Bachelorarbeit im Studiengang Audiovisuelle Medien

Using Generative Diffusion Models for Video Game Imagery

vorgelegt von **Jonas Seidl**
Matrikelnummer: 40210

an der Hochschule der Medien Stuttgart am 22.09.2023
zur Erlangung des akademischen Grades eines **Bachelor of Engineering (B. Eng.)**

Erstprüfer: Prof. Dr.-Ing. Martin Fuchs
Zweitprüfer: Prof. Dr. Bernd Eberhardt

Declaration on honour

Hiermit versichere ich, **Jonas Seidl**, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: **„Using Generative Diffusion Models for Video Game Imagery“** selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO, § 23 Abs. 2 Master-SPO (Vollzeit)) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Nufringen, 19.09.2023 — Jonas Seidl

Ort, Datum — **Unterschrift**

Abstract

Generative AI has improved a lot recently and with more advancements made we can control more and more the outcome of the images. In this thesis I will try to use generative AI for generating images for a video game. Since the generation process is still very expensive and time consuming we won't see games that can render 30 or more frames per second. But before we can do that we have to know how we can use the AI to generate the images we want. I will show what effects different control parameters have, how they work together and how we can use them for video game imagery.

Zusammenfassung

Generative KI wurde in den letzten Jahren immmer besser und mit jeder weiteren Neuerung können wir immer mehr Kontrolle über den Generierungsprozess erhalten. In dieser Arbeit versuche ich generative KI zu nutzen, um Bilder für Videospiele zu erstellen. Nachdem der Generierungsprozess rechnerisch noch sehr aufwendig ist, werden wir in der nahen Zukunft keine Spiele sehen, die mit 30 oder mehr Bildern pro Sekunde laufen. Um solche Spiele erstellen zu können, ist der erste Schritt herauszufinden, wie wir die KI nutzen könne, um genau die Bilder zu erzeugen die wir wollen. Ich werde die Effekte durch die verschiedenen Einstellungsmöglichkeiten vorstellen, wie diese zusammen funktionieren und wie wir sie zum Erstellen von Bildern für Videospiele nutzen können.

Contents

Chapter 1

Introduction

Video games have been around for decades and during this time the field of computer vision emerged as well. Video games need some type of visual representation and during the last decades we have seen numerous advances in the field of rendering algorithms. With the rise of Artificial Intelligence (AI) and especially the recent improvements in generative AI, with which we can generate complete artworks in a few seconds or minutes, the question arises if we can use this new technology for the rendering of video games. There are already tools available for AI generated textures (Katri, 2023), which utilise the power of these generative AI to generate textures for 3D models in no time compared to the traditional approach of creating the textures either from shaders or from hand painted images.

The goal of this thesis is to write a game that uses generative AI to potentially render the individual images of the game. As generative AI is still computationally very expensive at the time of writing it is not possible to create a game that gets rendered 30 or even 60 frames per second (fps). This means that we will write a game that is very slow paced and will be rendered with 3D rendering algorithms by the game engine. We will then have the opportunity to render the current state of the game with the AI through the user interface of the game. In the end we will have a game which we can play and display using a normal 3D render engine, but with the additional possibility of the new AI based rendering approach. We will also see how we can keep the style

between two renders with the AI and even how we can keep track of multiple players.

I will begin this thesis with a background of current methods and techniques of video games and generative AI. I will also explain the important terms in the next chapter. Then I will describe existing approaches of AI rendered graphics in video games and after that I introduce my approach and the tool and games I created for that. We will see a lot of experiments with the tool in the next chapter before I end this thesis with a conclusion and futher research ideas.

Chapter 2

Background

With the first approaches on video games appearing in the second half of the 20th century (Douglas, 1952; Russell et al., 1962; Alcorn, 1972) a lot of research has gone into the graphics of them. With the rise of generative AI we get more new possibilities of creating the graphics of video games.

2.1 Videogame imagery

2.1.1 The beginning of graphics

The very first video games were made with low-resolution cathode ray tubes (CRT) and had very basic graphics. The game Pong (Alcorn, 1972) for example, which was a game that simulates table tennis, used two rectangles on each side as the paddles of each player and a square as a ball. The positions of those were not calculated with a CPU and neither did a GPU work out the graphics. Everything was calculated by the circuits which were used to encode the video signal. Depending on the position of each object it turned the signal on or off (Winter, n.d.) and one could see the game on the CRT. Computer graphics have improved a lot since then. With the use of processing units we can specify coordinates in a 2D space and simple matrix operations give us the opportunity to translate, scale and rotate images and display

them on the screen. This paved the way for a lot of video games, like ”The Secret of Monkey Island” (LLC, 1990) or the first ”The Legend of Zelda” game (Nintendo, 1986).

2.1.2 3D graphics

One of the first approaches of trying to render 3D graphics in games was the ”Mode 7” in Nintendo’s SNES console (Nintendo, n.d.). This special rendering mode in the console took the 2D graphics of the map or texture it should render and used a scanline algorithm to scan over it. The further the pixels should be rendered ”in the distance” the bigger the offsets between the scanlines on the 2D map got. Additionally, as you did not have any 3D data available, you have to scale and translate the scanlines in order to have a greater field of view in the background (Vijn, n.d.). As revolutionary the Mode 7 was, it wasn’t the best possible output of 3D graphics as it still worked on the 2D domain. Nowadays, video games store the positions of the objects in a 3D space. Graphical Processing Units then calculate for every single pixel of the monitor the projection of the 3D scene onto a 2D plane. While the calculation of this projection is quite simple and only uses some basic matrix operations, the calculation of the actual colors of the scene get more complicated the more realistic it should look like. The simplest form of lighting in a 3D scene is ambient lighting which is calculated just by the color of the object with the intensity of the light source of the 3D scene. But as we want to achieve more sophisticated lighting we have to take physical properties of our objects into account. This results in computations that consider the angle of incidence, the drop-out angle of the light and the transmission of the objects. To create shadows we have to add even more computations. But to really get a level of realism, we have to trace every lightray that falls into our eyes back to its light source. This technique is called raytracing and was proposed by Whitted, 1980. But since this technique needs a lot of computational resources one of the first game that used it for rendering was only the Quake 2 remake ”Quake 2 RTX” (Nvidia, 2019) by Nvidia almost 30 years later who made this game as a demo of this rendering technique. Current research is done to reduce the complexity of

the algorithms while maintaining and improving the realism (Steinberg et al., 2023). Another approach to improve raytracing is by improving denoising algorithms using AI. Hofmann et al., 2023 for example developed a method that enables the denoising of surfaces as well as volumes in real time.

2.2 Generative AI

AI is everywhere and for the approach of this thesis to find a new way of video game imagery it is indispensable. But while Boden, 2018 states that ”Artificial Intelligence (AI) seeks to make computers do the sorts of things that minds can do” it is important to understand that we are not in need of a general artificial intelligence. For our approach we need a neural network that is capable to generate believable images with a high sample quality and has the possibility to consider previous states of an image. Our model should also be fast as we do not want to wait for an image for too long. In the last few years generative AI has seen an enormous rise and models get improved increasingly faster.

