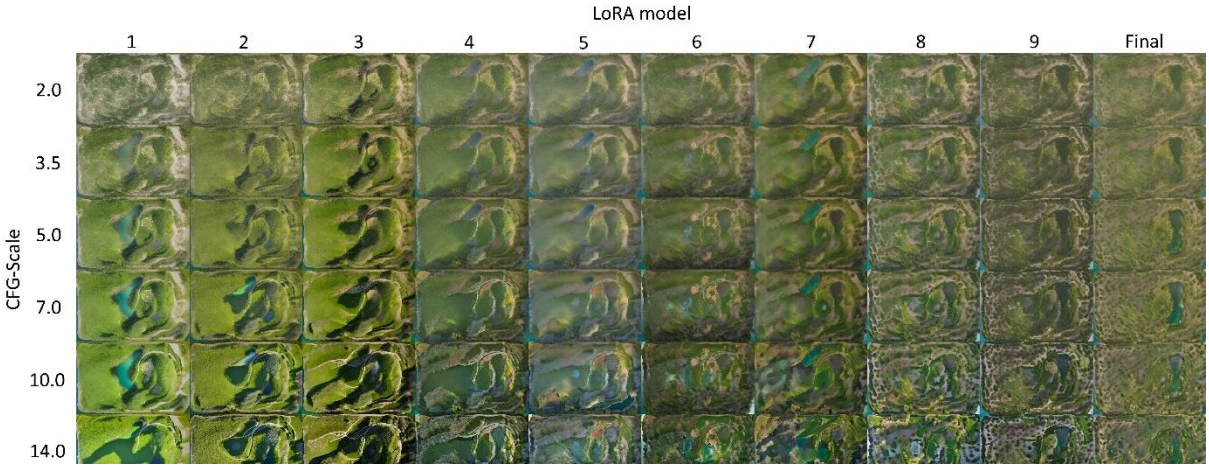

Figure 89 – Grid of images generated with the LoRA models and 50 steps

Figure 90 – Images generated with base Stable Diffusion v1.5 and 50 steps

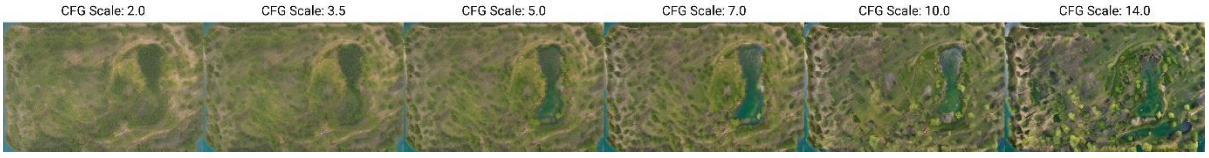

Figure 91 – Images generated with the Final LoRA model and 50 steps

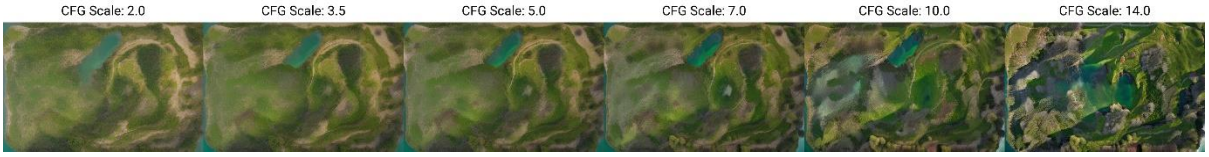

Figure 92 – Images generated with the 7th LoRA model and 50 steps

Figure 93 – Images generated with the 1st LoRA model and 50 steps

We can observe that rather that the first LoRA model is the one giving the most similar output to the original model, which makes sense considering that training a LoRA is an iterative process. The results though seem to deviate further from the prompt, as the pictures generated either don't have any lake or it is added in incorrect places. This is especially noticeable with the final LoRA file, as it places some water formation in what should be the highest point of the landscape.

We do see more water appearing when the number of steps is increased, but we get more coherent results with less steps by using Stable Diffusion v1.5 by itself. That is why further testing was done without using any LoRA models.

2.4. General Pipeline

2.4.1. Physical structure

Currently, the physical structure we can see in Figure 94 would consist of:

- a table with wheels for easy movement.
- a plastic box (3) with the right dimensions to be transported using the university elevators.
- a rigid arm used to support the Kinekt depth camera (2).
- a bendable arm to elevate the projector (1) higher up attached to the rigid one.
- A PC (4) connected currently both to the Kinekt and the projector.

A picture of the structure built is shown next, identifying the different parts of the system.

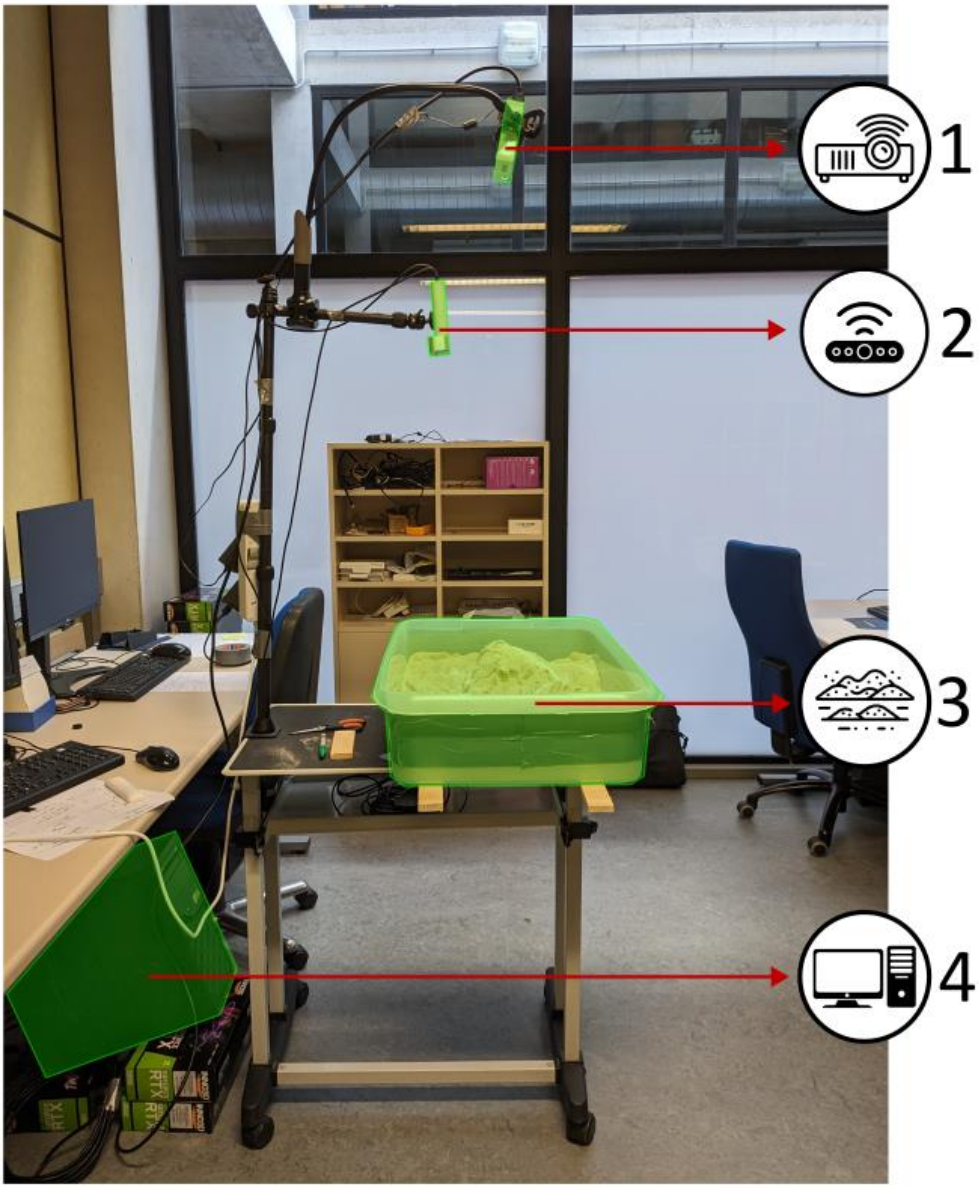

Figure 94 – Photo of the physical structure built: 1. EZCast Beam J2 projector 2. Azure Kinekt 3. Sandbox 4. PC

For the sand initially we had the idea of following the recommendation slayed out by the developer of the original sandbox idea, by using white sand, but given its lack of responsibility locally we decided to first do some test with regular darker sand. Whilst it's true that the visual clarity with white sand would most likely be slightly better, the result is good enough. A concern we had was the possibility of inconsistent refraction of the NIR light for the depth camera, but it worked fine.

On the image, we can also see some tape placed around the entire perimeter or the box. That is because during testing we found some problems with its transparency. When the sand was dragged away from the walls, some sections would be captured by the depth camera. Given that they are mostly transparent, when the light hit them, instead of correctly bouncing back to the sensor, it is lost. This means that dark spots appear on the image, because those zones are filled with 0's, which later interferes with the treatment of the image, as it messes up with the range of values, and the conditioning during the generation.

2.4.2. Data flow

We can see the original idea for the operation cycle in the Figure 95.

1. The Kinekt camera takes the depth information from the sand surface.
2. The miniPC takes the data from the Kinekt and processes it.
3. The miniPC sends the data to the PC over the network.
4. The PC uses the data to make a new image which it sends back to the miniPC over the network.
5. The minPC sends the new image to the projector.
6. The image the projector was displaying in the sand is updated.

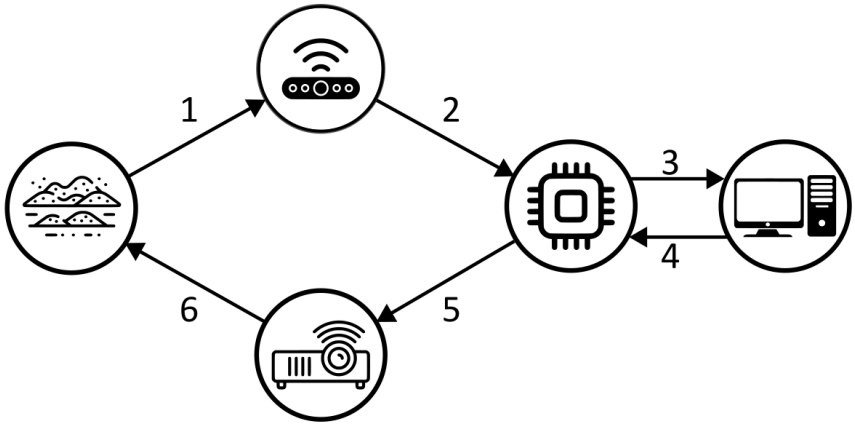

Figure 95 – Original design for the workflow

In reality, the miniPC was not made part of the final build due to lack of time, but the logic behind to establish the network connection was programmed, nonetheless. The actual implementation is shown on Figure 96.

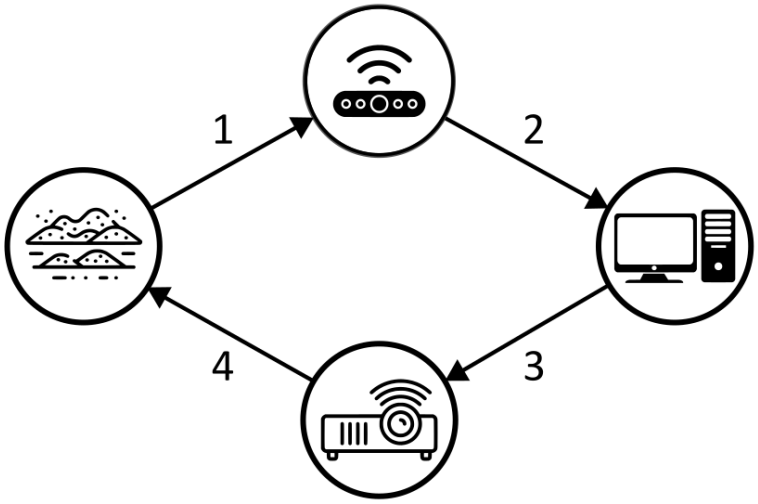

Figure 96 – Actual workflow

The steps that it takes are:

1. The Kinekt camera takes the depth information from the sand surface.
2. The PC takes the data from the Kinekt and processes it. It then uses that data to make a new image.
3. The PC sends the created image to the projector.
4. The image the projector was displaying in the sand is updated.

2.4.3. PC inner workflow

In Figure 97 we can see the processes that take place in the PC itself.

1. It gets a matrix of values from the Kinekt and the contents of a *config.txt* file located in the local storage. The matrix is a two-dimensional numpy.ndarray(), where each value represents the distance from the sensor to the surface the ray bounces of measured in millimetres. The *config.txt* file contains the prompt, negative prompt, model id, steps, CFG-scale, and seed for the generation of the next image. This data is collected by the client process running in the PC. There it is processed and prepared to be sent to the server.
2. The client process establishes a TCP connection using Python socket libraray with the server process and sends him the data.
3. The server process sends a POST petition to the web-UI server with the data needed for the generation of the image. The web-UI server processes the petition and creates a new picture.
4. The web-UI server sends the new image as a response to the POST request.
5. The server process forwards that image to the client process and closes the TCP connection.
6. The client process treats the image and sends it to the projector to be displayed.

Ideally, if we were to follow the original design idea, the PC would only have the server and web-UI server processes running on it, the first one serving basically as a wrapped for the second one. The client would be located on the miniPC and establish the TCP connection with the server through a wireless network.

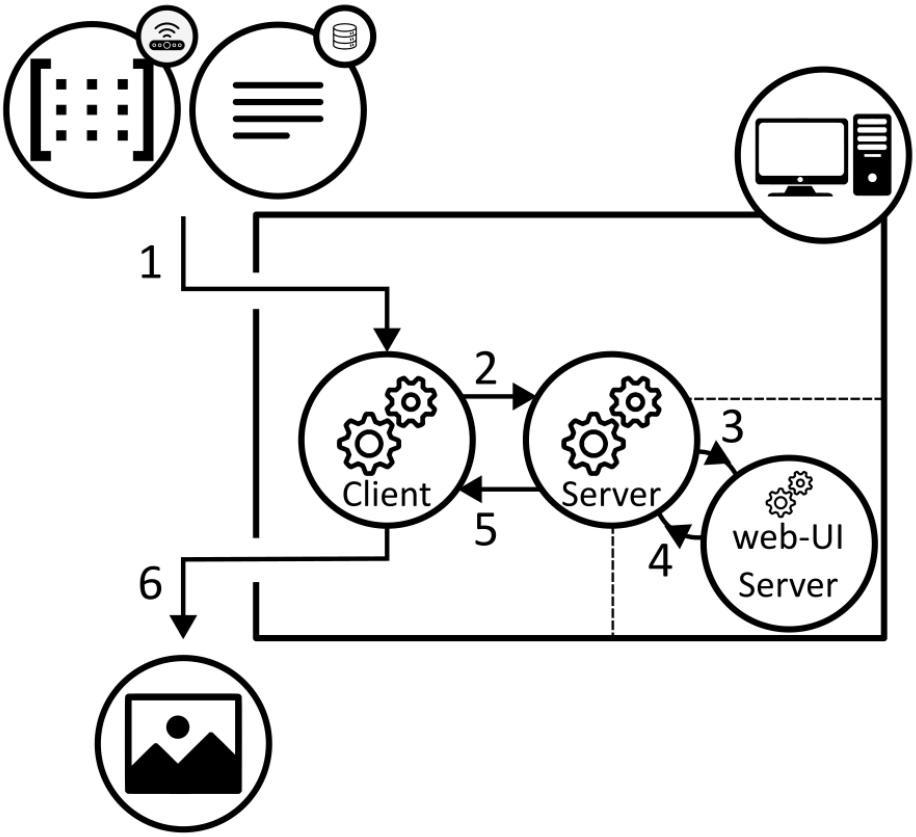

Figure 97 – Diagram of the workflow inside the PC in the current setup

2.4.4. Client process workflow

In Figure 98 we can see the processes that take place inside the client program.

1. The client takes the information from the Kinekt camera in the form of a two-dimensional numpy.ndarray(), where each value represents the distance from the sensor to the surface in front of it measured in millimetres.
2. The matrix is converted into an image using OpenCV's .colorize() method, normalizing for the range of values we are interested in which is between 50 (sand level) and 85 (table level) cm. It is then converted to a greyscale image.
3. The image is downscaled so that it shortest side matches 512px as the standard size SDv1.5 works on is 512x512px and downscaling the image, although losing some quality, makes speeds up the generation process itself. Then it is rotated and flipped over to match the projectors perspective. This last two operations could be applied after getting the generated image instead.
4. The resulting image is encoded into UTF-8 to be sent over the TCP connection.
5. The UTF-8 encoded image is encoded into base 64 so that it is storable as a string inside a JSON structure.
6. The encoded image together with the contents of the *config.txt* file read from the PC internal storage are encoded into a JSON. Storing the parameters in files allows making presets finetuned depending for the contents of the image we desire to generate.
7. The JSON encoded data is sent over the TCP connection to the server process.

8. The server process returns an image that was generated by Stable Diffusion in the web-UI server.
9. The image is upscaled to fit the projectors screen height and cantered by adding black strips to its sides.
10. The OpenCV window responsible of displaying the image on the projector replaces the old one with the new one. If there was no window a new is created.