2.2.1 Generative Adversarial Networks

One of the first generative image generation models were made with a Generative Adversarial Network (GAN) (Goodfellow et al., 2014). GANs consist of two models: a generative model G and a discriminating model D. The generator tries to generate an image that is as realistic as possible, while the discriminator tries to guess whether the image it gets as an input is made by the generator or is a real image. Goodfellow et al., 2014 describe this process as follows:

> The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency.

The discriminator gets trained by getting real images and fake images generated by the generator. The model is trained to maximize the correct classification of the model. Meanwhile, the generator gets trained by generating fake data in a way that minimizes $log(1 - D(G(z)))$. That means we want the output of the discriminator to be one, which would indicate a real image, and would have fooled the discriminator. The training ends as soon as this happens. The generator can then be used to generate images that resemble the real images of the dataset. (Goodfellow et al., 2014)

2.2.2 Early generative AIs

GANs were not the only generative approach. Other models emerged in the following years, like Variatonal Autoencoders (VAEs) (D. P. Kingma and Welling, 2022), Non-linear Independent Components Estimation (NICE) (Dinh et al., 2015) or ”real-valued non-volume preserving (real NVP)” models (Dinh et al., 2017). In its core a VAE is an Autoencoder model (Rumelhart et al., 1986), which tries to find latent information of a given input and outputs the recreated signal from this information. Therefore, the first part of this model is an encoder that encodes the input into the latent information and the second part is a decoder that retrieves the original data from the latent information as good as possible. Instead of mapping the input data to deterministic latent information, VAEs map the input data to a probability distribution. This introduces some uncertainty to the latent information which in turn captures the variability of the data. It also means that we can use standard stochastic gradient methods to easily differentiate and optimize the model (D. P. Kingma and Welling, 2022). NICE and real NVPs are flow-based models, which means that they define ”a set of powerful, stably invertible, and learnable transformations” to model high-dimensional data (Dinh et al., 2017). While these models can efficiently generate high-resolution images (D. P. Kingma and Dhariwal, 2018; Child, 2021) their sample quality is not as good as with GANs (Rombach et al., 2022).

2.2.3 Probabilistic diffusion models

Another approach for a generative AI model is the probabilistic diffusion model (DM) (Sohl-Dickstein et al., 2015). Based upon the idea to use ”a Markov chain to gradually convert one distribution into another” (Sohl-Dickstein et al., 2015), which comes from the field of non-equilibrium statistical physics (Jarzynski, 1997), a DM takes a simple known distribution, like a Gaussian distribution, and transforms it into another distribution, the target distribution, which in this case is the data distribution (Sohl-Dickstein et al., 2015). This is done by defining a forward trajectory that takes the data distribution and convert it through multiple steps into an analytically tractable distribution, which means a distribution that can be performed using known mathematical functions instead of numerical methods or approximations. Sohl-Dickstein et al., 2015 use for this a Gaussian diffusion which results in a Gaussian distribution. This forward trajectory is what the model learns, but in reverse to result in the reverse trajectory that takes the Gaussian distribution and results in the data distribution. The more steps the trajectory has the easier and simpler it is to estimate the reversal distribution (Sohl-Dickstein et al., 2015), but this also means that we need more forward passes and more resources. With the diffusion model by D. Kingma et al., 2021 DMs achieve now state-of-the-art density estimations that also challenge the dominance of autoregressive models which have long been used for various applications. Furthermore, Dhariwal and Nichol, 2021 find that diffusion models result in a higher sampling quality than GANs. But while resulting in a higher sample quality diffusion models are yet slower and computationally more expensive than GANs because of the multiple diffusion steps and multiple forward passes that come with it (Dhariwal and Nichol, 2021).

2.2.4 Latent diffusion models

Dhariwal and Nichol, 2021 mention that, due to the lack of a latent representation of the data, diffusion models will have difficulties with tasks like representation learning or image editing. However, Rombach et al., 2022 proposed a latent diffusion model

(LDM) for image synthesis which works just as the aforementioned diffusion models but in the data's latent space instead of its pixel space. Because of the lower dimensional representation of the latent space LDMs are more efficient than DMs and need less computational resources. This means that both the training and the sampling efficiency is significantly improved (Rombach et al., 2022). To compress the pixel space into a latent space LDMs use an additional autoencoder that learns the latent space. That means that the diffusion process and the denoising of a latent diffusion model is made inside the latent space instead of the pixel space. For training the images of the dataset get encoded with an autoencoder, the resulting latent data gets diffused and our model learns the backward trajectory with the diffused latent data. Lastly, the denoised data gets decoded again to represent it in pixel space (Rombach et al., 2022). To have more control over the content of our image we additionally add a pretrained language model that encodes text inputs, so called text prompts, into a latent vector. This can be done with a model like CLIP (Radford et al., 2021), a model that is especially made for text-to-image generative models. This pretrained model allows us to tell the diffusion model through a text prompt what we want on the image.

Chapter 3

Terminology

3.1 Stable Diffusion

Stable Diffusion is a latent diffusion model as explained in the previous chapter. It is an open source model and everyone can train their own checkpoints for it. The base model by Rombach et al., 2022 was trained on a subset of the LAION-5B dataset (Schuhmann et al., 2022). This dataset contains around 5.85 billion CLIP-filtered image-text pairs. I use Stable Diffusion as my image generating AI in the practical part of this thesis.

3.1.1 Txt2img & img2img

As mentioned earlier diffusion models can have an additionally pretrained language model to help us to describe the image we want to generate. Stable diffusion lets us do this as well in the so called txt2img mode where we can define a prompt which the AI uses to generate the image.

Stable Diffusion has also an img2img mode. It has the additional feature that we can add another image as input and use it as a reference of what we want the generated image to look like. This mode can be quite useful for example to transfer a style from

an image to the generated one.

3.1.2 Inpaint

When dealing with images as input for Stable Diffusion for the img2img mode we have the option of inpainting. With this function we can remove a bit of the input image and only render the missing part. The generated outcome will fit into the missing part that we removed. This is especially useful if we have for example a generated image that we like but a character in it has a poorly drawn face. Then we can use this method and let Stable Diffusion render a new person or face, depending on what we want to inpaint.

3.1.3 WebUI

Stable Diffusion itself is only an AI that can be used to train models or to use already trained models for image generation. There is, however, no built-in user interface to interact with the AI and generate images. You would have to download the AI to your computer and use it through the python scripts inside the terminal. While it is not too difficult it certainly is tedious to always write out the path to the model, the path to the configuration file and any other settings we want to use. And using add-ons on top of Stable diffusion only adds to the complexity of this method. Therefore, there are so called ”WebUI”. In the context of Stable Diffusion they implement a graphical user interface by starting a local server, running Stable Diffusion on it and providing a website where the user can control the settings using buttons, textfields, sliders and other interface elements. WebUI for Stable Diffusion include for example Automatic1111 (AUTOMATIC1111, 2022) or sygil-webui (Dev, 2022). comfyanonymous, 2023 is another user interface for Stable Diffusion but it is not web based.