The client process uses the projector as if it was a desktop. It keeps the OpenCV window always open and in Fullscreen mode. Observing the Code 2 we can see that it is stuck in an infinite loop, blocked by the .waitKey(0) call to the OpenCV method, and once some key has been pressed, it continues the loop, placing a new canvas in the window. This is designed to have manual control over when a new image would be displayed. We can also see how the black strips are added to prevent the image from getting distorted.

```
[…]
# Display the first image
if received_images:
        display_image_with_bars('filename')
while True:
    received_images = generate_image()

    if received_images:
            update_displayed_image('filename')

def update_displayed_image(image_path):
    # Open an image file
    img = cv2.imread(image_path)
    # Calculate the aspect ratio of the image
    img_width, img_height = img.shape[1], img.shape[0]
    aspect_ratio = img_width / img_height
    # Calculate the size of the resized image to fit the screen
    if aspect_ratio > (screen_width / screen_height):
        new_width = screen_width
        new_height = int(screen_width / aspect_ratio)
    else:
        new_height = screen_height
        new_width = int(screen_height * aspect_ratio)
    # Resize the image while maintaining the aspect ratio
    img = cv2.resize(img, (new_width, new_height))
    # Create a black canvas to place the resized image
    canvas = np.zeros((screen_height, screen_width, 3), dtype=np.uint8)
    start_x = (screen_width - new_width) // 2
    start_y = (screen_height - new_height) // 2
    canvas[start_y:start_y + new_height, start_x:start_x + new_width] =
img
    # Display the image with black bars
    cv2.imshow('Image', canvas)
    cv2.waitKey(0)

def display_image_with_bars(image_path):
    # Get the screen dimensions
    cv2.namedWindow('Image', cv2.WND_PROP_FULLSCREEN)
    cv2.setWindowProperty('Image', cv2.WND_PROP_FULLSCREEN,
cv2.WINDOW_FULLSCREEN)
    cv2.moveWindow('Image', 1680, 0) #Move window to second desktop
```

```
    update_displayed_image(image_path)
```

Code 3 – Snippet of the client process code responsible of updating the image on the projector.

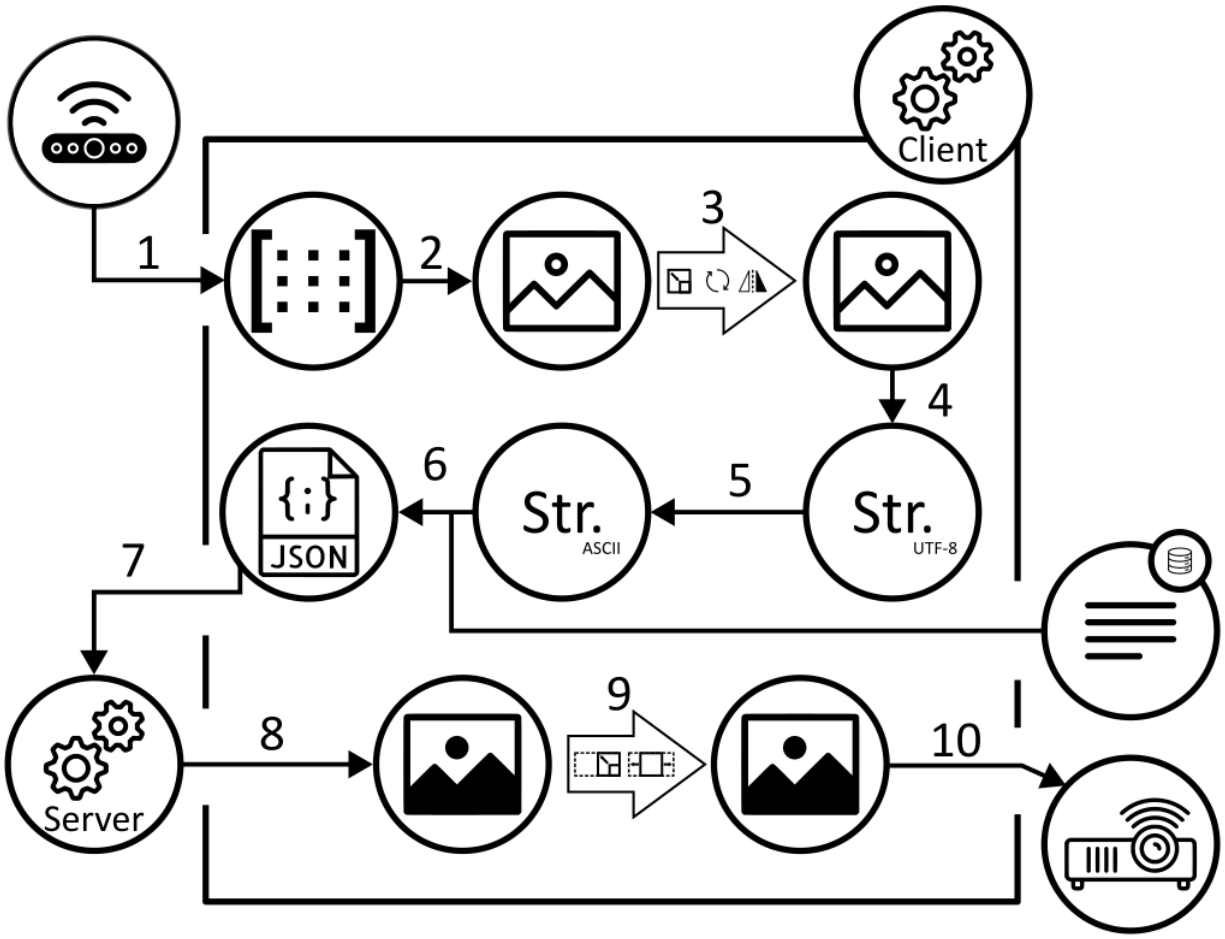

Figure 98 - Diagram of the workflow inside the client process regarding data treatment

2.4.5. Server process workflow

In the diagram on Figure 99 we can see how the server process treats the data and an approximation of how Stable Diffusion works inside of it.

1. The server process receives JSON encoded data. To do this it first has read a first message containing the length of in bits of the entire content he is expected to receive, and then reads until that number of bits are collected.
2. It sends the data inside a POST petition to the web-UI server which is listening on http://172.0.0.1:7860/sdapi/v1/txt2img to get image generations requests. There the data from the JSON is extracted, most crucially, the image on one side and the parameters on the other.
3. The parameters for Stable Diffusion are sent to the model to generate an image.
4. The model could be simply a version of Stable Diffusion or have some LoRA files influencing it, it depends on whether the activation tokens are present in the prompt.

5. The depth image on the other hand is sent to the Control-Net depth model which convolutes it down into a low dimension latent
6. The compressed image is sent to the Stable Diffusion model during the process of generating the image and is used to condition the output.
7. After iterating the required steps, the generated image is sent in a response to the POST request.
8. The server process forwards the resulting image back to the client process through the TCP connection and closes it.

The server process is constantly waiting for new TCP connections. It is when one new connection is established that the explained steps take place.

```
------------Client process code-------------
def generate_image():
    # Prepare tha image to be sent
    # Encode the image for transmission
    retval, img_bytes = cv2.imencode('.png', depth_image)
    encoded_image = base64.b64encode(img_bytes).decode('utf-8')
    # Create a socket to connect to the PC server
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
client_socket:
        client_socket.connect((PC_IP, PC_PORT))

        # Prepare payload to be sent

        # Send the message length and payload to the server
        msg_length = len(payload_json)
        client_socket.sendall(struct.pack("I", msg_length))  # Pack and
send message length
        client_socket.sendall(payload_json)  # Send the payload

        # Receive, decode and save data

------------Server process code-------------
# Create a socket to listen for incoming connections
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind((PC_IP, PC_PORT))
    server_socket.listen()

    while True:
        # Accept incoming connections
        connection, address = server_socket.accept()

        # Receive message length
        msg_length = connection.recv(4)
        if not msg_length:
            break
        msg_length = struct.unpack("I", msg_length)[0]

        # Receive JSON data from the client based on the message length
        data = b""
        while len(data) < msg_length:
            packet = connection.recv(msg_length - len(data))
            if not packet:
                break
            data += packet
```

```
        # Decode and use data
        # Send response_payload to the client
        response_payload_json = json.dumps(response_payload).encode()
        connection.sendall(struct.pack("I", len(response_payload_json)))
        connection.sendall(response_payload_json)

        connection.close()
```

Code 4 – Code snipped responsible for the communication between the client process and the server process.

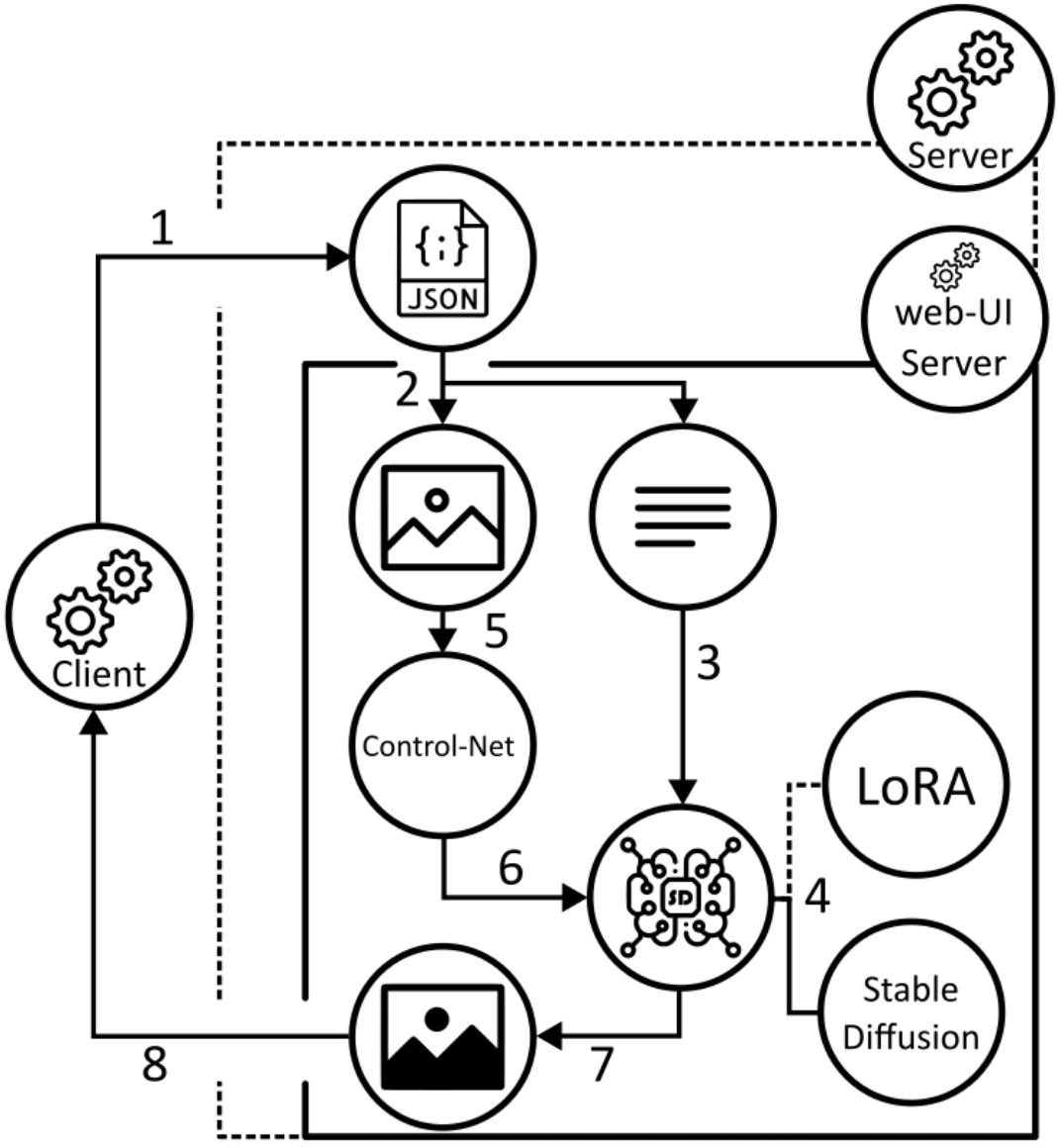

Figure 99 - Diagram of the workflow inside the server processes regarding data

Aside from the described workflow, an extra petition can be made by the server process, but this time around to the http://172.0.0.1:7860/sdapi/v1/pnginfo URL. That will return us information on how the image was generated. This mainly means recovering the generation parameters. The usefulness comes when you have used a random seed in the last generation, and want to recover its value, because liked the result obtained.

2.5. Azure Kinekt

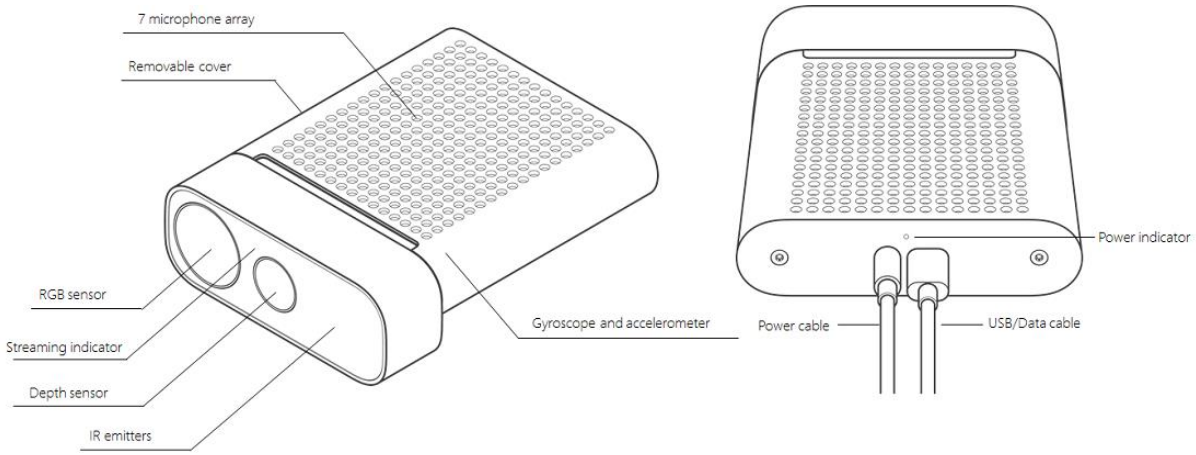

Figure 100 – Diagram of the Azure Kinekt side and back view

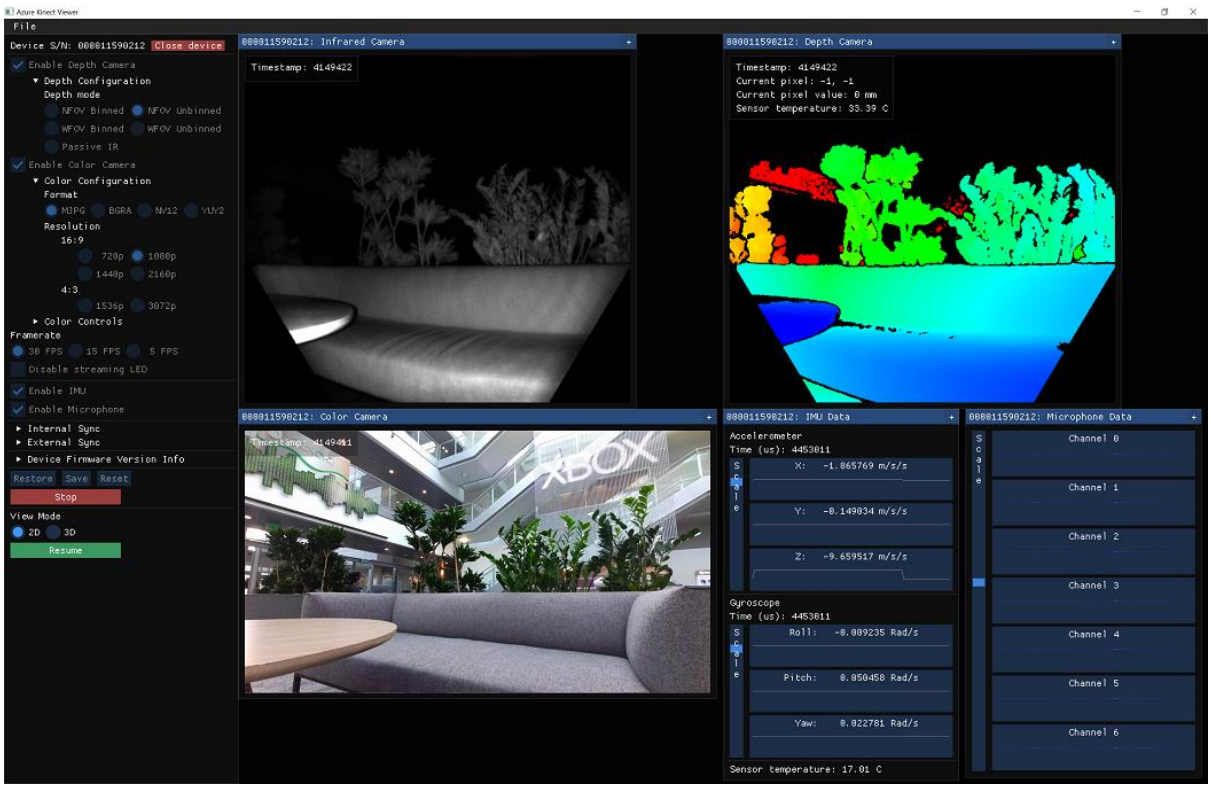

Figure 101 – Azure Kinekt Viewer GUI were we can find the output of its sensors

The Azure Kinekt[41] camera has multiple sensors:

- Acceleration
- Gyroscope
- IR camera
- RGB camera

- Microphone
- Temperature sensor
- Depth camera

We are interested on the latter. It works by shining modulated NIR (near-infra red) light onto the scene continuously, and then takes an indirect measurement of the time it takes the light to travel from the camera to the scene and back.

This same light is used for the IR camera, which displays a simple IR image. These measurements are processed to generate a depth map. The client to connect with the camera uses pyk4a, which is a Python wrapper library over the original Azure Kinekt SDK. The depth information is captured into a 2D numpy.ndarray(), where each value shows the distance from the sensor to the surface n millimetres. Given that the values represent distance, we actually want to invert the image generated from the matrix, because ControlNet depth model considers the brighter zones to be closer to the viewer, and thus have a greater value.

```
def capture_kinect_image():
    k4a = PyK4A(
        Config(
            color_resolution=pyk4a.ColorResolution.OFF,
            depth_mode=pyk4a.DepthMode.NFOV_UNBINNED,
            synchronized_images_only=False,
        )
    )
    k4a.start()

    capture = k4a.get_capture()
    k4a.stop()

    if np.any(capture.depth):
        depth_image = capture.depth
        return depth_image

[…]

# Capture the Kinect image
depth_image = capture_kinect_image()
scale_factor = max(512 / depth_image.shape[1], 512 /
depth_image.shape[0])
depth_image = cv2.resize(depth_image,
                         (int(depth_image.shape[1] * scale_factor),
int(depth_image.shape[0] * scale_factor)))
print(depth_image.shape)
# Colorize the image
depth_image = colorize(depth_image, (500, 850), cv2.COLORMAP_BONE)
# Put image in grayscale
depth_image = cv2.cvtColor(depth_image, cv2.COLOR_BGR2GRAY)
# Invert the image colors
depth_image = cv2.bitwise_not(depth_image)
```

Code 5 – Code used to extract a depth image from the

The depth camera also has two working configurations:

- NFOV (narrow field of view): It is used when the X and Y range of the image is relatively small, and we want a bigger range on the Z dimension.
- WFOV (wide field of view): This mode allows for a wider X-Y range at the expense of the Z dimension.

For this project, the NFOV configuration was employed, as there was no requirement for a broader XY extension. Additionally, the WFOV resulted in a distorted image (fish-eye view) of the box, necessitating additional transformations to be bone on the image before proceeding to the generation.

2.6. EZCast Beam J2

Figure 102 – EZCast Beam J2

Ideally the projector and would work as the main display of the computer executing the client-side code, but given that that finally wasn't implemented, it is used as a second screen for the PC. To take advantage of the full size of the sandbox, the projector must be higher than the depth camera. The code creates an OpenCV window with the image displayed, sets it as full-screen, and moves it to the second desktop. That window is kept continuously opened, only updating the canvas displayed when a new image is produced. This allows always keeping a projection on the sand.

In our current setup it is connected via an HDMI cable. It also offers a wireless connection option, but in our testing that one had an unreliable performance, forcing us to reset it sometimes because of the image getting frozen.

2.7. Virtual test setup

For testing purposes and before the physical structure was built, a virtual simulated environment was created with blender. There a box model was made and for the sand we used a simple textured surface which could be deformed as a mesh.

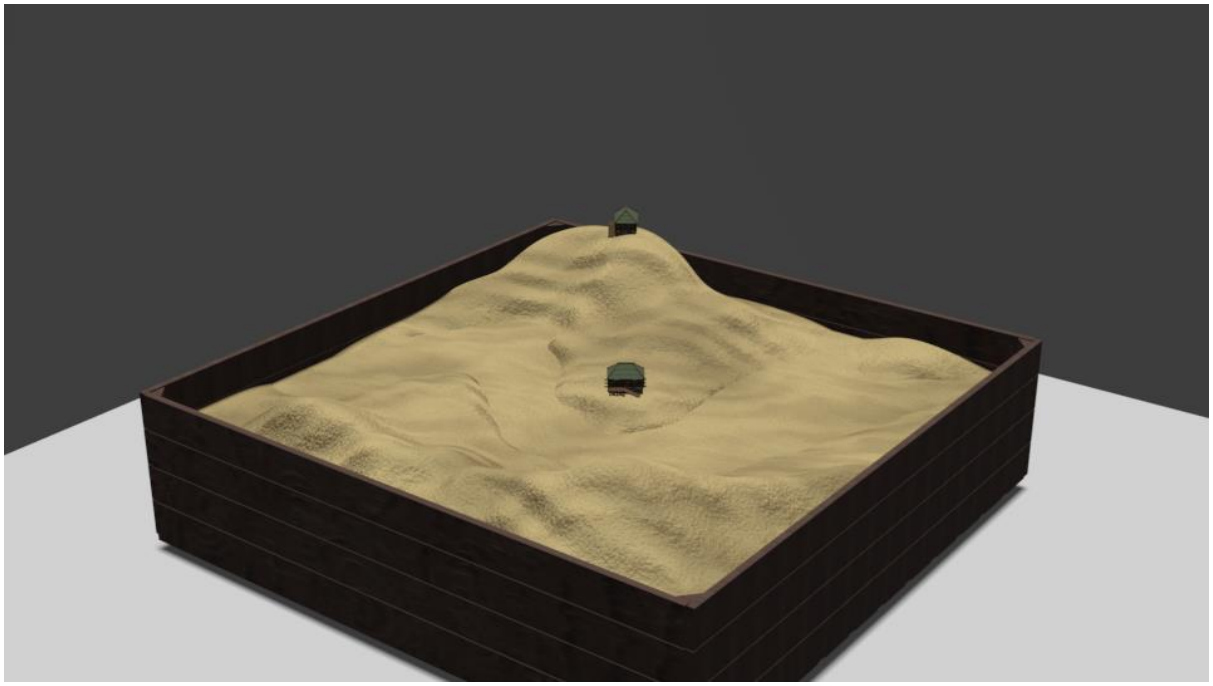

Figure 103 – 3D render of the sandbox model before projection

The render camera was placed facing downwards over the box. The camera gives us two types of information, the image seen and the alpha values representing the distance of each point to the camera. Using the compositing tool, we can normalize these values to be between 0 and 1 which allows to map the values to a white-to-black colour range. Additionally, we can add intermediate grey values to accentuate some shapes over others, or simply adjust the result to one's needs. Now, by setting it as the main image and sending it to the view and composition, the camera render will work as a depth sensor that allows us to obtain the shape of our composition.

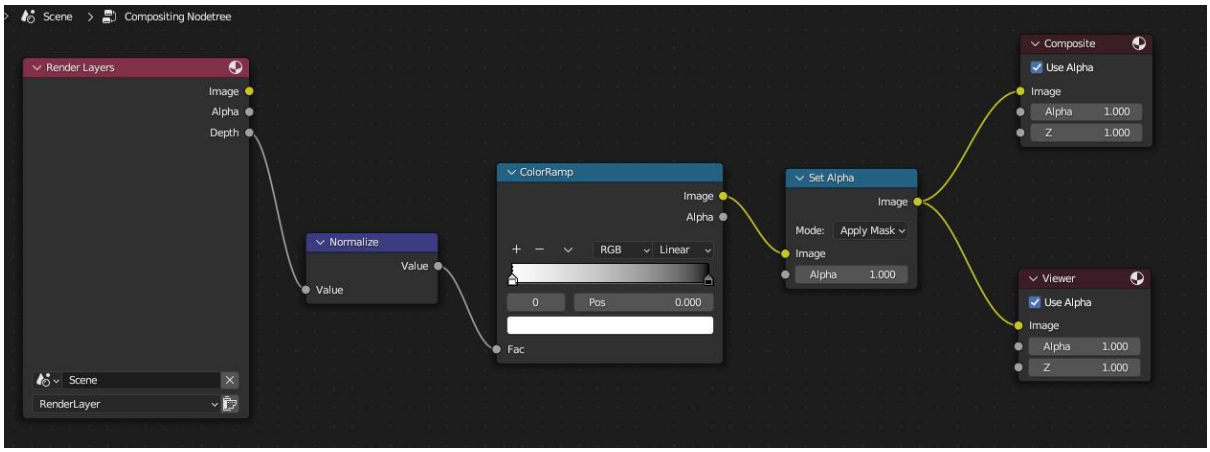

Figure 104 – Compositing node configuration for the camera

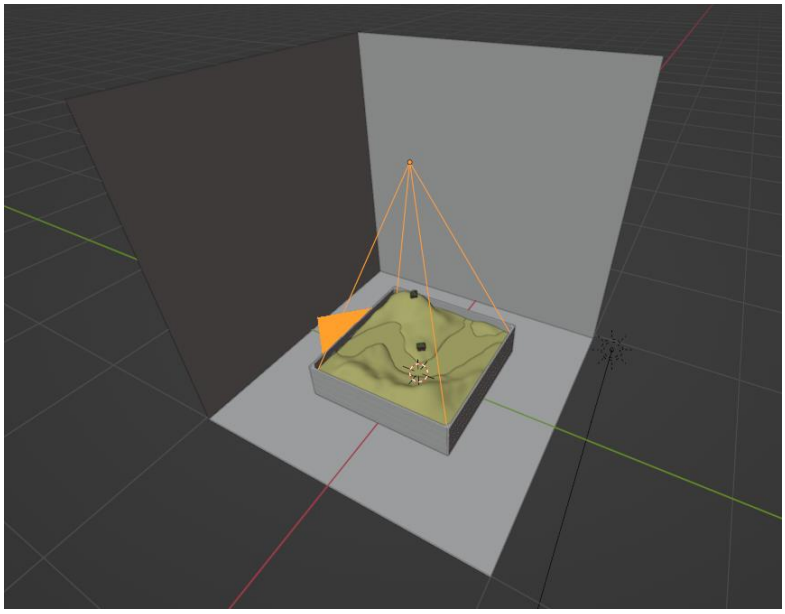

Figure 105 – 3D setup, with the camera highlighted in orange.

The rendered image was then fed to the software processes and the result mapped back onto the surface.

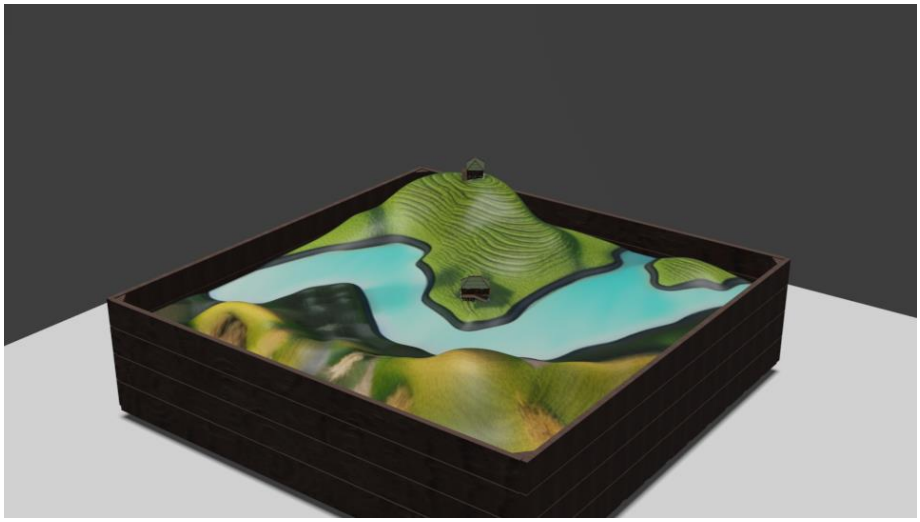

Figure 106 - 3D render of the sandbox after adding projection.

3. Experiments

In this section we will explore the results obtained with various parameters and conditioning combinations, making some comparisons between them and drawing conclusions from that.

3.1. Landscapes

The common parameters for all the following images, unless stated otherwise, are:

- sd_model_hash=cc6cb27103 (*v1-5-pruned-emaonly.safetensors*)
- sampler_name=Euler a
- batch_size=1
- steps=30
- cfg_scale=3.5

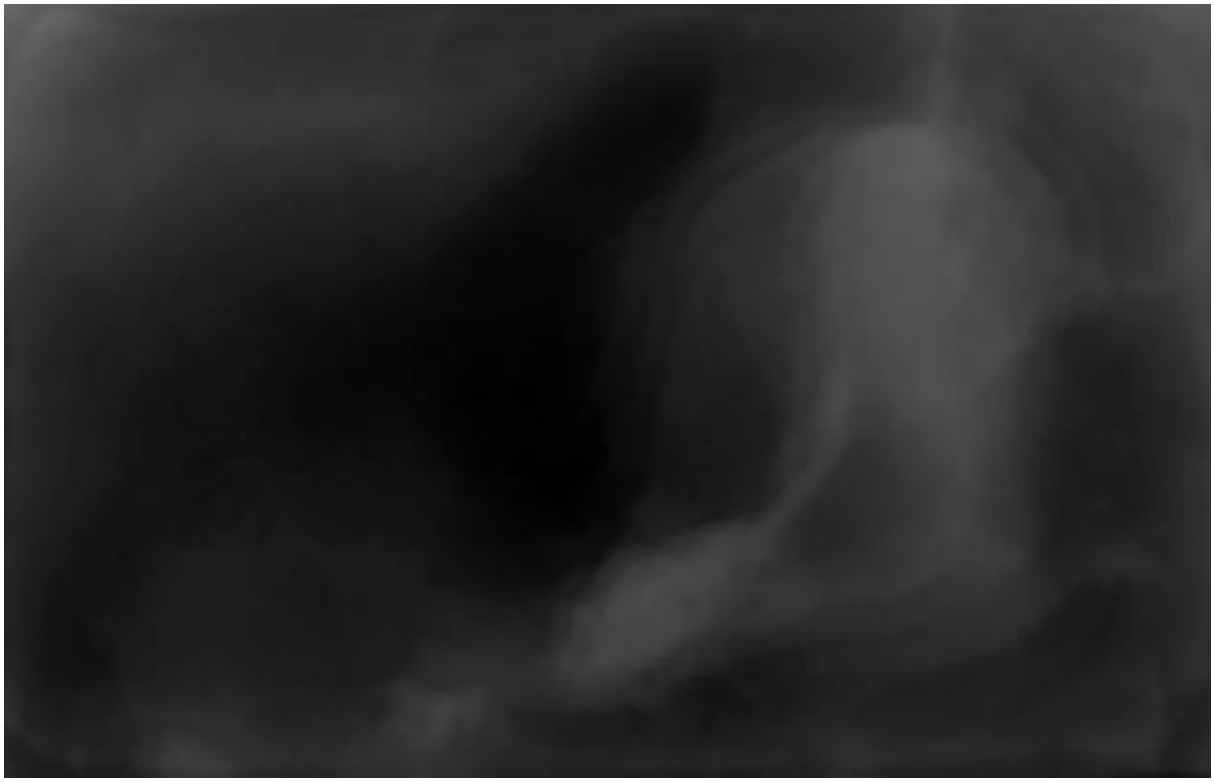

Figure 107 – Depth map used for the generation.

Green landscape with lake:

- prompt=bird's eye view of a landscape with green hills, a few trees, and a lake in the middle with crystal clear blue water, realistic
- negative_prompt=clouds
- seed=3108541024

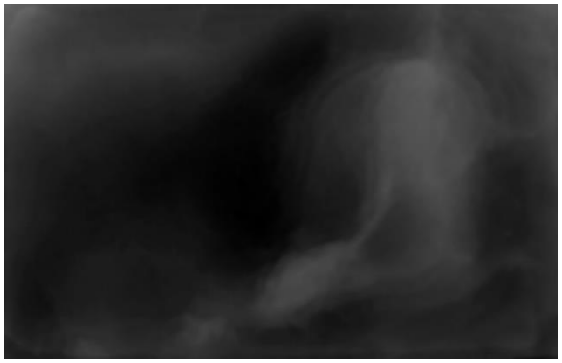

Figure 108 – Image used for depth conditioning.

Figure 110 – Generated image

Figure 109 – Top view of the sandbox with the projection

Figure 111 – Side view of the sandbox with the projection

Red desertic landscape with vegetation:

- prompt=top-down view of a desert landscape with red sandy hills, and an oasis lake in the middle with blue waters, realistic
- negative_prompt=clouds

Figure 112 - Generated image

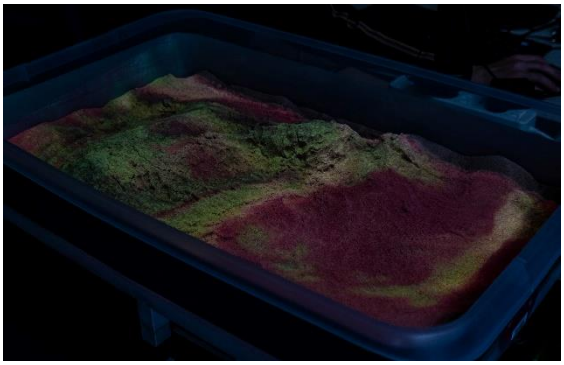

Figure 113 – Side view of the sandbox with the projection

Snowy forest with lake:

- prompt=top-down view of a snowy landscape with white snowy hills, pinecone trees, and a frozen lake in the middle with deep blue waters, realistic
- negative_prompt=clouds

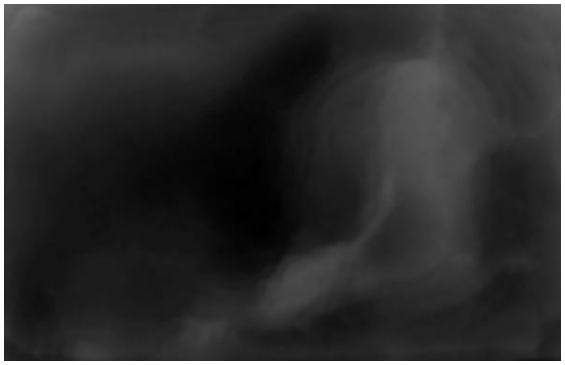

Figure 114 – Image used for depth conditioning.