3.2 ControlNet

For more control over the generative process Zhang and Agrawala, 2023 created a neural network structure, called ControlNet, that enables various conditional inputs like segmentation maps, depth maps and much more. It works by cloning the weights of a diffusion model into two copies: the ”locked copy” and the ”trainable copy”. The locked copy is just the diffusion model itself and serves as the provider of the diffusion/denoising process and the end result. The trainable copy on the other hand is trained to learn the conditional control and manipulate the outcomes of the locked copy (Zhang and Agrawala, 2023). This works by convoluting the two neural network blocks by a ”zero convolution” layer, a convolution layer where the weights grow from zero to the optimized values (Zhang and Agrawala, 2023). This has the advantage that the training is as fast as just fine tuning the model instead of training the complete parameters from scratch. Additionally, the zero convolution layers do not add new noise to the features of the deep layers (Zhang and Agrawala, 2023) resulting in the same sample quality as the diffusion model itself. How well this conditional input works is shown in Zhang and Agrawala, 2023 with the canny edge map of a deer as an input and the very simple text prompt ”a high-quality detailed and professional image” where they can generate different deers that have the same posture.

Chapter 4

Related Work

4.1 Using AI for video game imagery

To the best of my knowledge no research has been made at the time of writing into using generative AI as a tool to render complete images for a video game. However, Wang et al., 2018b developed a video-to-video synthesis model that uses a GAN to output a video from another video (e.g. a sequence of an edge map). The model was trained on videos of street scenes in various countries, a database of videos of news briefings from reporters and a dance video dataset. While there is no cue about video games in their paper, they showed an example of how one of the authors used a wheel to steer a car in a driving simulation prototype (Wang et al., 2018a). So, we can assume that this AI works in realtime and can be used to render video games. One problem with this approach is that you still need a video as input for the AI, which you have to generate before rendering. Another current problem is the lack of generalised training for this model. For anything other than a driving simulation there is not enough training done.

4.2 Prompt engineering

The term ”prompt engineering” describes the search of prompts that achieve a desired output of a generative AI (Reynolds and McDonell, 2021; Liu and Chilton, 2022). Reynolds and McDonell, 2021 also describe it as the ”programming in natural language”. Engineering in this context does not refer to engineering as in a hard science discipline, like mathematics, computer science or technology. It is more a systematic experimentation of different descriptions of the desired output (Liu and Chilton, 2022). Liu and Chilton, 2022 also carried out a series of five experiments to explore the results of different prompts. They tested different phrasings of the prompt, random seeds, different number of iterations, style words as parameters, and subjects as parameter of the prompt. Their results show that when picking a prompt one should focus on the subject and the style keywords instead of connecting the words together. Furthermore, they found that the seed of the model can influence the quality of the result, while the number of iterations will not necessarily result in a more desirable generation (Liu and Chilton, 2022).

4.3 Using reference images for the img2img mode

While there is no sufficient academic research about how to use the img2img mode to get the best possible result, there are some videos online about this topic. Kamph, 2023 for example showed how to control the lighting of a scene. Therefore, he created an image with the txt2img mode and used this as input for a ControlNet in depth mode. For the input to the img2img mode he used a grayscale image of where the light beams should be. The brighter the lights in the image are the harsher they are in the result. Sarikas, 2023 built upon this technique and showed how it is possible to control the colour of the generated image by using reference images. Like in the method for controlling lighting he created a reference image for a person, by taking a previously generated image with the txt2img mode. But instead of keeping the image as it is he separates the different colours and paint them with no colour variations

and used this as input to the img2img mode. He also used it as input to a ControlNet in depth mode and he used a grayscale version of the original image as input to a ControlNet in edge mode. To get the desired result he also used the colours he wanted in the resulting image in the text prompt in relation with the object that should get this colour. In the video he also showed how to get a custom background from the img2img mode by using the mlsd mode of ControlNet instead of the depth mode. Both techniques used the same prompt (with exception of the changed colours in the second technique) as with the generation of the image in the txt2img mode.

Bozesan, 2023 used stock images to create a reference images. But instead of using the img2img mode from the beginning he only used txt2img with a ControlNet in edge mode and the reference image as input and a ControlNet in OpenPose mode also with the reference image as input to get the people he has in his image in the generated image. He then uses elements of the generated images he likes, replace the equivalent areas in the reference image with them and uses the new reference in the ControlNet. After a few iterations, when he is satisfied with the overall look of the generation, he switches over to img2img and uses the generated image as new reference and input image of the img2img mode. Now he does not use the ControlNet in edge mode anymore, but uses the inpaint function to only create the parts of the images he want to improve. Again, he spends some iterations improving the image until he is satisfied with the end result.

All of the approaches mentioned here are time consuming and need someone who creates the reference images. For a video game that should potentially be rendered with 30 or more frames per second this is not usable, but maybe we can utilize the methods they use and create the neccessary references inside of our game.

Chapter 5

Methodology

With the general background of methods and existing research, the following part will describe the approach of this thesis. We use the Godot 4.0 Game Engine (Juan Linietsky, 2023b) as basis for the game as it is an open source engine with a very easy and user friendly add-on support. Furthermore, we use Stable Diffusion (Rombach et al., 2022) as our generative AI with the Automatic1111 WebUI (AUTOMATIC1111, 2022) as interface for accessing the stable diffusion models as well as the ControlNet for Stable Diffusion WebUI by Mikubill (Mikubill, 2023) as ControlNet for more control over the image generation. All of the used software solutions are open source and therefore perfectly suitable to build upon and use them for the experiments. The complete Godot project and all images in full resolution can be found on Github[1], but as this is made only for fast experimentation a lot of the features are hard coded and not fully developed. The model used for Stable Diffusion is the Stable Diffusion v1.5 model (runwayml, n.d.)

5.1 Using Stable Diffusion in Godot

To be able to use Stable Diffusion inside of a game we first need to develop an add-on for the Godot Engine which sends an API call to one of the Automatic1111 APIs and

[1] Github repository of the project: https://github.com/Joonnas/Godot_SDRendering

receives the generated image as response. This add-on not only needs to handle the API call it also is responsible for the generation of images that can be used as inputs to the ControlNet or as input to the img2img mode and it is responsible to display the AI rendered image we receive from the API. As Stable diffusion returns a base64 encoded image and because the Godot Engine has no own base64 en- and decoder the add-on needs to handle the decoding of it as well. Likewise, the input image to the ControlNet or the input image to the img2img generation also needs to be base64 encoded, so the add-on has to take care of that also.

5.1.1 Base64 En- and Decoding

The images we send to Stable Diffusion and the images we get from it are encoded with the Base64 codec. Therefore, the add-on needs an encoder and a decoder for the images. For the decoder I wrote a function which gets the base64 encoded data as a String parameter. To decode the data we iterate over it and always take the next four characters, retrieve the binary representations of these values from a map I created, concatenate those values to form one big value with 24 bits and lastly split it again into three eight bit values, which are the decoded bytes we can add to a buffer array. After we have iterated over the complete data we then can create a PackedByteArray from my buffer array. The PackedByteArray is a class from Godot with which Godot can create images from the decoded bytes (see A.1).

The encoding works analogous to the decoding. We have a function that takes in the data as an Image (another class in Godot) and can retrieve the PackedByteArray with the method $save\_png\_to\_buffer()$ which comes from the Image class. Then we iterate over the bytes and always take the next three bytes with which we again create a 24 bit long value. This value we split up into four six bit values and retrieve the base64 encoded characters for these values through another map that I created. In the end we concatenate the characters and return the data as a string (see A.2).

At the end of a base64 encoded string there might be one or two equal signs. These

are a padding to ensure that the amount of bits of the data is divisible by four but do not have anything else to add to the data. They are usually decoded as six or twelve bits with the value zero. The two functions take care of them as well.

5.1.2 Sending data to the API

Sending data to the API is fairly simple. First, we specify the URL the API is running on. As my WebUI is running on my local computer the base URL is $https://127.0.0.1:7860$. The port 7860 can be specified in the settings of the WebUI. To this we add the specific path to call the correct API. For the txt2img mode this is $/sdapi/v1/txt2img$ and for the img2img mode we use $/sdapi/v1/img2img$. We also need a header that tells the API that we send the data in a JSON format. Lastly, we need to generate the JSON object with our data we want to send. For txt2img and img2img we create almost the same object. The only difference is that we need an additional input image for the img2img mode in order that we do not raise an error. After we collected the data we can send it with an object of Godot's HTTPRequest class. In the add-on we create this object once at the beginning and every time we want to use Stable Diffusion we can call the $request()$ method on it which takes the URL, the header, the HTTP Method and the data as parameters (see A.3).

For the JSON object we can also use Godot's built-in JSON class. There are a lot of parameters we can specify to influence the generation process. We will only go with the most important ones as fine tuning the parameters with lesser influence would exceed the scope of this thesis. Among the ones we use are the prompt, the negative prompt, the diffusion steps, the size of the output image, the seed and the options for the ControlNet. Inside the options for the ControlNet we specify the input image, the type of the ControlNet (e.g. edge or depth control), the preprocessor and the weight of the ControlNet. For the img2img data I also included the input image (see A.4).

After we sent the data to the API we can now wait until Stable Diffusion has ren-

dered an image and the WebUI sends it back. The HTTPRequest object registers this and fires a $request_completed$ event which we can receive and then process the returned image. The data that the WebUI sends us is a JSON object as a UTF-8 encoded string that we can parse into Godot's JSON object. We can then decode the image with our base64 decoding function and create an ImageTexture from it by first creating an Image object that can create the image from the decoded data with the aforementioned $load\_png\_from\_buffer()$ method and then using the $create\_from\_image()$ method of the ImageTexture class. We can use this ImageTexture now to display it on the screen (see A.5).

5.1.3 Images for ControlNet

For the ControlNet we will focus on two main modes: the edge and the depth mode. While we can work with the preprocessors of ControlNet which generate a canny edge map or a depth map from the given input I decided to create two shaders in addition to the preprocessors. With that we get more control over the mode as we can for example generate an actual depth map from the 3D scene instead of an estimation on the depth values from the preprocessor. For this to work we need to render the depth buffer or an edge map of the 3D scene in Godot without displaying it on the main viewport. I created a class that inherits from Node3D and generates a subviewport, a camera and a mesh instance at the initialisation of an object of that class. We assign a shader to the mesh instance, either for depth control or for edge control. The vertex part of the shaders assign the vertex positions to the screen position which assures that the mesh instance fills the screen and is not displayed somewhere in the 3D scene, no matter where the camera is. We also set the layer mask for the mesh instance on layer 20 true and false on every other layer. We do the same with the camera's cull mask. This means that we also have to set the layer mask on layer 20 to true for every object we add to the scene and leave the cull mask layer 20 of my visible camera to false so we do not render the mesh instance in the main viewport. In the end we assign the mesh as a child of the camera, the camera as a child to the subviewport and the subviewport as a child to the node of the script

itself. If we now want the edge or depth map we can simply get the texture of the subviewport (see A.6).

The shaders themselves are very simple as well. The depth shader makes use of Godot's built-in depth_texture which returns a non-linear depth texture with the values between 0.0 and 1.0. To linearize it we transform the space coordinates into normalized device coordinates (NDC), convert these to the view space and get the result as the negative of the z-value from the view space (Juan Linietsky, 2023a) (see A.7).

The edge shader uses the sobel's edge kernel to get the edges of the scene. To achieve this we can use the built-in screen_texture to get a copy of the rendered scene. We also calculate the screen pixel size to reference parts of the screen texture in pixel space. After that we create a vec3 for the calculation of the colour at the current position of the shader and apply the kernel to the current position of the screen texture. First, we add the horizontal kernel values and then the vertical values. At the end we turn the result into a gray value, which represents the gradient at that position of the texture, and set the colour of the mesh instance to either 1.0 or 0.0, depending on how strong the gradient at that point is (see A.8).

5.2 Game design: City build

To test the add-on I created a small prototype for a city build game where the player can build three different types of houses: A small one, a big one and a skyscraper. There is a world that is divided into a grid of four by four fields where the houses can be placed. The houses are simple red-coloured cuboids and the floor is a simple blue plane. In Godot we then have the possibility to decide the prompt and negative prompt we want to use and the ControlNet option we want to use. Through the code we can change all other parameters, like the seed or the size of the generated image. In this game we only use the txt2img mode of Stable Diffusion and therefore,

we do not need refined colours for reference. To generate a better guide for the edge mode of the ControlNet we could have used higher-detailed models. Since this game prototype serves only as an environment to test the add-on and to validate the first prompts and parameter settings we leave them as simple shapes. The goal with this prototype is to test whether it is possible in general to create the images for a video game and not to create a complete game. We see the results in the experiments part of the thesis.

5.3 Game design: Chess

After the initial tests of the city build game we ask the question how we can build a complete game that can be rendered once every few seconds. The main aim of this thesis is not to create a big game with a lot of game mechanics or a big open world using this method. The answer to that question is to take a game that exists since hundred or even thousands of years: chess. Creating a chess game not only has the advantage of not having to think about a game design. The game is very small with a few simple rules that can be programmed quite fast. In order to receive fast results, the game does only feature the general movements of the chess pieces and does not check whether a move is allowed or not. Unlike with the city build prototype we use higher resolution 3D models (see B) of the pieces to give the ControlNet a chance to distinct which piece is for example a rook and which one is a pawn. With this game we have now a good foundation to conduct some experiments which are explained further in the next chapter.