Figure 116 Figure 117 – Generated image

Figure 115 – Top view of the sandbox with the projection

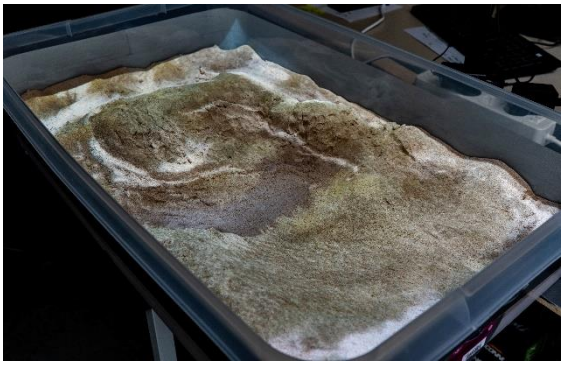

Figure 118 – Side view of the sandbox with the projection

Martian surface:

- prompt=top-down view of a Martian landscape with hills and craters, realistic
- negative_prompt=blurry

Something worth noting here, is that the results obtained look more desert-like that if you simply asked for a desert to be made. This looks like one loophole that can be abused when a sandy landscape is wanted to be made.

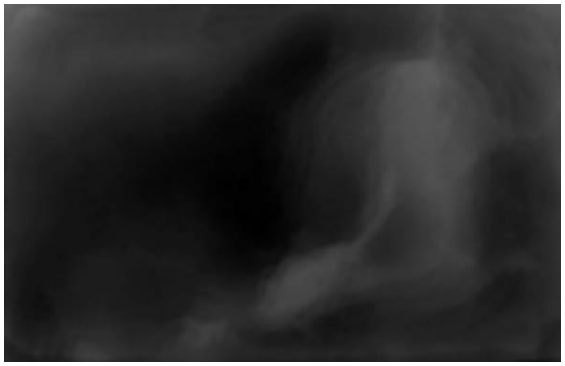

Figure 119 – Image used for depth conditioning.

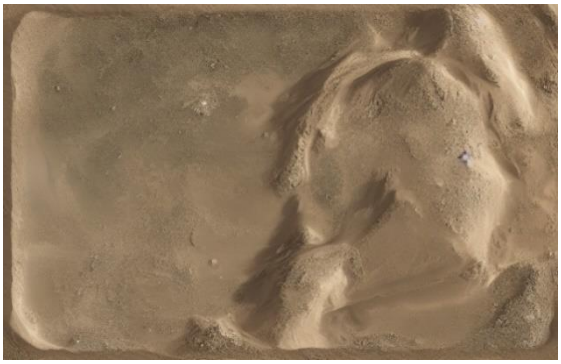

Figure 120 – Generated image

Figure 121 – Top view of the sandbox with the projection

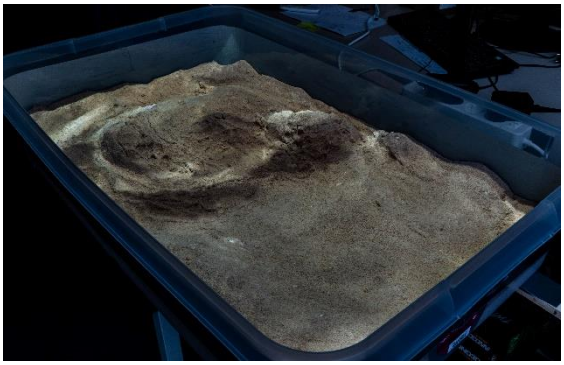

Figure 122 – Side view of the sandbox with the projection

Exaggerated mars surface:

- prompt=top-down view of a red Martian landscape with hills and craters, vibrant colors
- negative_prompt=blurry, green

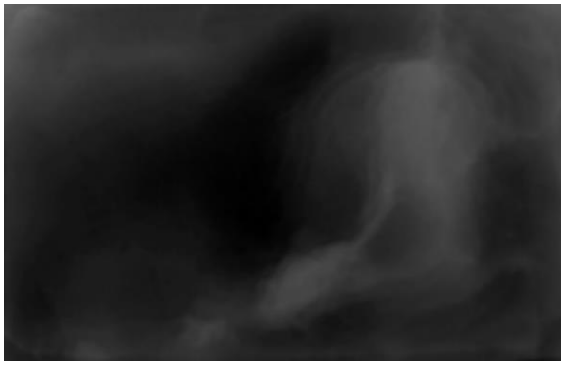

Figure 123 – Image used for depth conditioning.

Figure 125 – Generated image

Figure 124 – Top view of the sandbox with the projection

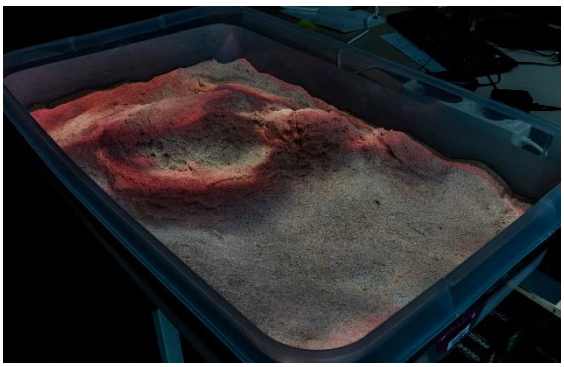

Figure 126 – Side view of the sandbox with the projection

Cartoony landscape:

- prompt=top-down view of a cartoony landscape with hills and craters, cartoony
- negative_prompt=blurry, green

This was an experimentation prompt to see how it would be understood by the model. It seemed like if it tried to make some kind of topographic map over the landscape. It is not precise but is interesting nonetheless.

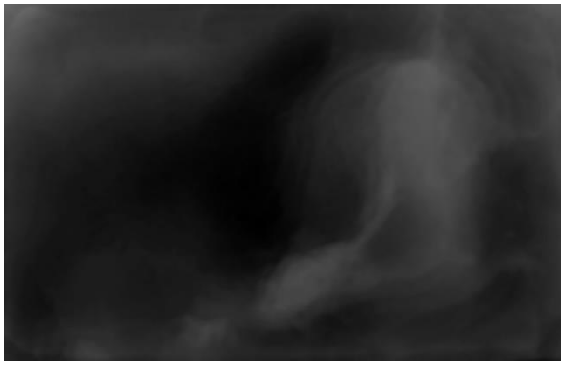

Figure 127 – Image used for depth conditioning.

Figure 129 – Generated image

Figure 128 – Top view of the sandbox with the projection

Figure 130 – Side view of the sandbox with the projection

Snowy mountain peak:

- prompt=top-down view of a mountain landscape, mountain top realistic
- negative_prompt=blurry, green

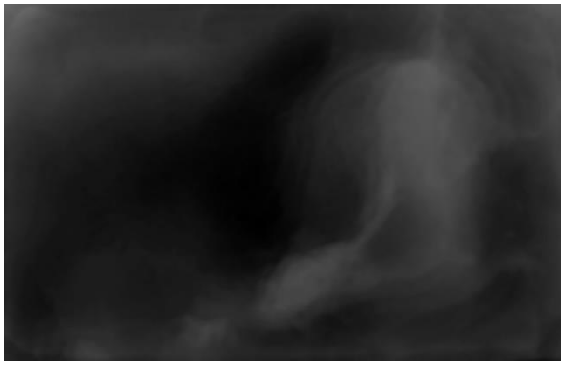

Figure 131 – Image used for depth conditioning.

Figure 132 – Generated image

Figure 133 – Top view of the sandbox with the projection

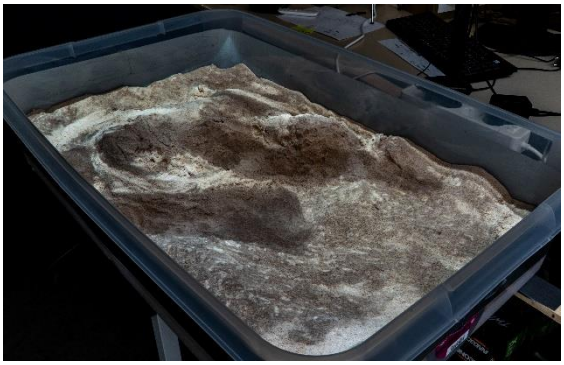

Figure 134 – Side view of the sandbox with the projection

Tropical beach:

- prompt=top-down view of Caribbean beach, seashore, crystal clear waters, realistic
- negative_prompt=blurry, green

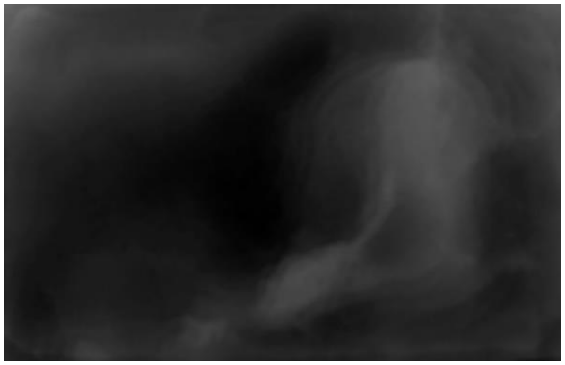

Figure 135 – Image used for depth conditioning.

Figure 137 – Generated image

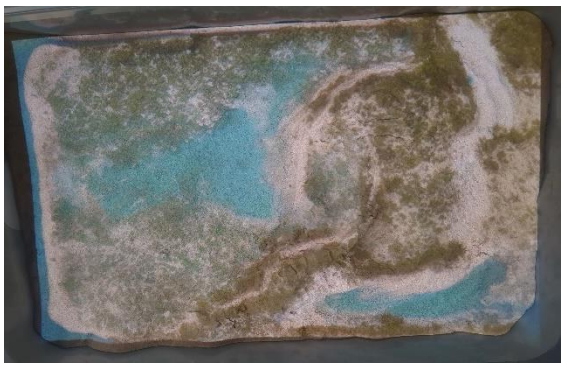

Figure 136 – Top view of the sandbox with the projection

Figure 138 – Side view of the sandbox with the projection

- prompt=top-down view of Caribbean beach, seashore, crystal clear waters, realistic, vibrant colours
- negative_prompt=blurry

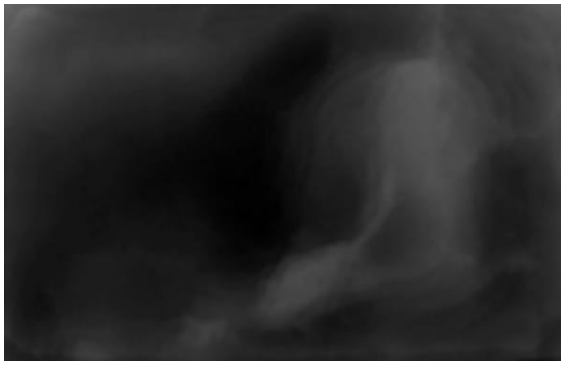

Figure 139 – Image used for depth conditioning.

Figure 141 – Generated image

Figure 140 – Top view of the sandbox with the projection

Figure 142 – Side view of the sandbox with the projection

In these two results we can appreciate the influence of the negative prompt on the final output, how the removal of "green" affects the final landscape, making the vegetation look paler.

Surreal landscape:

- prompt=top-down view of purple fantasy landscape, purple trees, a lake with crystal clear waters, realistic, vibrant colours
- negative_prompt=blurry, green, green trees

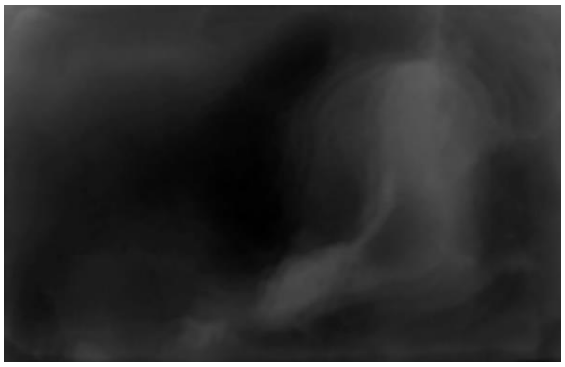

Figure 143 – Image used for depth conditioning.

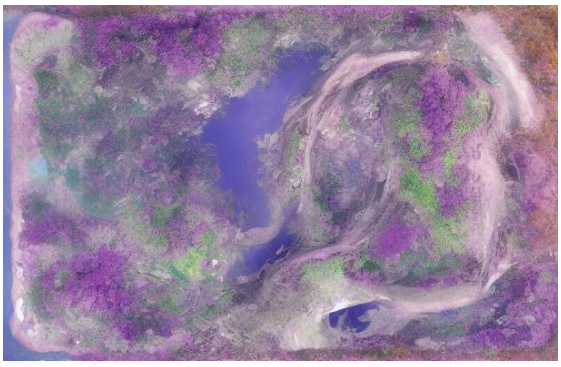

Figure 144 – Generated image

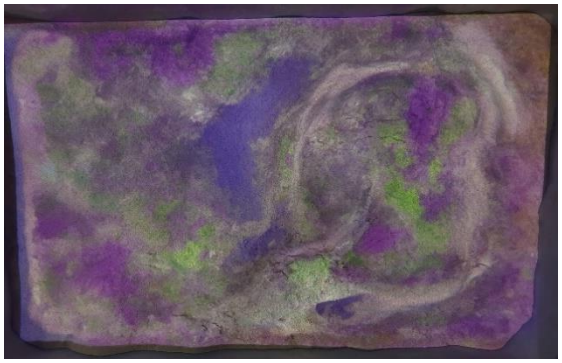

Figure 145 – Top view of the sandbox with the projection

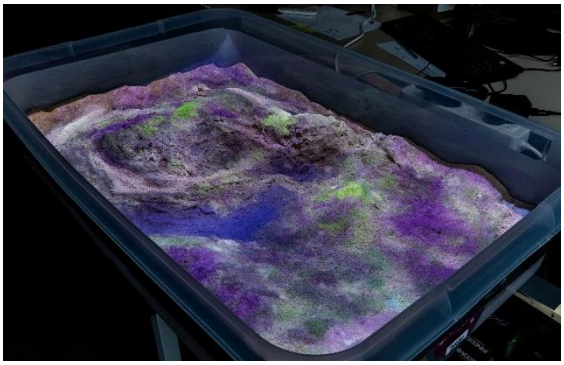

Figure 146 – Side view of the sandbox with the projection

- prompt=top-down view of rainbow fantasy landscape, purple trees, a lake with crystal clear waters, realistic, vibrant colours
- negative_prompt=blurry, green, green trees

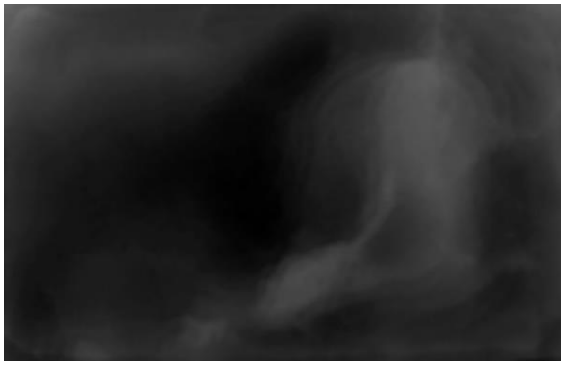

Figure 147 – Image used for depth conditioning.

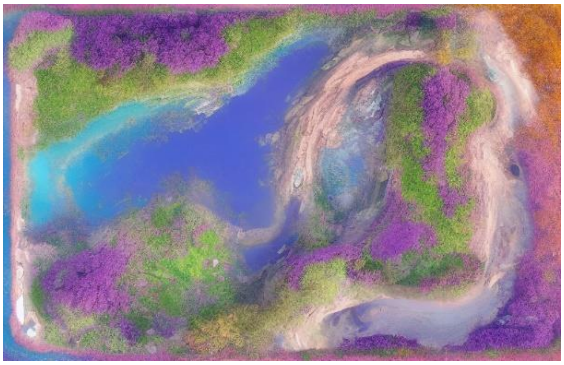

Figure 149 – Generated image

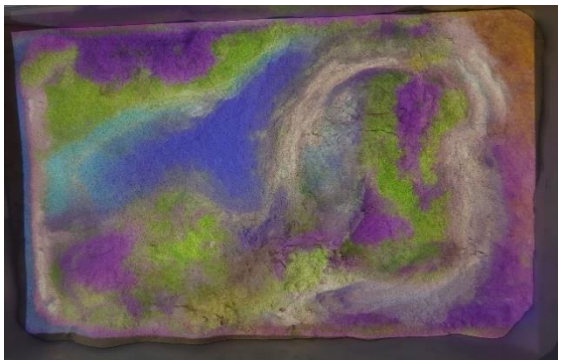

Figure 148 – Top view of the sandbox with the projection

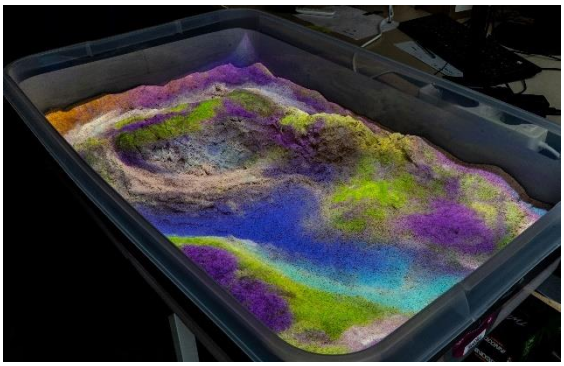

Figure 150 – Side view of the sandbox with the projection

Here we observe how the contents of the prompt overpower the negative prompt, as adding “rainbow” to it makes appearing many green patches over the canvas, despite we asking fro them not to be there.

Seafloor:

- prompt=top-down view of sea floor landscape, with a large coral formation and a school of fish swimming above it, realistic
- negative_prompt=blurry, green

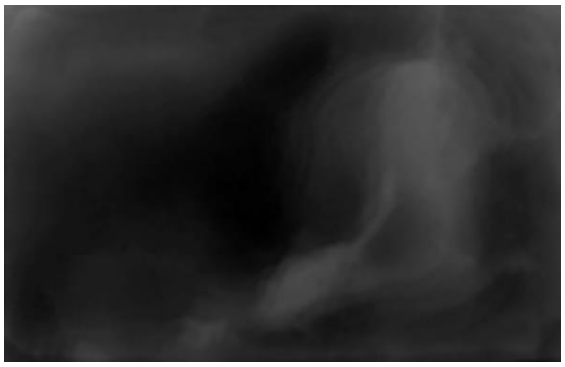

Figure 151 – Image used for depth conditioning.

Figure 153 – Generated image

Figure 152 – Top view of the sandbox with the projection

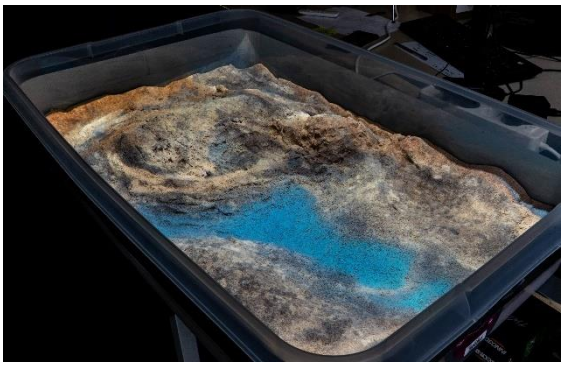

Figure 154 – Side view of the sandbox with the projection

We can observe how all the soil has been bleached out and the patches of what was vegetation earlier now look like coral riffs. It still makes a lake which doesn't make much sense underwater, but it was a tricky prompt nonetheless.

Volcano:

- prompt=top-down view of a landscape, with a large volcano crater and a lot of flowing incandescent lava rivers, black ground, realistic, vibrant colors
- negative_prompt=blurry, green
- seed=3108541024

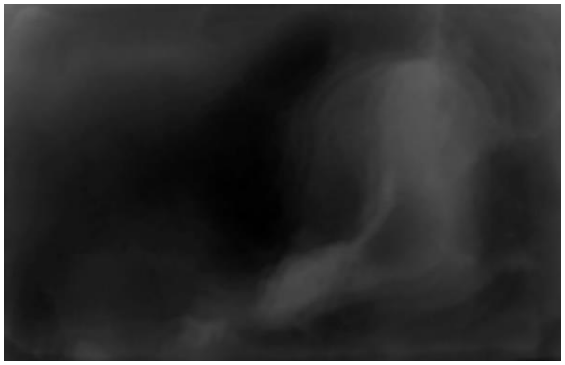

Figure 155 – Image used for depth conditioning.

Figure 157 – Generated image

Figure 156 – Top view of the sandbox with the projection

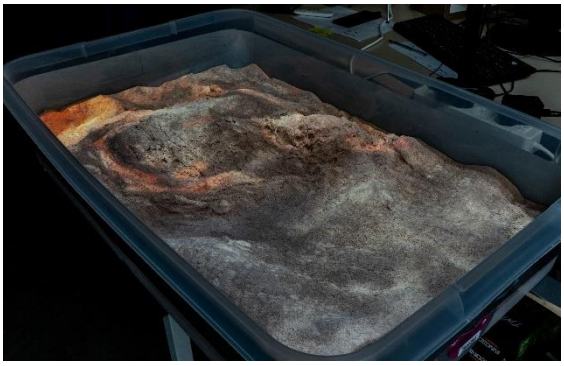

Figure 158 – Side view of the sandbox with the projection

- prompt=top-down view of a landscape, with a large volcano crater and a lot of flowing incandescent lava rivers, black ground, realistic, vibrant colors
- negative_prompt=blurry, green
- seed=4074514245

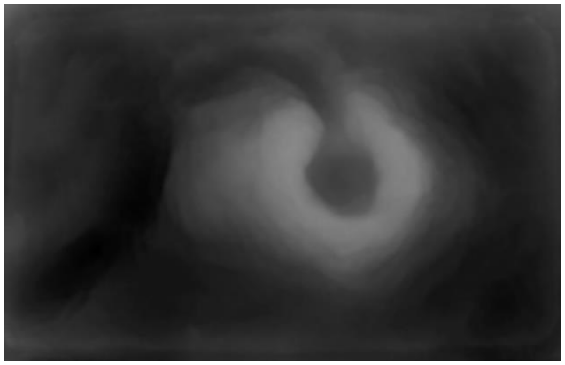

Figure 159 – Image used for depth conditioning.

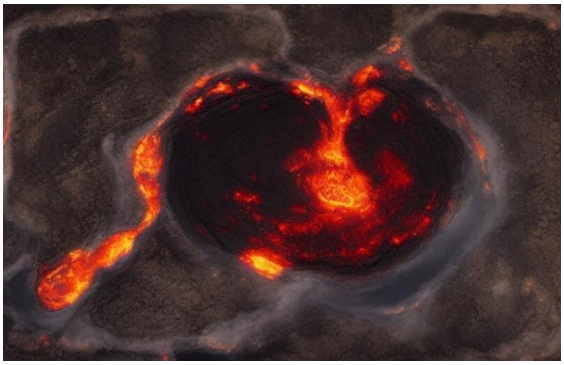

Figure 160 – Generated image

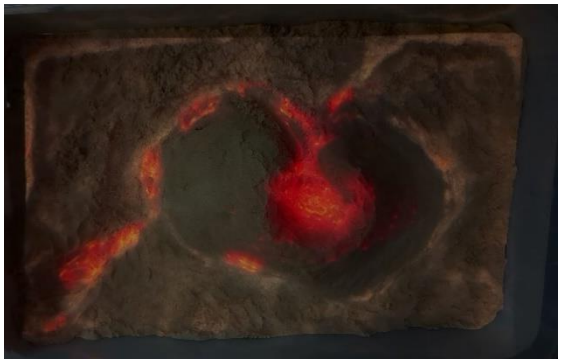

Figure 161 – Top view of the sandbox with the projection

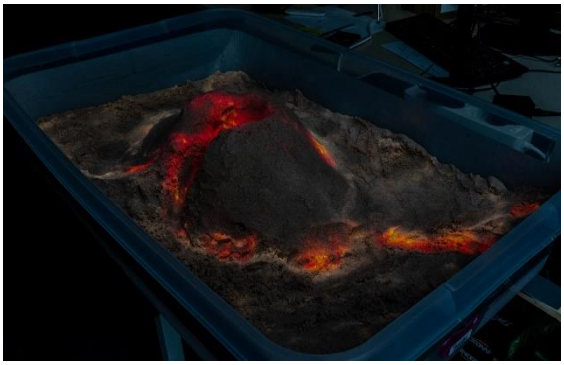

Figure 162 – Side view of the sandbox with the projection

We can see how the model doesn't really know what to do with the landscape when used on the first depth map, but when we change it, it gives us a much more appealing result.

Geyser:

- prompt=top-down view of a landscape, with a large geyser crater and flowing river, realistic, vibrant colors
- negative_prompt=blurry, green
- seed=4074514245

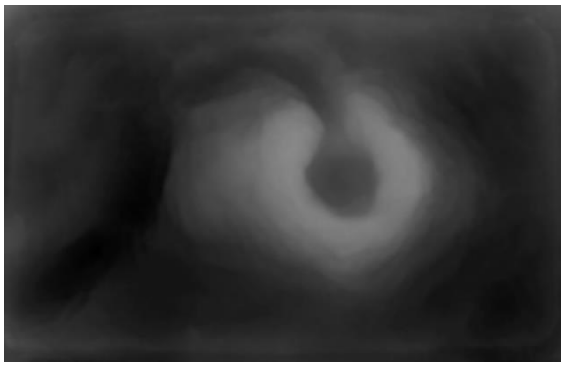

Figure 163 – Image used for depth conditioning.

Figure 165 – Generated image

Figure 164 – Top view of the sandbox with the projection

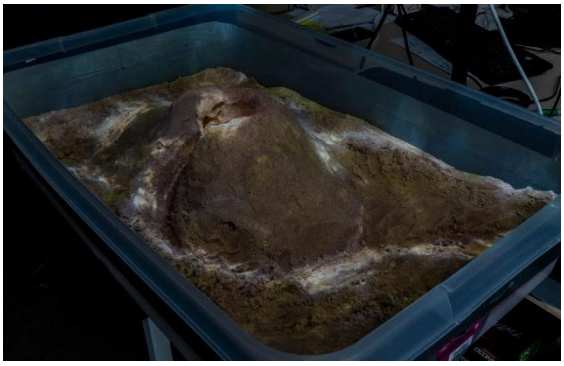

Figure 166 – Side view of the sandbox with the projection

Mountain peak:

- prompt=top-down view of a landscape, a mountain peak with a crater, realistic, vibrant colors
- negative_prompt=blurry, green
- seed=4074514245

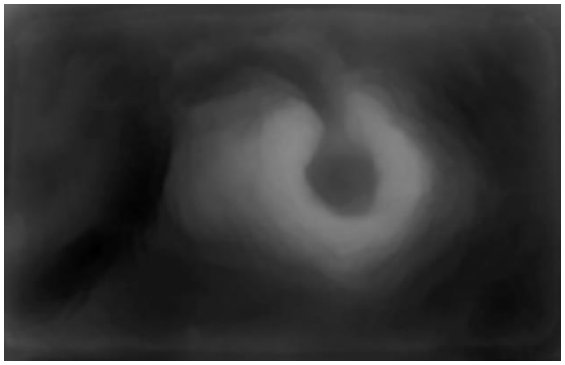

Figure 167 – Image used for depth conditioning.

Figure 169 – Generated image

Figure 168 – Top view of the sandbox with the projection

Figure 170 – Side view of the sandbox with the projection

3.2. Faces

First thing to be noted here is sculpting faces require some ability, and perhaps, tools to, because otherwise the result looks quite messy and irregular. Still some impressive results were achieved thanks to the lax interpretation of the depth map the model did.

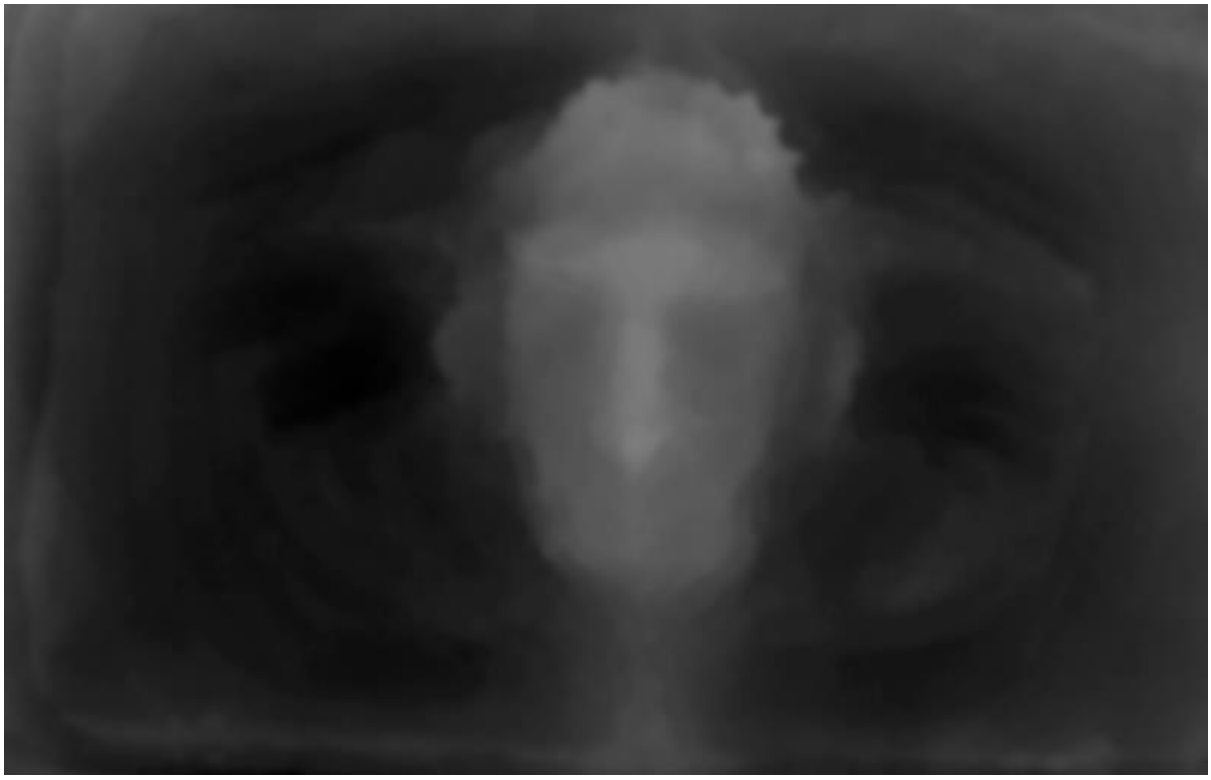

Figure 171 – Depth map used for the generation.

Figure 172 – Top view of the sandbox with the projection

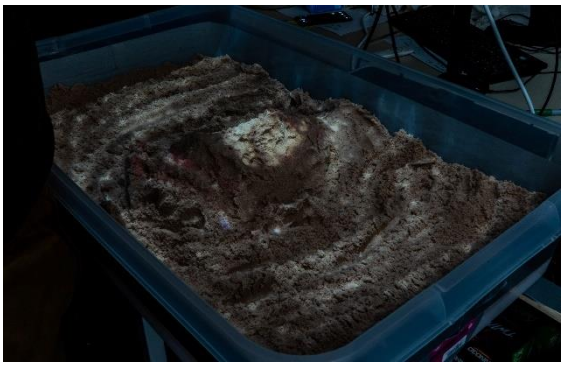

Figure 173 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a portrait photo of a man looking at the camera, realistic, with a blurry background
- negative_prompt=blurry
- steps=30

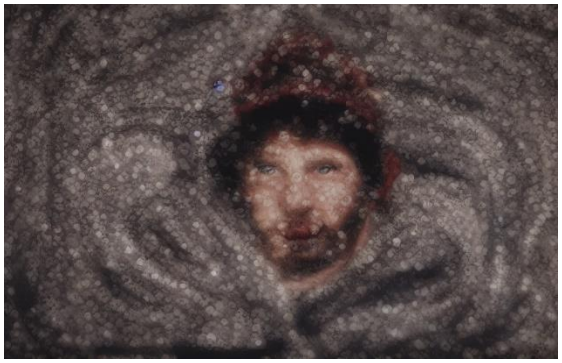

Figure 174 – CFG-scale: 3.5 | seed: 4074514245

Figure 176 – CFG-scale: 7 | seed: 4074514245

Figure 175 – CFG-scale: 12 | seed: 4074514245

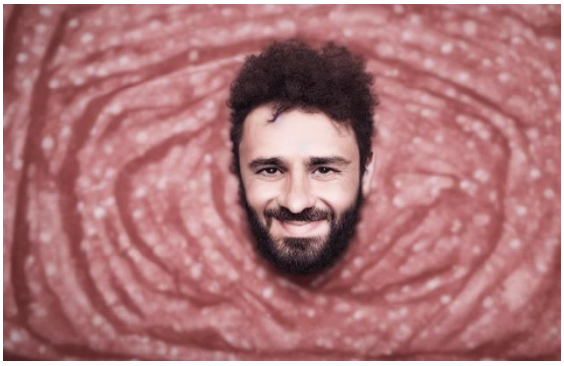

Figure 177 – CFG-scale: 12 | seed: 1800161063

Figure 178 – CFG-scale: 3.5

Something worth noting is that making when making faces, it is required to increase the CFG-scale, as sticking to the 3.5 value we used for the landscapes output really poor results, as seen on Figure 178.

3.3. Food

An apple and an amphora:

Similarly to how it happened with landscapes, here if the CFG-scale is too big, the results begin to look oversaturated. The output with a value of 5 would be slightly better than wit 3.5, but any of those are preferred to 7 or 12.

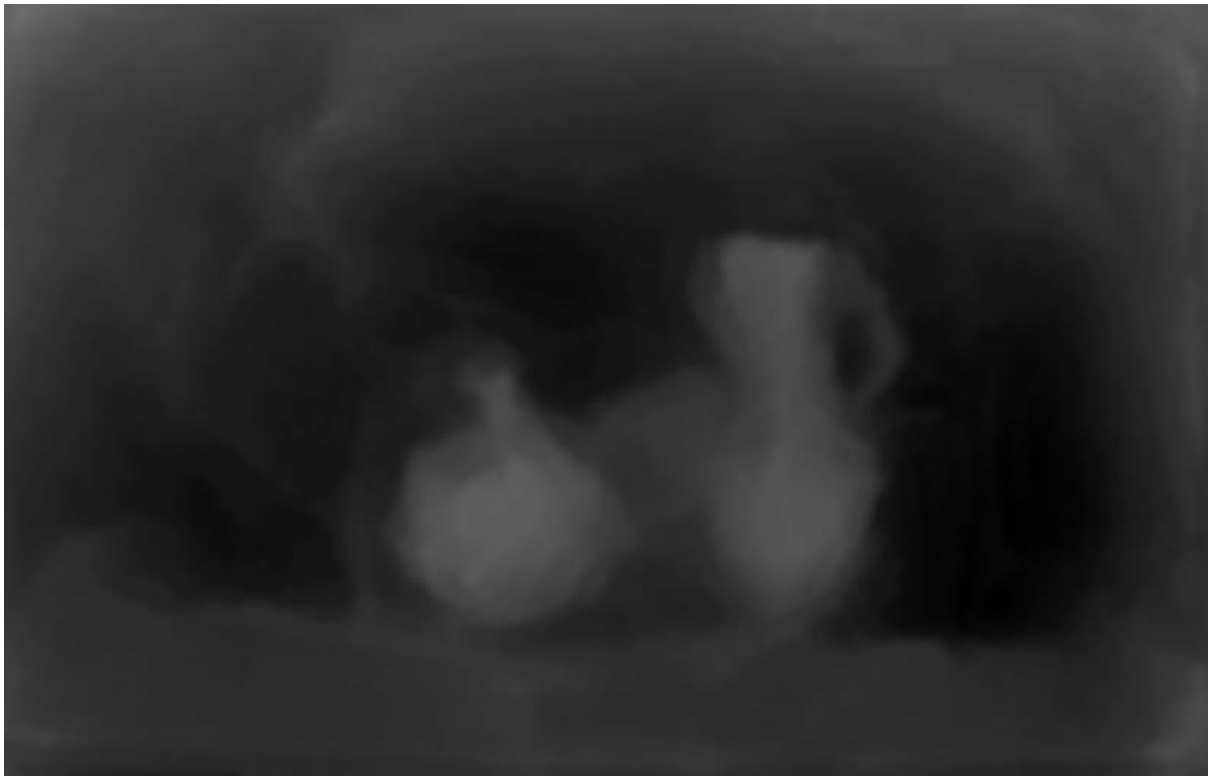

Figure 179 – Depth map used for the generation.