The first thing we want to achieve is to have a basic chessfield with some pieces on it. Therefore, we will search for a good prompt for the txt2img mode in the next chapter. The next step will be to add a ControlNet to control the positions of the pieces on the field. To ensure a consistency in the players colours (e.g. if the black player moves a pawn it should not switch colours between renders) there have been numerous ideas. One idea has been to separate the scene into the board, the black

pieces and the white pieces, render each part separately and composite everything back together to one image. The problem with this approach is that we would need to render every row of the chessfield separately in order to be able to place black pieces behind white pieces or the other way around. This would be computationally and timewise very costly as we would need 17 renders (eight rows times two colours plus the chessfield). Testing it with the txt2img mode and an edge ControlNet I received promising results in terms of colour accuracy of the pieces (see Figure 5.1), so it might be a good way to assure that the colours stay correct. However, as this approach not only is costly in terms of rendering but also in implementing I abandoned the idea. Another suggestion was to use an additional ControlNet with the segmentation mode. Although the segmentation preprocessors created poor results in generating a good segmentation map for my images (see Figure 5.2), the general idea is worth looking further into it. A segmentation algorithm does not necessarily take the colours of the image into account, though. But if we could find a way to segment the image accordingly with the colours of the pieces we might come closer to our goal. Unfortunately, this would be also quite expensive to implement and therefore is out of the scope for this thesis. The approach that we use is incorporating the img2img mode and using the rendered image of the game engine as input image. With a ControlNet in edge mode we can then further improve the accuracy of the position of the pieces.

Of course, there is also the possibility to combine the named approaches and use for example the img2img mode in combination with an additional segmentation ControlNet. This could further improve the accuracy of the right colour and positions. However, since we do not work with the segmentation mode because of mentioned problems, this approach is not relevant for the further approach in this thesis.

Figure 5.1: Two test generations to try whether multiple render calls would be a solution for a better frame generation

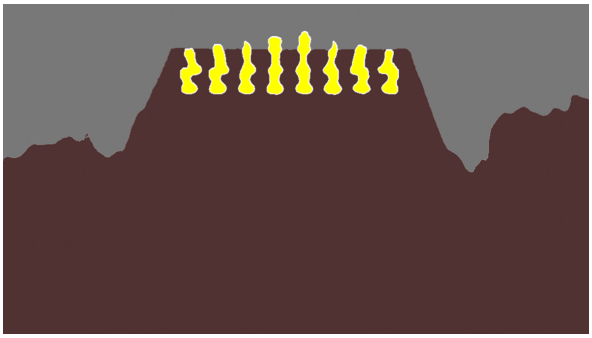

Figure 5.2: Segmentation map of the chess starting setup using the seg_ofade20k preprocessor

5.4 Setting our prompts and controls

For our experiments we try different kinds of prompts. Some of them are very simple and short, with other prompts we try to describe the image as accurately as possible which results in long and complex prompts. While Stable Diffusion has a sense of what colour is and maybe even has some ”knowledge” as to how the pieces of chess look like it certainly does not know the concept of our games and we cannot simply say for our chess game for example to put one of the black rooks on a specific field. Stable Diffusion has no idea where the specified field is or even what we mean with it and therefore will not return the anticipated result. As we look more into the future this problem gets even more depth as we want animations, like the movement of an

arm or the grass that sways in the wind, to be continuous and correct. We do not want the arm in one frame in an upward motion and the next frame in a downward motion.

Fortunately, our setup gives us some options to influence the generation process. First is of course the prompt itself where we can use keywords to describe the image we want as a result. Additionally to the already mentioned findings in the Related Work chapter, we can give terms a weight and tell Stable Diffusion if that term is more or less important. To do this we can simply use bracket pairs which add 0.1 to the weight per bracket pair. Alternatively, we can also just mention the weight we want the term to have by putting the term inside brackets, add a colon and specify the weight after the colon. Next, we can add a negative prompt which tells Stable Diffusion what it should not generate. This negative prompt usually consists of adjectives and descriptions like $ugly, disfigured, poorlydrawn, mutant$, etc. We can weight the individual terms in the negative prompt as well. While we will not control the actual content of the image with the width and height parameters, we will get different results when changing them, as they introduce a complete new ordering of the pixels. Likewise, we will not control the content of the generation with the seed parameter, but it surely is useful to try out different seeds to get the best output.

With the ControlNet add-on we can influence the generation process even more. We will see later how we can specify locations with the depth and edge mode of the ControlNet and that we also can control the shapes of our objects with it. We can even combine multiple ControlNets and use multiple edge modes, an edge mode and a depth mode or any other combination. Moreover, as with the prompt and negative prompt, we have the possibility to set a weight to the ControlNets to specify how much influence they should have on the generation.

When we go one step further, we can use a reference image in the img2img mode to give Stable Diffusion a reference of how we envision our image. This can be just the overall tone of what we envision or, together with the ControlNet, we can also use

the reference as actual reference of where we want the objects to be positioned and what colour they will have. We will try all of this in the next chapter and in the end we should get to the point where we have a solid prompt that describes the visuals of our game very well, include one or more ControlNets to describe positions and use a reference image to describe the colours of our objects.

5.5 How the experiments are structured

In order to find the settings that have the most impact on our end results we conduct some experiments with the described Godot add-on. We start by testing the overall capacity of the add-on using the citybuild game where we experiment with the edge and the depth control of the ControlNet, try different styles on the game and test how well a custom seed keeps the continuity between two renders.

After we have an initial idea of what we can do with Stable Diffusion and the add-on we use the chess game for some further experiments. The goal is to have a chess game that can be rendered with the AI. This means that in the end we should have the correct amount of black pieces, the correct amount of white pieces, a good-looking chessfield and the correct positions of the pieces in every rendered state. To find the best settings we first try to generate only a chessfield without pieces on it and then we try to add the pieces on top of it. We iterate over this procedure by using only the txt2img mode without a ControlNet in the beginning, try the same with a ControlNet, switch to the img2img mode without a ControlNet and finally add again a ControlNet. In the end we take the procedure that was the most promising and weight some of the terms to try to achieve better results.

Chapter 6

Experiments

Before we start with the experiments we have to get to know how Stable Diffusion works in general. For the experiments we use a laptop which has an Nvidia RTX 3080 graphics card built-in. I also tried using Stable Diffusion on older hardware. On an Nvidia GTX 1650 we receive a generation time of about five seconds per sample which results in an average time of about 1:30 minutes with 20 samples and a size of 512px by 512px. On an Nvidia GTX 1080 we receive times of about 1.3 iterations per second resulting in an average generation time of 15 seconds with 20 samples and a size of 512px by 512px. On the RTX 3080 graphics card we receive times of of about 4.8 to 5 iterations per second and an average generation time of 4 seconds with 20 samples and a size of 512px by 512px. We can see that the generation process gets faster with newer and more powerful graphics cards which is important to create 30 fps games in the future.

6.1 Testing the add-on with an AI rendered citybuild game

The citybuild game described in the previous chapter is a simple and small prototype that does not have much to offer. Therefore, it is perfect for testing the overall capabilities of the Godot add-on. With some simple prompts we can use it to test

the overall behaviour of some prompting techniques. The setup with a few houses, as shown in Figure 6.1, is the starting point of this experiment, With this setup we will try a few settings in this section. With the citybuild game we will always have a sample count of 20 and a resolution of 1280px by 720px.

Figure 6.1: Basic setup for the citybuild game. Left: Screenshot from the game. Middle: Depth map of the scene. Right: Edge map of the scene

6.1.1 Difference between Edge and Depth controls

With a few houses on the field the first thing we can to try is to use the depth mode of the ControlNet. As the prompt we use ”skyscraper in san francisco” and with the depth map in Figure 6.1 we get the result in Figure 6.2. As the prompt is yet very

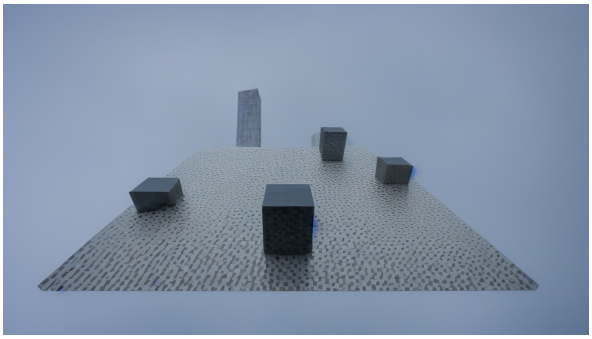

Figure 6.2: Citybuild game rendered with the prompt ”skyscraper in san francisco”, one ControlNet in depth mode with the depth map from Figure 6.1. Resolution: 1280x720, Seed: 2887216139

simple the result is not perfect as well. We can see that we have some kind of houses

where we wanted them to be. Only the skyscraper in the background is not on the field anymore. Trying another four renders gives us the results in Figure 6.3. Still,

Figure 6.3: Four renders of the citybuild. Prompt: ”skyscraper in san francisco”, ControlNet in depth mode with the depth map from Figure 6.1 Resolution: 1280x720, Seeds (from left to right): 1769960022, 1555279224, 2877313234, 3401316410

with different seeds we mostly have the issue with the skyscraper in the back of the field. With the next render we see if the edge mode of our ControlNet can get us a better result (see Figure 6.4). The edge map we use as input for the ControlNet is the same as shown in Figure 6.1. These are not quite the results we want visually,

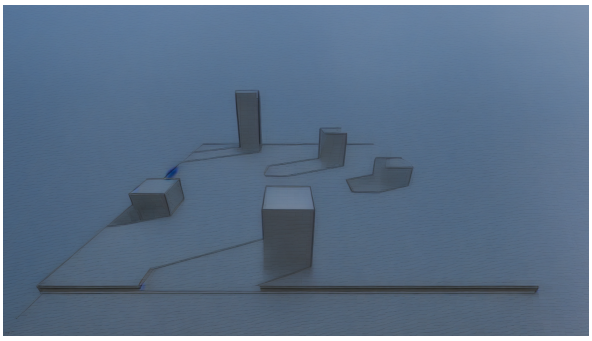

Figure 6.4: Citybuild game rendered with the prompt ”skyscraper in san francisco”, one ControlNet in edge mode with the edge map from Figure 6.1. Resolution: 1280x720, Seed: 1265577581

but the houses are at the correct position. This is probably a good point to find a better prompt. Using ”skyscrapers in san francisco, hyperrealistic, photo, shot from a plane” as the prompt and ”transparent” as the negative prompt I get the results in Figure 6.5. We can still see the outlines from before but now we have a recognisable city. Furthermore, the outlines of the shadows from the scene in Godot are also visible and do not get ignored by the ControlNet and Stable Diffusion. In the rendered

Figure 6.5: Four renders of the citybuild game. Prompt: ”skyscrapers in san francisco, hyperrealistic, photo, shot from a plane”, Negative prompt: ”transparent”, ControlNet in edge mode with the edge map from Figure 6.1 Resolution: 1280x720, Seeds (from left to right): 2042656844, 373582847, 3965700763, 1823212082

result they appear as roads and paths. With no other settings changed and a more extensive prompt and negative prompt we get the examples in Figure 6.6. To further

Figure 6.6: Two renders of the citybuild game. Prompt: ”skyscrapers in san francisco, hyperrealistic, photo, dslr, shot from a plane, a shore in the background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in edge mode with the edge map from Figure 6.1 Resolution: 1280x720, Seeds (from left to right): 1088916345, 941144367

improve the prompt we could mention the surrounding houses, the sky, how we want the field to look like, and so on. As the aim in this section of the thesis is only to test the ControlNet feature of the add-on the prompt is extensive enough. In comparison of the depth mode and the edge mode we can see that in the depth mode we get the playing field without a huge background setting, while with the edge mode the field is much better integrated into the scene.

Before continuing, we will test to use two ControlNets, one in depth mode and the

other in edge mode. As this is not implemented inside of the add-on, we will simply use the depth map and the edge map we get from Godot and put them inside of the WebUI. Every other setting will stay the exact same (see Figure 6.7). We get the

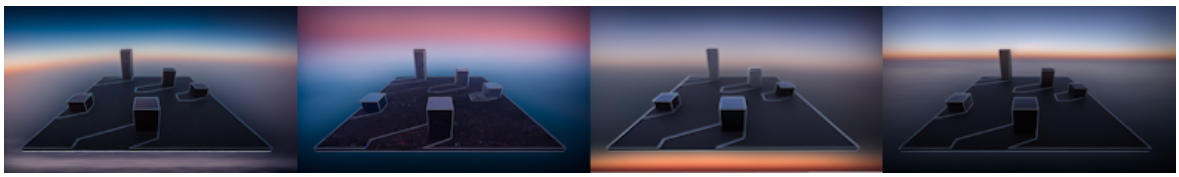

Figure 6.7: Four renders of the citybuild game. Prompt: ”skyscrapers in san francisco, hyperrealistic, photo, dslr, shot from a plane, a shore in the background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in edge mode with the edge map from Figure 6.1 and weight 1, ControlNet in depth mode with the depth map from Figure 6.1 and weight 1, Resolution: 1280x720, Seeds (from left to right): 590593906, 590593907, 590593908, 590593909

exact positioning of the depth mode while still maintaining the positions in the further back from the edge mode. But the depth mode controls the positioning a little too much and therefore we do not have any surroundings anymore. So with the adjusted weights of the ControlNets to 0.55 on the depth mode and 0.5 on the edge mode we see what we get with that in Figure 6.8. The results look much more like a gamefield with a nice surrounding.