Figure 180 – Top view of the sandbox with the projection

Figure 181 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a photo of a table with an apple and an amphora with a handle to its side
- negative_prompt=blurry
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 182 – CFG-scale: 3.5 | seed: 4074514245

Figure 183 – CFG-scale: 7 | seed: 4074514245

Figure 184 – CFG-scale: 5 | seed: 4074514245

Figure 185 – CFG-scale: 12 | seed: 4074514245

Bowl of cereal

Here we can see some examples of how the either the prompt or the depth map, or both, are misunderstood. The prompt, because we can see in all the images the cereal being spilled around the bowl, and in one of them, the bowl is even face down. The depth map, because in the results, especially when looking at the corners, it seems to understand that the entire image is that of a bowl.

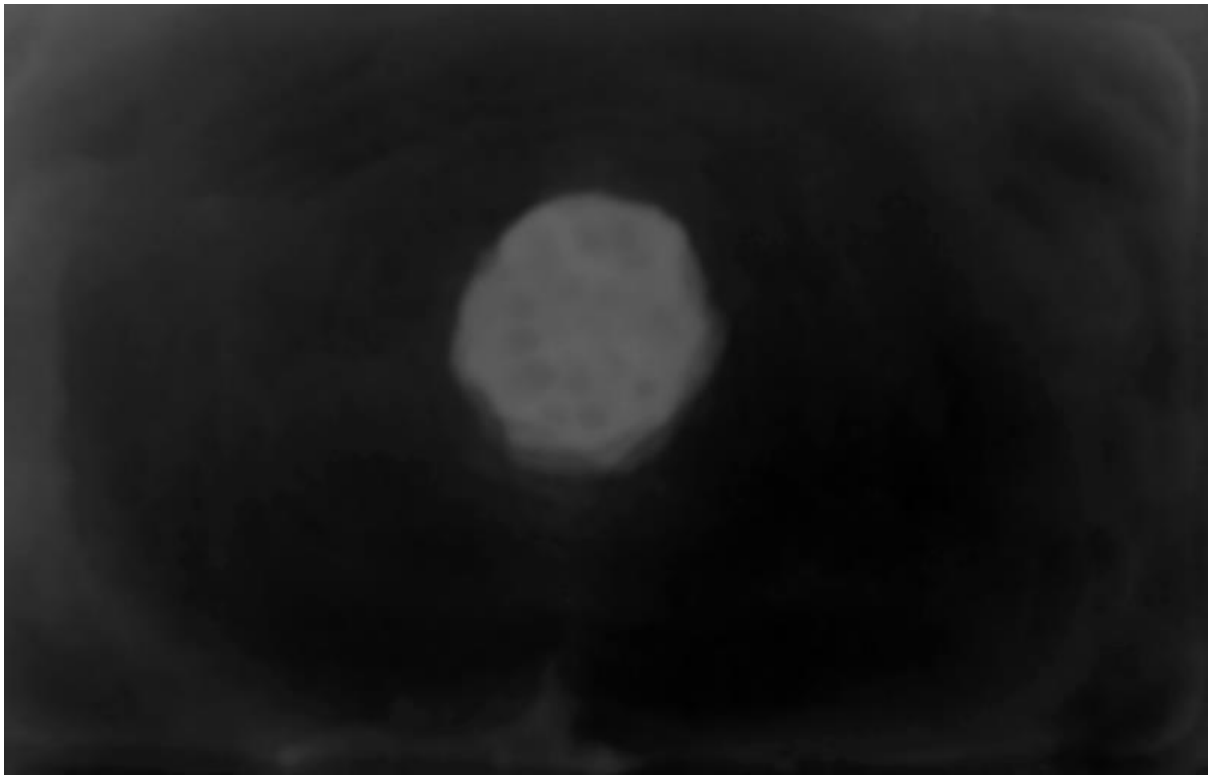

Figure 186 – Depth map used for the generation.

Figure 187 – Top view of the sandbox with the projection

Figure 188 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a top-down view of a bowl with cereals, realistic
- negative_prompt=deformed, land, sand, beach
- sd_model_hash=cc6cb27103

- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 189 – CFG-scale: 3.5 | seed: 3733629334

Figure 191 – CFG-scale: 3.5 | seed: 3586570324

Figure 190 – CFG-scale: 12 | seed: 3733629334

Figure 192 – CFG-scale: 7 | seed: 3586570324

3.4. Animals

Fish

Here, similarly with how it happened with the cereal, we have a somewhat of an incorrect interpretation of the depth map, although only when the CFG-scale increases. We can see how the lower half of the tail gets inflated for apparently no reason, although it also depends on the seed chosen, as in the second case that doesn't seem to happen.

Additionally, unlike with the faces, here upper threshold for the CFG-scale giving good results, seems to be lower, with the value 7 already giving too oversaturate results on some cases.

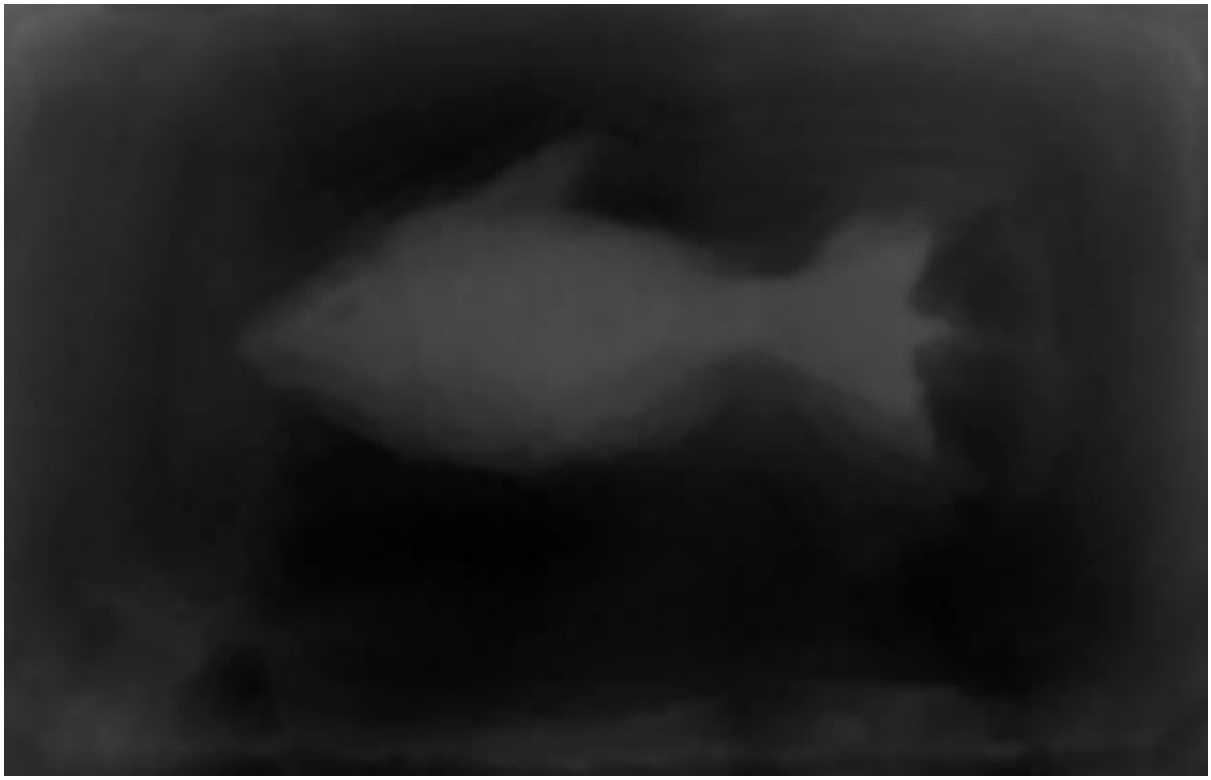

Figure 193 – Depth map used for the generation.

Figure 194 – Top view of the sandbox with the projection

Figure 195 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a photo of a fish on its side, realistic
- negative_prompt=blurry
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 196 – CFG-scale: 3.5 | seed: 4074514245

Figure 199 – CFG-scale: 5 | seed: 4074514245

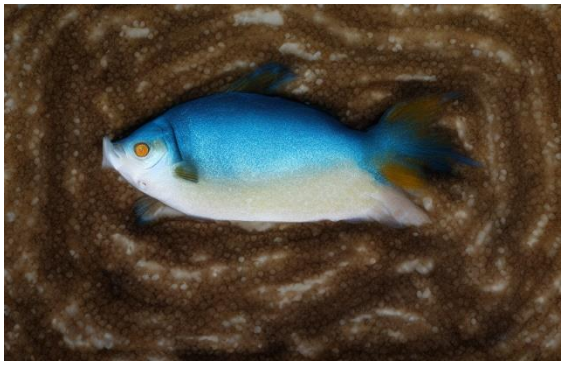

Figure 197 – CFG-scale: 7 | seed: 4074514245

Figure 200 – CFG-scale: 12 | seed: 4074514245

Figure 198 – CFG-scale: 3.5 | seed: 2448783668

Figure 201 – CFG-scale: 5 | seed: 2448783668

Figure 202 – CFG-scale: 7 | seed: 2448783668

Figure 203 – CFG-scale: 12 | seed: 2448783668

Monkey

In this case, something worth noting is that the CFG-scale, although it increases somewhat the quality of the image, introduces some anatomical malformities, trying to paste a hand in place of the tail.

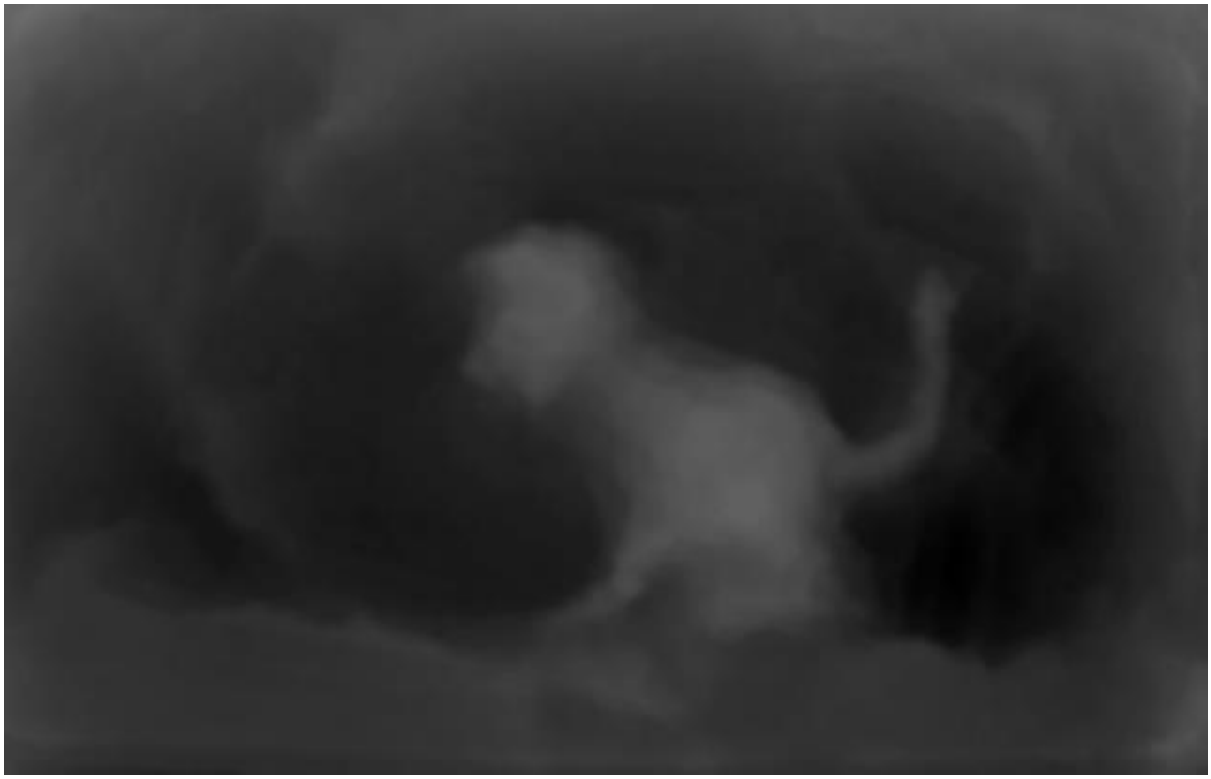

Figure 204 – Depth map used for the generation.

Figure 205 – Top view of the sandbox with the projection

Figure 206 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a side photo of a monkey sitting on the ground
- negative_prompt=blurry
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 207 – CFG-scale: 3.5 | seed: 3677210160

Figure 209 – CFG-scale: 7 | seed: 3677210160

Figure 208 – CFG-scale: 3.5 | seed: 3295393173

Figure 210 – CFG-scale: 5 | seed: 3677210160

Figure 211 – CFG-scale: 3.5 | seed: 4257347534

Figure 212 – CFG-scale: 5 | seed: 1358773249

Turtoise

Similarly, to what happened with the cereals, where we found some cases where the contents of the prompt were misunderstood, as instead of placing a tortoise in a pond, we got a pond in a tortoise.

Additionally, it seems that the model struggles to correctly locate the anatomy of the creature, as when looked closely, most of the had their facial features mixed with the spots.

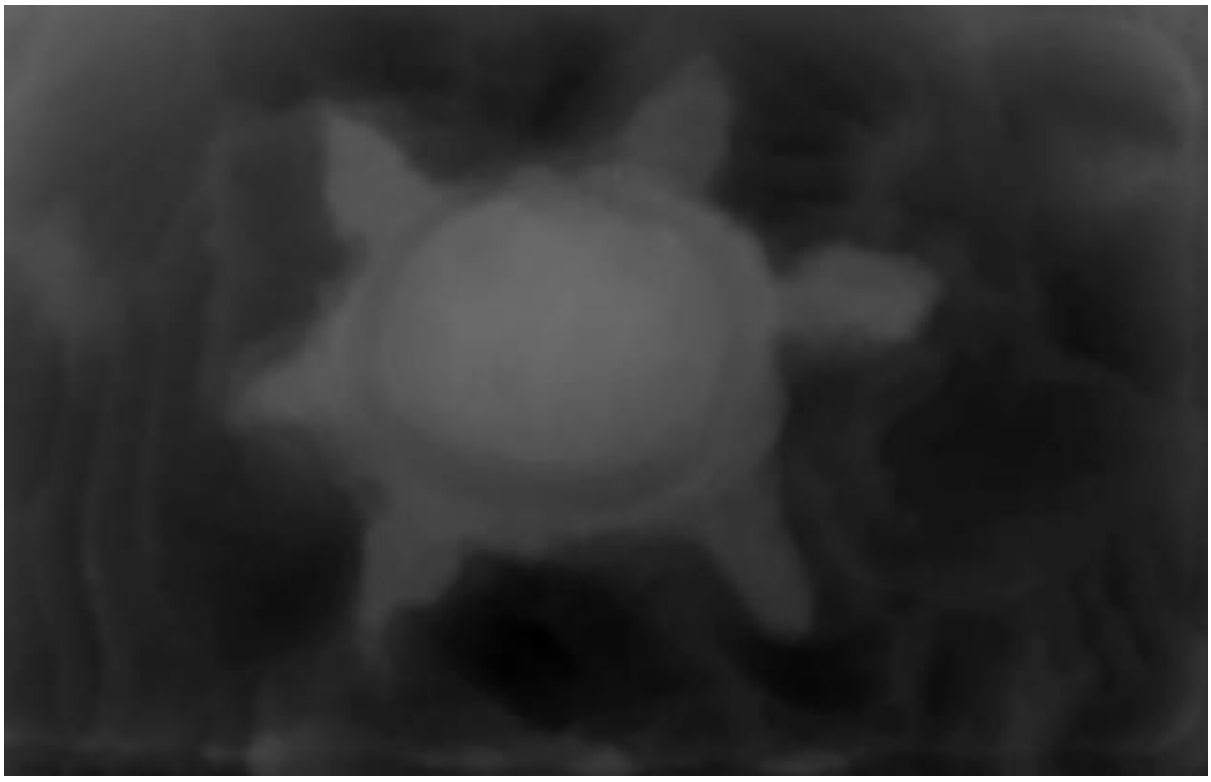

Figure 213 – Depth map used for the generation.

Figure 214 – Top view of the sandbox with the projection

Figure 215 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a top-down view of turtle swimming in a lake, realistic
- negative_prompt=blurry
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 216 – CFG-scale: 3.5 | seed: --

Figure 218 – CFG-scale: 5 | seed: 3295393173

Figure 217 – CFG-scale: 5 | seed: 2150377811

Figure 219 – CFG-scale: 5 | seed: 2007121471

Figure 220 – CFG-scale: 5

3.5. Man made objects

Boat

Here we found out an interesting phenomenon, and that is that even small variations in the words used in the prompt can noticeably influence the final image. Here for example, we tried to get a ship seen from up above, but it didn't quite seem to work until we substituted ship for warship, which seemed to be more easily recognized by the model. The extent to which these changes made an impact seemed to depend on the seed, but still were noticeable in some CFG-scales.

Another thing to note is that if the vessel being generated is white it gets easily confused during the generation process with the sea waves themselves.

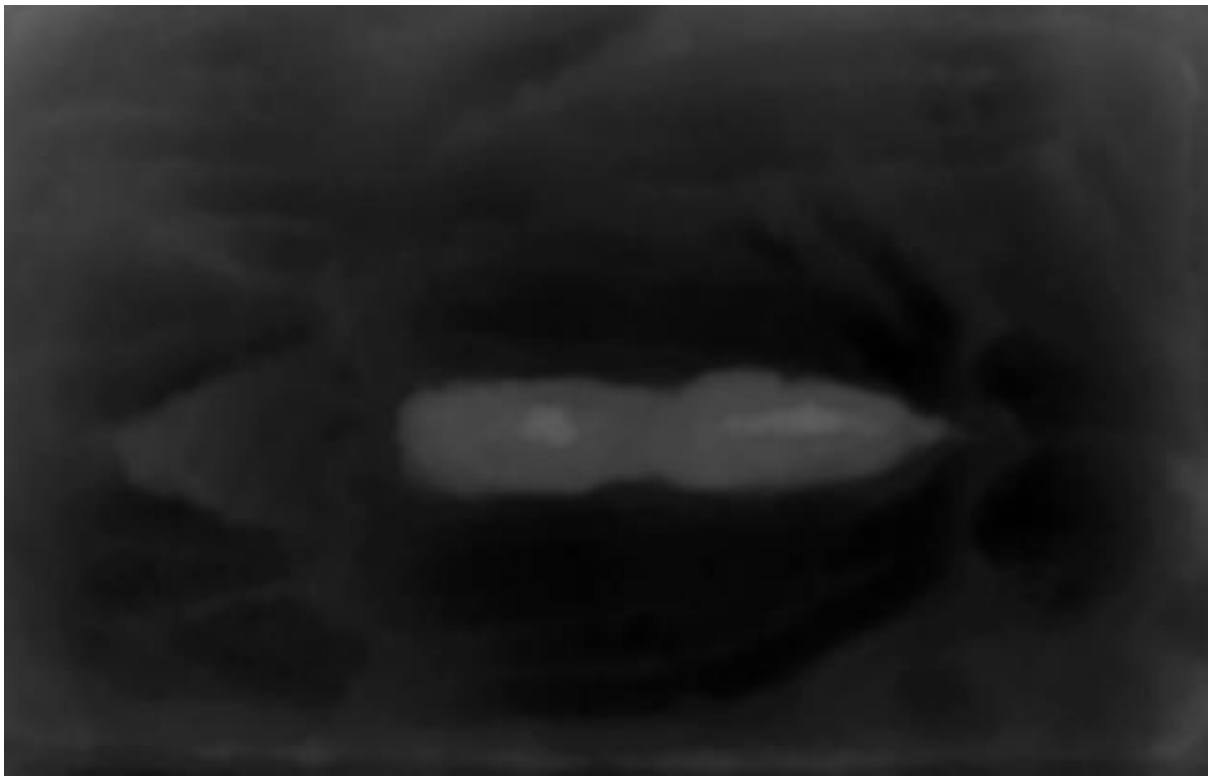

Figure 221 – Depth map used for the generation.

Figure 222 – Top view of the sandbox with the projection

Figure 223 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a top-down view of a [word] sailing through the sea, with small waves
- negative_prompt=blurry
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 224 – word: ship | CFG-scale: 5 | seed: 1923694344

Figure 227 – word: warship | CFG-scale: 5 | seed: 1923694344

Figure 225– word: ship | CFG-scale: 7 | seed: 1923694344

Figure 228– word: warship | CFG-scale: 7 | seed: 1923694344

Figure 226– word: ship | CFG-scale: 12 | seed: 1923694344

Figure 229 – word: warship | CFG-scale: 12 | seed: 1923694344

Figure 230– word: ship | CFG-scale: 5 | seed: 2690869929

Figure 233 – word: warship | CFG-scale: 5 | seed: 2690869929

Figure 231 – word: ship | CFG-scale: 7 | seed: 2690869929

Figure 234 – word: warship | CFG-scale: 7 | seed: 2690869929

Figure 232– word: ship | CFG-scale: 12 | seed: 2690869929

Figure 235 – word: warship | CFG-scale: 12 | seed: 2690869929

Castle

Here we could witness two important things. Firstly, that the better results required a bigger CFG-scale, and secondly, the influence the negative prompt had on the images. As we previously noted, higher CFG-scale images had pretty sharp shadows painted over them. But if we use the negative prompt to state that we want to avoid them, the pictures generated seemed to mostly loose it. They still get a saturated look, but now there is not such dark patches like previously.

Another flaw with this example is the vertical surfaces, as they remain dark when the light shines from above.

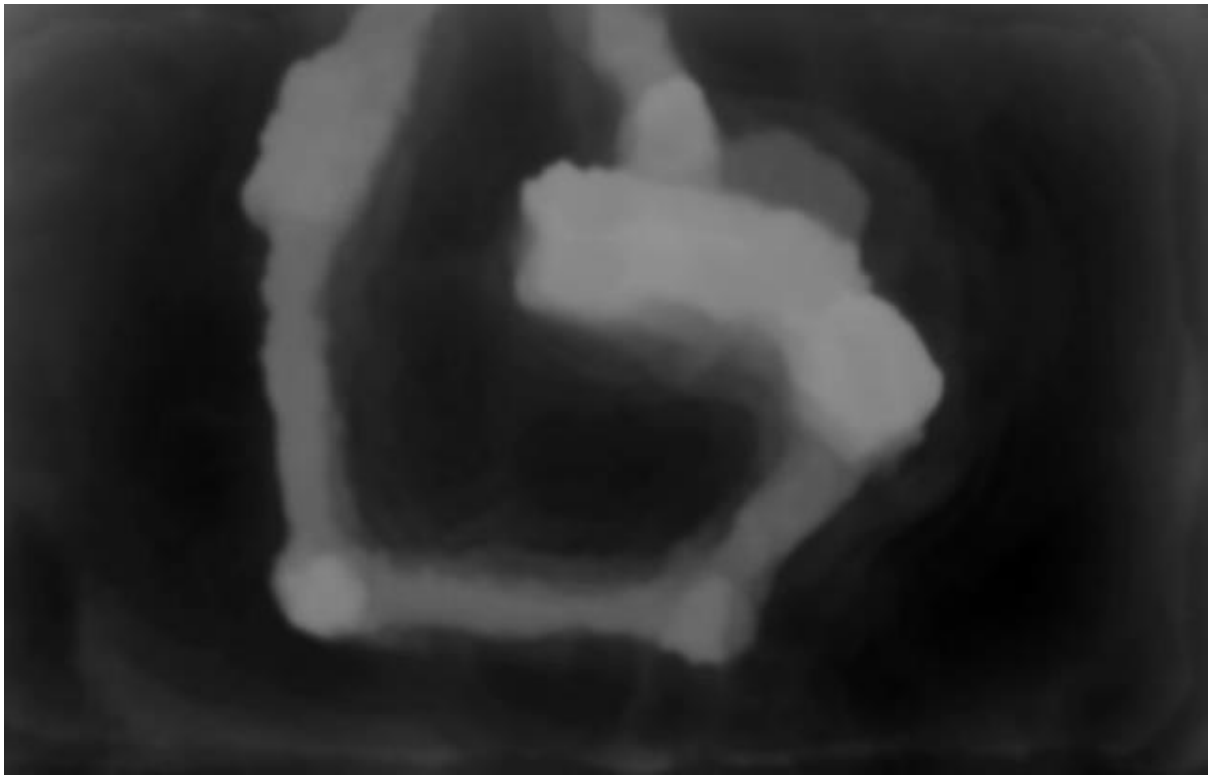

Figure 236 – Depth map used for the generation.

Figure 237 – Top view of the sandbox with the projection

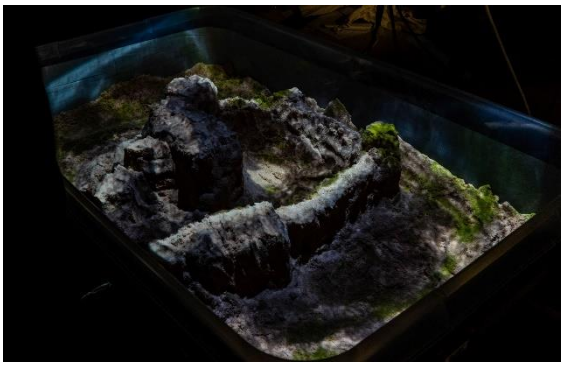

Figure 238 – Side view of the sandbox with the projection

Fixed parameters:

- prompt= a top-down view of a castle with a wall around it and several towers around it with wooden rooftops, realistic
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30

Figure 239 – neg_prompt: blurry | CFG-scale: 5 | seed: 4012900951

Figure 240 – neg_prompt: blurry | CFG-scale: 12 | seed: 4012900951

Figure 241 – neg_prompt: blurry | CFG-scale: 14 | seed: 4012900951

Figure 242 – neg_prompt: blurry | CFG-scale: 12 | seed: 4165823642

Figure 243 – neg_prompt: blurry, shadows | CFG-scale: 5 | seed: 4012900951

Figure 244 – neg_prompt: blurry, shadows | CFG-scale: 12 | seed: 4012900951

Figure 245 – neg_prompt: blurry, shadows | CFG-scale: 14 | seed: 4012900951

Figure 246 – neg_prompt: blurry, shadows | CFG-scale: 12 | seed: 4165823642

3.6. Other

Space

- prompt= a photo of space, with a big planet in the middle and a few planets in the background and stars and galaxies in the background, realistic, vibrant colors
- negative_prompt=deformed
- sd_model_hash=cc6cb27103
- sampler_name=Euler a
- batch_size=1
- steps=30
- cfg_scale=5

Here we can observe the importance of the depth map used to condition the generation process. In the first example, the central planet has less contrast with the background and consequently, Stable Diffusion struggles to understand that it should be a planet. The shape is turned either into void or some pattern along which to draw. If we elevate the surface like in the second example, although it didn't make much sense from out perspective to have a pillar in the place of the planet, Stable Diffusion now manages to identify much easily the shape correctly.

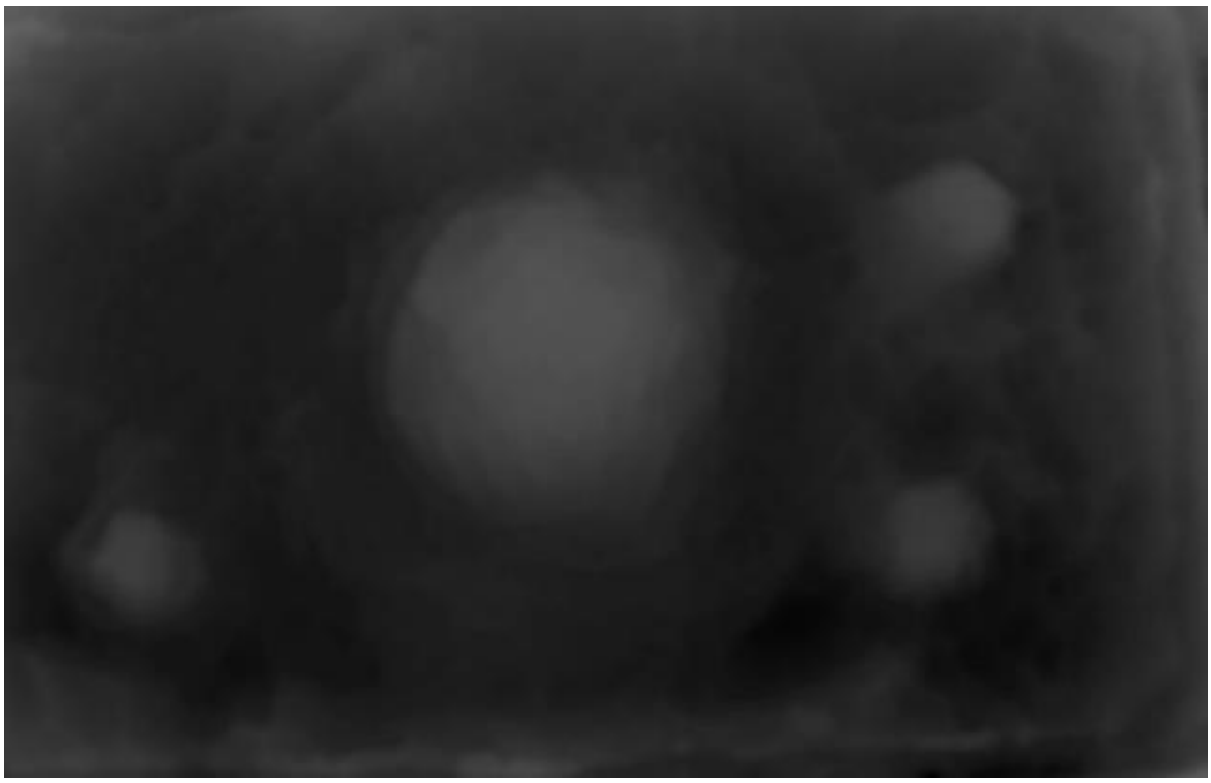

Figure 247 – Depth map used for the generation.

Figure 248 – Top view of the sandbox with the projection

Figure 249 – Side view of the sandbox with the projection

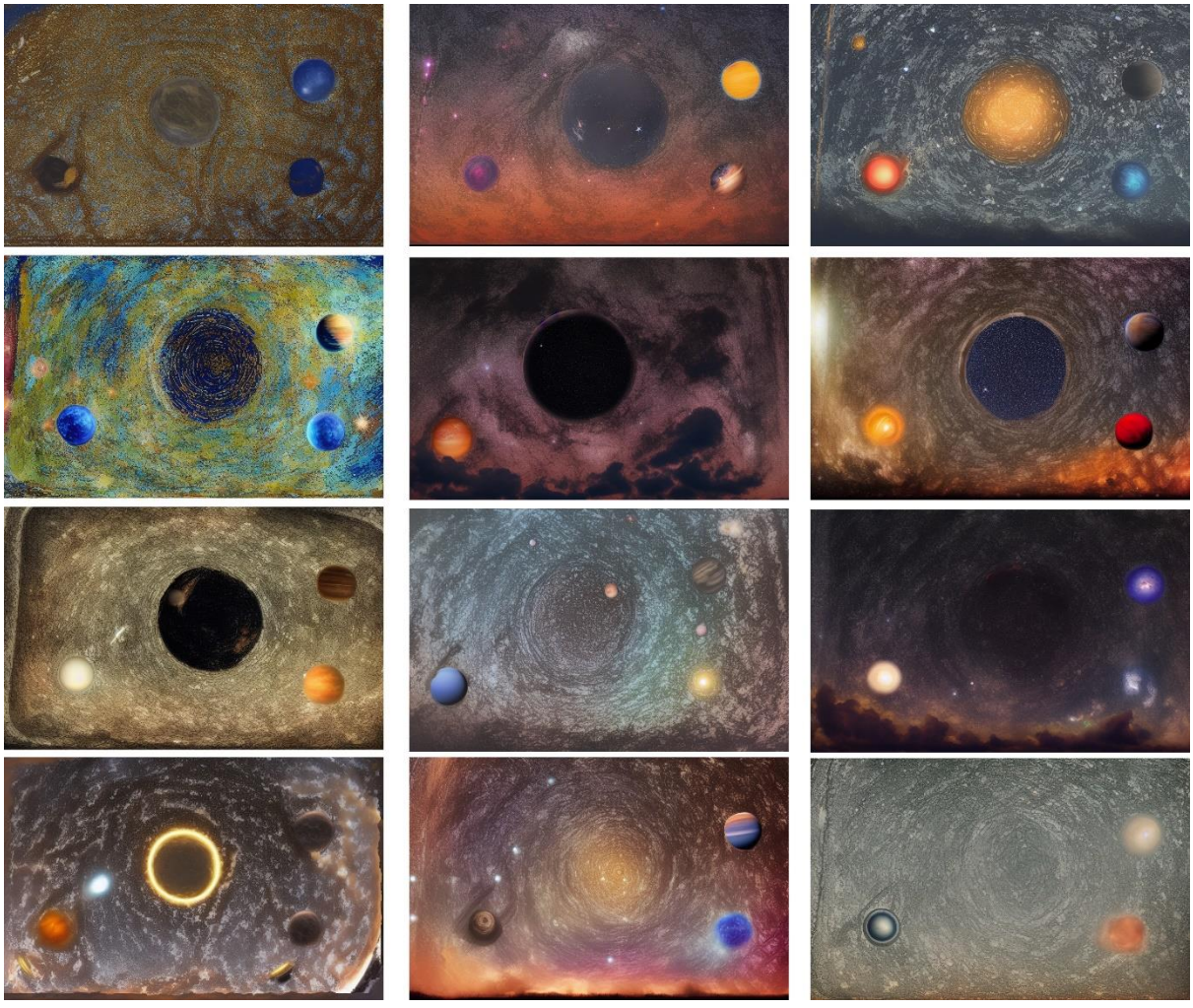

Figure 250 – CFG-scale: 5

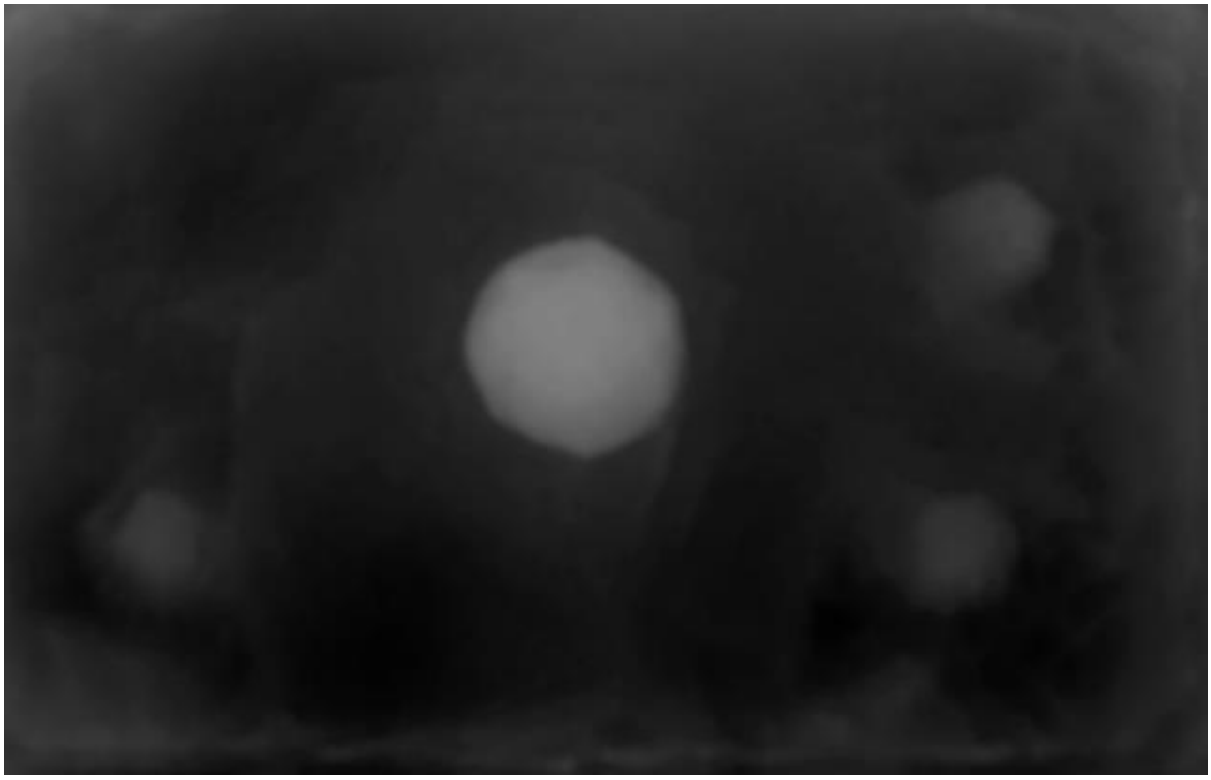

Figure 251 – Depth map used for the generation.