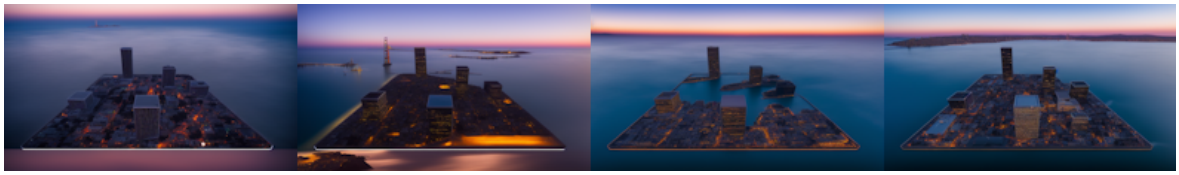

Figure 6.8: Four renders of the citybuild game. Prompt: ”skyscrapers in san francisco, hyperrealistic, photo, dslr, shot from a plane, a shore in the background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in edge mode with the edge map from Figure 6.1 and weight 0.5, ControlNet in depth mode with the depth map from Figure 6.1 and weight 0.55, Resolution: 1280x720, Seeds (from left to right): 3495354305, 3495354306, 3495354307, 3495354308

6.1.2 Changing the style

The next step with the add-on is to see if we can change up the style a little bit. Instead of building a mega city we will try to convert the game to a little farming game by setting up a small village. For this we change the prompt to ”A little village between fields, cartoon, low poly” and the negative prompt stays the same. Figure 6.9 shows the results we get from it using the depth mode of the ControlNet. Using the same settings with the edge mode of the ControlNet we can render the images in Figure 6.10. We can see that we can change the style of our game as easy as modifying

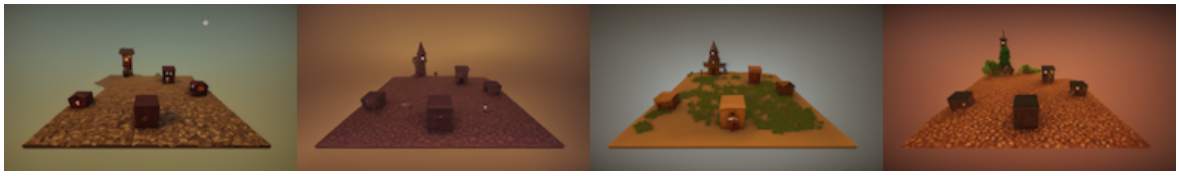

Figure 6.9: Four renders of the citybuild game. Prompt: ”A little village between fields, cartoon, low poly”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in depth mode with the depth map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 1617989392, 382355388, 2819504970, 3453477069

Figure 6.10: Four renders of the citybuild game. Prompt: ”A little village between fields, cartoon, low poly”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in edge mode with the edge map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 3346859821, 3600218588, 2745909202, 1337102410

the prompt for Stable Diffusion. Again we can see the effects of the different modes of the ControlNet. While we have more details in the houses with the depth mode, we get a better integration into the surroundings with the edge mode. So, depending

on the style and feel of the game we can choose or combine them again. With a ControlNet weight of 0.2 on the depth mode and 0.5 on the ControlNet in edge mode we get the renders in Figure 6.11. We use a 0.2 weight on the depth mode since the

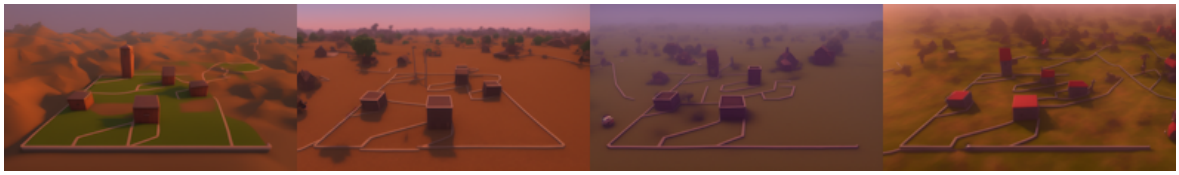

Figure 6.11: Four renders of the citybuild game. Prompt: ”A little village between fields, cartoon, low poly”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in edge mode with the edge map from Figure 6.1 and a weight of 0.5, ControlNet in depth mode with the depth map from Figure 6.1 and a weight of 0.2, Resolution: 1280x720, Seeds (from left to right): 8386122, 8386123, 8386124, 8386125

ControlNet has still too much influence with a weight of 0.5. With this settings we get more detailed houses, while also having a nice surrounding. If we want to improve the details of the houses even more we might have to adjust the prompt a little bit. But since we do not want to create the best possible images in this section we try yet another style.

We start again with one ControlNet in depth mode and try to create a game where the player has to place miniature wooden houses which stand on a table. We use ”miniature houses made of wood on a table, hyperrealistic, photo, dslr, focus” as prompt and added ”bokeh” to the negative prompt we used before. This is because with this setting we receive images where the houses in the back are very out of focus. The results are in Figure 6.12. Unlike the other styles we have always the same look and feel to the images. While they are not the same images there is certainly a consistency between them. In comparison to the results of the edge mode in Figure 6.13 we can see that the houses in the depth mode have more details in it as in the edge mode.. But something interesting is happening here. While we get more or less accurate buildings in the other styles, the edge mode has some problem in this style. We also do not have major variations in the background or the

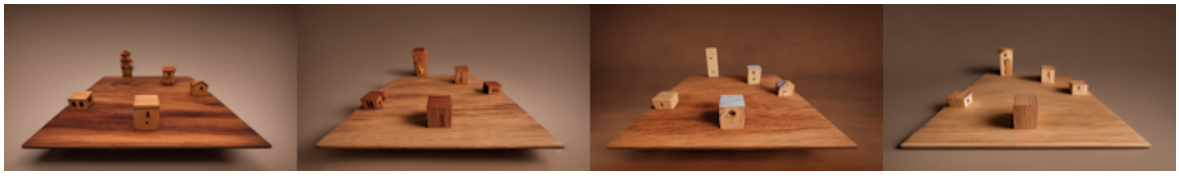

Figure 6.12: Four renders of the citybuild game. Prompt: ”miniature houses made of wood on a table, hyperrealistic, photo, dslr, focus”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows, bokeh”, ControlNet in depth mode with the depth map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 1695448060, 3116592849, 1689327836, 4101170097

Figure 6.13: Four renders of the citybuild game. Prompt: ”miniature houses made of wood on a table, hyperrealistic, photo, dslr, focus”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows, bokeh”, ControlNet in edge mode with the edge map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 2388605994, 1040504688, 3474193983, 3597566479

surroundings. Even after we specify the background in the prompt we get no more variation from the surroundings (see Figure 6.14). As both control modes give us no real control over the background we can assume that this will also hold true if we combine them. This time we use a weight of 0.3 with the depth mode and 0.5 with the edge mode (see Figure 6.15). As expected there are no major changes in the background. This means that we would have to add details to our depth and edge maps to give the Stable Diffusion more information on what we want to see.