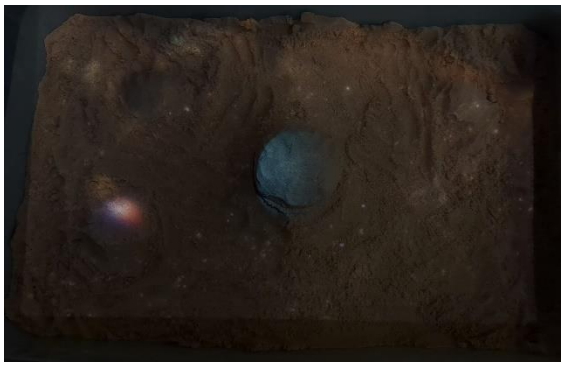

Figure 252 – Top view of the sandbox with the projection

Figure 253 – Side view of the sandbox with the projection

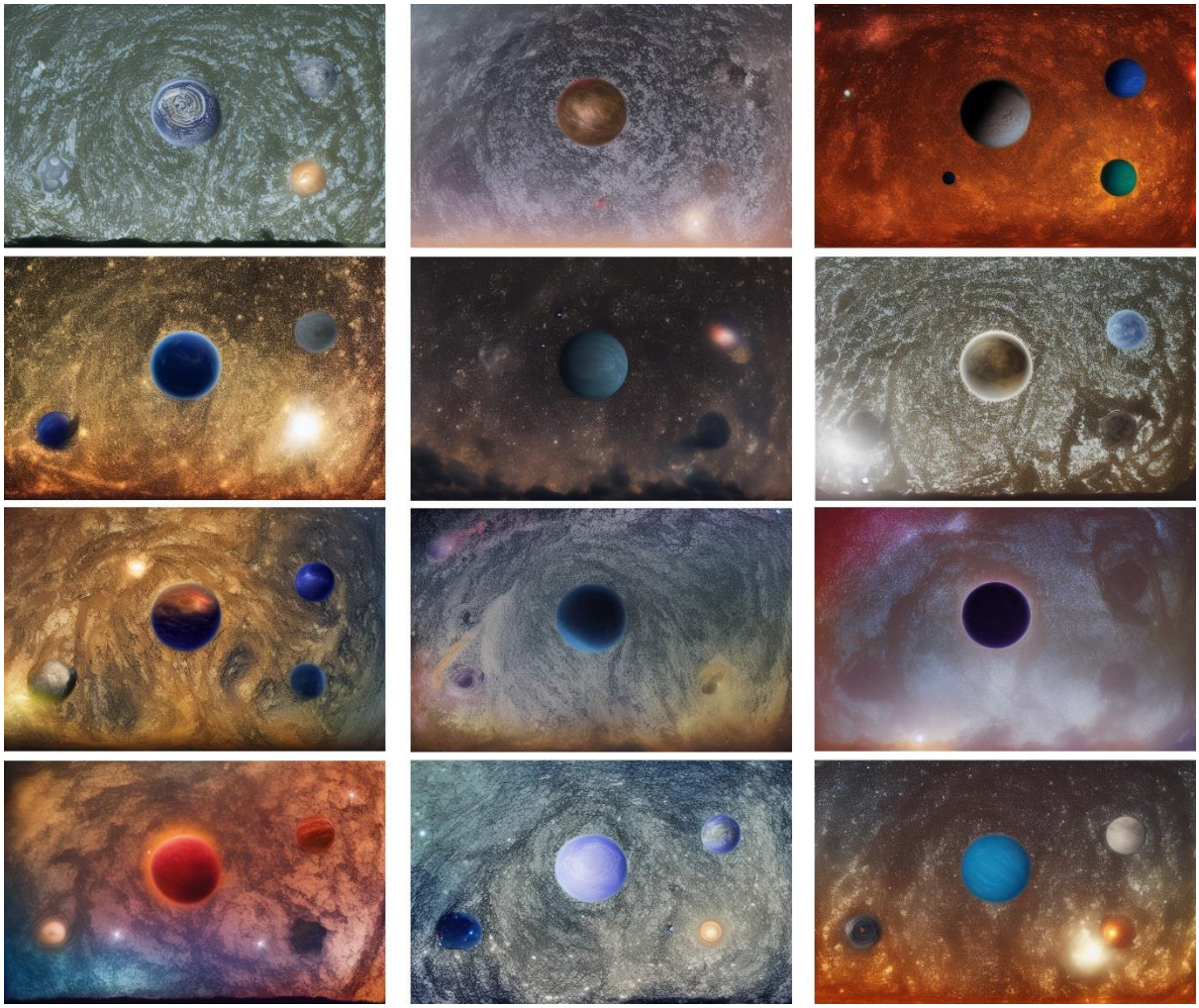

Figure 254 – CFG-scale: 5

4. Conclusions

This project has explored building custom tools to work with Stable Diffusion through an interactive interface. Overall, the results were satisfactory, given that a basic implementation of the original idea was achieved. In this regard, it is important to stress the fact that there is still plenty of room for improvement, as will be laid out in the future work section.

As for the conditioning, note that the results obtained only using ControlNet with depth conditioning were overall satisfactory. We saw that the outputs were mostly congruent with what would be expected in each generation, and consistent.

This, however, is not true for the conditioning done with LoRA, as the quality and consistency seemed to degenerate. This is something that could be explained by the vagueness of the concept that was trained, or due to the dataset and parameters used. Still testing out the models gave some insight on how they work and set a basis for possible future improvements.

Finally, for the physical structure, and despite the flaws it may have, it should be said that the results were satisfactory, as it worked correctly as a bridge between the user and the machine, reacting correspondingly to both sides.

5. Future work

- Improve structure:
    - Design a more stable physical structure: This would make for easier calibration of camera and projector, as well as a better user experience, and for reducing the shakiness of the image when an interaction is made.
    - Network connection: Complete the original idea of having a miniPC detached form the PC responsible of making the computation. This will ease transporting the structure around.
    - Better projector: a more potent projector may be desirable, as in ambient lighting, some colours may become imperceptible. Either that or find a setup where the intensity can be regulated to convenience.

- Improve pipeline:
    - Iterate on the LoRA models: Try other configurations or datasets that may improve the results, or change the approach we take, for example, making more models finetuned for specific landscapes instead of a single general one.
    - Use SDXL turbo: This would make for a close to real time interaction, smoothening out the user experience, as you would see a faster reaction to your changes.
    - Automatize image update: With a similar intention as the previous. The idea is that after a change is made, the code would update the image by itself, without the user needing to signal it.
    - Make preset configuration files: Coud be combined with creating more LoRA models for different things to be displayed. Basically, the idea is depending on what the user

asked to be generated one or another *config.txt* would be read an the parameter from that file used for the generation.

- Replicability: Create the necessary installation files and a guide that would help anyone who would like to replicate the project, offering them a standardized set of steps and taking a away the hassle of versioning.

6. Bibliography

[1] "Highly accurate protein structure prediction with AlphaFold" by John Jumper et al. [Online] Available: https://www.nature.com/articles/s41586-021-03819-2 [Accessed: 31-dec-2023]

[2] "Discovering faster matrix multiplication algorithms with reinforcement learning" by Alhussein Fawzi et al. [Online] Available: https://www.nature.com/articles/s41586-022-05172-4 [Accessed: 31-dec-2023]

[3] "Hierarchical Text-Conditional Image Generation with CLIP Latents" by Aditya Ramesh et al. [Online] Available: https://cdn.openai.com/papers/dall-e-2.pdf [Accessed: 31-dec-2023]

[4] "Stable Diffusion Public Release" by Stability AI. [Online] Available: https://stability.ai/news/stable-diffusion-public-release [Accessed: 31-dec-2023]

[5] "Score-Based Generative Modeling through Stochastic Differential Equations" by Yang Song et al. [Online] Available: https://arxiv.org/abs/2011.13456 [Accessed: 31-dec-2023]

[6] "U-Net" on Wikipedia. [Online] Available: https://en.wikipedia.org/wiki/U-Net [Accessed: 31-dec-2023]

[7] "Stable Diffusion Samplers: A Comprehensive Guide" by Andrew Wong. [Online] Available: https://stable-diffusion-art.com/samplers/ [Accessed: 31-dec-2023]

[8] "Imagen unprecedented photorealism × deep level of language understanding" by Saurabh Saxena et al. [Online] Available: https://imagen.research.google [Accessed: 31-dec-2023]

[9] "Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding" by Saurabh Saxena et al. [Online] Available: https://arxiv.org/abs/2205.11487 [Accessed: 31-dec-2023]

[10] "Manifold Hypothesis" on Wikipedia. [Online] Available: https://en.wikipedia.org/wiki/Manifold_hypothesis [Accessed: 31-dec-2023]

[11] "Learning Transferable Visual Models From Natural Language Supervision" by Alec Radford. [Online] Available: https://arxiv.org/abs/2103.00020 [Accessed: 31-dec-2023]

[12] CLIP GitHub Repository by OpenAI. [Online] Available: https://github.com/openai/CLIP/?tab=readme-ov-file [Accessed: 31-dec-2023]

[13] CLIP Documentation on Hugging Face. [Online] Available: https://huggingface.co/docs/transformers/model_doc/clip [Accessed: 31-dec-2023]

[14] "What are LoRA models and how to use them in AUTOMATIC1111" by Andrew Wong. [Online] Available: https://stable-diffusion-art.com/lora/ [Accessed: 31-dec-2023]

[15] "LoRA: Low-Rank Adaptation of Large Language Models" by Edward J. Hu. [Online] Available: https://arxiv.org/abs/2106.09685 [Accessed: 31-dec-2023]

[16] "ControlNet v1.1: A complete guide" by Andrew Wong. [Online] Available: https://stable-diffusion-art.com/controlnet/ [Accessed: 31-dec-2023]

[17]ControlNet Models on Hugging Face by lllyasviel. [Online] Available: https://huggingface.co/lllyasviel/ControlNet [Accessed: 31-dec-2023]
[18]MiDaS on PyTorch Hub by Intel ISL. [Online] Available: https://pytorch.org/hub/intelisl_midas_v2/ [Accessed: 31-dec-2023]
[19]"How to use Dreambooth to put anything in Stable Diffusion (Colab notebook)" by Andrew Wong. [Online] Available: https://stable-diffusion-art.com/dreambooth/ [Accessed: 31-dec-2023]
[20]"How to use embeddings in Stable Diffusion" by Andrew Wong. [Online] Available: https://stable-diffusion-art.com/embedding/ [Accessed: 31-dec-2023]
[21]"Diffusion Models Beat GANs on Image Synthesis" by Prafulla Dhariwal and Alex Nichol. [Online] Available: https://arxiv.org/abs/2105.05233 [Accessed: 31-dec-2023]
[22]"Classifier-Free Diffusion Guidance" by Jonathan Ho and Tim Salimans. [Online] Available: https://arxiv.org/abs/2207.12598 [Accessed: 31-dec-2023]
[23]"Stable Diffusion 2.0 Release" by Stability AI. [Online] Available: https://stability.ai/news/stable-diffusion-v2-release [Accessed: 31-dec-2023]
[24]laion2B-en on Hugging Face. [Online] Available: https://huggingface.co/datasets/laion/laion2B-en [Accessed: 31-dec-2023]
[25]laion-high-resolution on Hugging Face. [Online] Available: https://huggingface.co/datasets/laion/laion-high-resolution [Accessed: 31-dec-2023]
[26]"LAION-AESTHETICS" by Christoph Schuhmann. [Online] Available: https://laion.ai/blog/laion-aesthetics/ [Accessed: 31-dec-2023]
[27]Stable Diffusion v2-base Model Card on Hugging Face. [Online] Available: https://huggingface.co/stabilityai/stable-diffusion-2-base [Accessed: 31-dec-2023]
[28]"LAION-5B: A NEW ERA OF OPEN LARGE-SCALE MULTI-MODAL DATASETS" by Romain Beaumont. [Online] Available: https://laion.ai/blog/laion-5b/ [Accessed: 31-dec-2023]
[29]CLIP-based-NSFW-Detector on GitHub by LAION-AI. [Online] Available: https://github.com/LAION-AI/CLIP-based-NSFW-Detector [Accessed: 31-dec-2023]
[30]improved-aesthetic-predictor on GitHub by christophschuhmann. [Online] Available: https://github.com/christophschuhmann/improved-aesthetic-predictor [Accessed: 31-dec-2023]
[31]"Progressive Distillation for Fast Sampling of Diffusion Models" by Tim Salimans and Jonathan Ho. [Online] Available: https://arxiv.org/abs/2202.00512 [Accessed: 31-dec-2023]
[32]"Stable Diffusion XL 1.0 model" by Andrew Wong. [Online] Available: https://stable-diffusion-art.com/sdxl-model/ [Accessed: 31-dec-2023]
[33]"SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis" by Dustin Podell et al. [Online] Available: https://arxiv.org/abs/2307.01952 [Accessed: 31-dec-2023]
[34]"Class Action Filed Against Stability AI, Midjourney, and DeviantArt for DMCA Violations, Right of Publicity Violations, Unlawful Competition, Breach of TOS" by Joseph Saveri Law Firm LLP. [Online] Available: https://www.prnewswire.com/news-releases/class-action-filed-against-stability-ai-midjourney-and-deviantart-for-dmca-violations-right-of-publicity-violations-unlawful-competition-breach-of-tos-301721869.html [Accessed: 31-dec-2023]
[35]Case 1:23-cv-00135-UNA, GETTY IMAGES (US), INC. v. STABILITY AI, INC. [Online] Available: https://fingfx.thomsonreuters.com/gfx/legaldocs/byvrlkmwnve/GETTY%20IMAGES%20AI%20LAWSUIT%20complaint.pdf [Accessed: 31-dec-2023]
[36]"Explaining LoRA Learning Settings Using Kohya_ss for Stable Diffusion Understanding by everyone" on GitHub by bmaltais. [Online] Available: https://github.com/bmaltais/kohya_ss/wiki/LoRA-training-parameters [Accessed: 31-dec-2023]

[37]Diffusers Documentation on Hugging Face. [Online] Available: https://huggingface.co/docs/diffusers/main/en/index [Accessed: 31-dec-2023]
[38]stable-diffusion-webui on GitHub by AUTOMATIC1111. [Online] Available: https://github.com/AUTOMATIC1111/stable-diffusion-webui [Accessed: 31-dec-2023]
[39]Automatic1111 API on GitHub by Mikubill. [Online] Available: https://github.com/Mikubill/sd-webui-controlnet/wiki/API#migrating-from-controlnet2img-to-sdapiv12img [Accessed: 31-dec-2023]
[40]"Stable Diffusion. Machine Learning from Scratch" by Binxu Wang and John Vastola. [Online] Available: https://scholar.harvard.edu/files/binxuw/files/stable_diffusion_a_tutorial.pdf [Accessed: 31-dec-2023]
[41]"Azure Kinect DK depth camera" on Microsoft Learn by Tetyana Sych, Phil Meadows and Brent Allen. [Online] Available: https://learn.microsoft.com/en-us/azure/kinect-dk/depth-camera [Accessed: 31-dec-2023]
[42]sd-webui-controlnet on GitHub by Mikubill. [Online] Available: https://github.com/Mikubill/sd-webui-controlnet [Accessed: 31-dec-2023]
[43]"Augmented Reality Sandbox" by Oliver Kreylos. [Online] Available: https://web.cs.ucdavis.edu/~okreylos/ResDev/SARndbox/ [Accessed: 31-dec-2023]