For the last style in this section we want a very colourful scene with houses and their doors painted in different colours. As before we start with one ControlNet in depth mode, use ”gray houses in a village, red and blue doors, gray cobblestone paved streets, cartoon, shot from a plane, sky in background” as the prompt and ”ugly disfigured, transparent, poorly drawn buildings” as the negative prompt. Figure 6.16

Figure 6.14: Four renders of the citybuild game. Prompt: ”miniature houses made of wood on a table, hyperrealistic, photo, dslr, focus, wood city in the background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows, bokeh”, ControlNet in edge mode with the edge map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 165118300, 3771961415, 2838319472, 3324834936

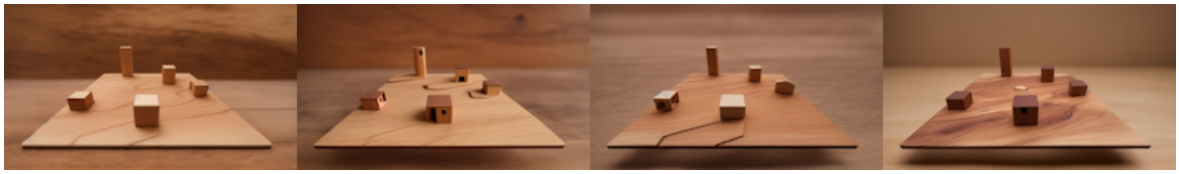

Figure 6.15: Four renders of the citybuild game. Prompt: ”miniature houses made of wood on a table, hyperrealistic, photo, dslr, focus, wood city in the background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows, bokeh”, ControlNet in edge mode with the edge map from Figure 6.1 and a weight of 0.5, ControlNet in depth mode with the depth map from Figure 6.1 and a weight of 0.3, Resolution: 1280x720, Seeds (from left to right): 3719335697, 3719335698, 3719335699, 3719335700

shows the results we get. From the prompt alone we can see that we have to be more specific about what we want. Without specifying the colour of the houses we will get red or blue houses as well, the same issue arises with the floor and we can still see red cobblestones and houses there. We can also see that there are not much blue doors and the colours are not necessarily associated with the objects in the prompt. With the edge map we have the same problem. The colours are everywhere and not only where we specified them (see Figure 6.17). The combination of the two ControlNet modes does again help to mix the traits of the modes, but is still not very useful to constrain the colours to where we want them. The results in Figure 6.18 are made with a weight of 0.5 for both ControlNets. We can see that we need more

Figure 6.16: Four renders of the citybuild game. Prompt: ”gray houses in a village, red and blue doors, gray cobblestone paved streets, cartoon, shot from a plane, sky in background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings”, ControlNet in depth mode with the depth map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 1734589501, 642736486, 3234666540, 793334818

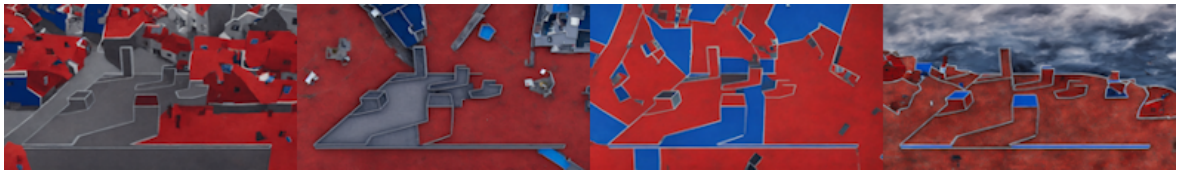

Figure 6.17: Four renders of the citybuild game. Prompt: ”gray houses in a village, red and blue doors, gray cobblestone paved streets, cartoon, shot from a plane, sky in background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings”, ControlNet in edge mode with the edge map from Figure 6.1, Resolution: 1280x720, Seeds (from left to right): 859317643, 1503493787, 4063665964, 3522478992

elaborate approaches to get the colours where we want them.

6.1.3 Using custom seeds vs random seeds

Up until now we always used the same setup of houses with the same depth and edge maps to render the images. This is not a game though. To play a game we need different states and want them to have a continuous style. In the best case there is even no difference in style between frames. Right now, if we place houses and render every state we always use random seeds. As I liked the results with the little village from the previous experiments the most, we will use the same prompt again with one ControlNet in depth mode. Starting from the position we used up

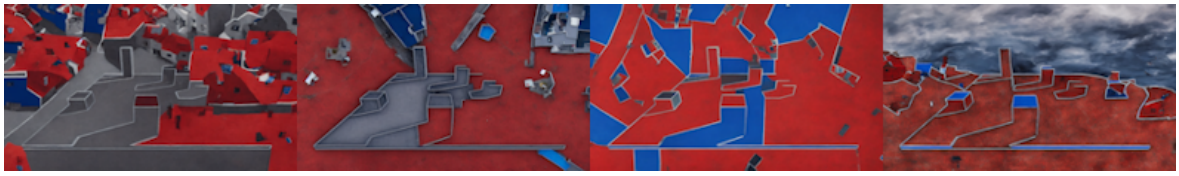

Figure 6.18: Four renders of the citybuild game. Prompt: ”gray houses in a village, red and blue doors, gray cobblestone paved streets, cartoon, shot from a plane, sky in background”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings”, ControlNet in edge mode with the edge map from Figure 6.1 with a weight of 0.5, ControlNet in depth mode with the depth map from Figure 6.1 with a weight of 0.5, Resolution: 1280x720, Seeds (from left to right): 3584442335, 3584442336, 3584442337, 3584442338

to now and adding three houses we get the ingame states in Figure 6.19 and with using the depth maps in Figure 6.20 we get the results in Figure 6.21. As we can

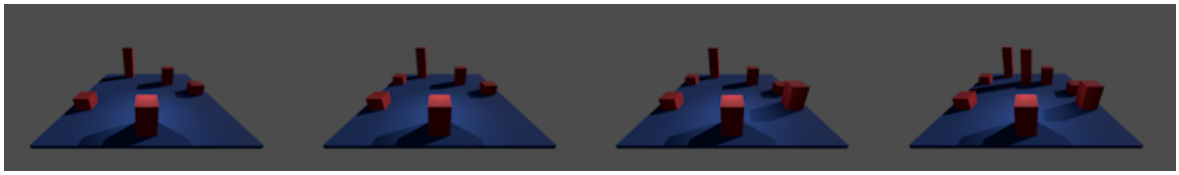

Figure 6.19: Four subsequent ingame states of the citybuild game

Figure 6.20: Four depth maps of subsequent states of the citybuild game

see we get accurate results in terms of the placement of the houses but these are four completely different images. In the first state we have a green-dominant style while the other states are more brown or red. In the first state we have a complete green grass floor, in the states two and three we have some brown dirt and in the last state we have a mix of both. We can also see that the houses change their shapes

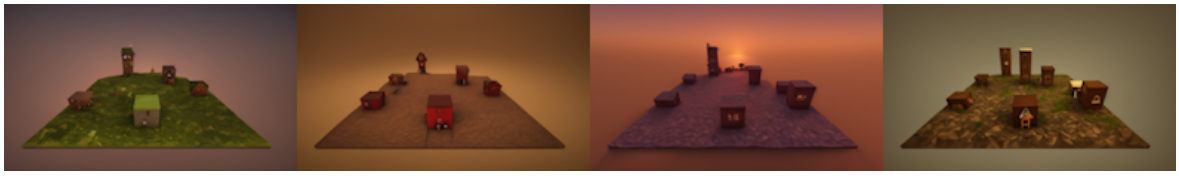

Figure 6.21: Four subsequent renders of the citybuild game. Prompt: ”A little village between fields, cartoon, low poly”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in depth mode with the depth maps from Figure 6.20, Resolution: 1280x720, Seeds (from left to right): 159103597, 58985176, 3347810961, 1876452271

between two renders. Next, we will recreate the same order of the states again and render them with a specified seed (see Figure 6.22). The seed will be determined by the first render where the seed will be random once again. We still have some

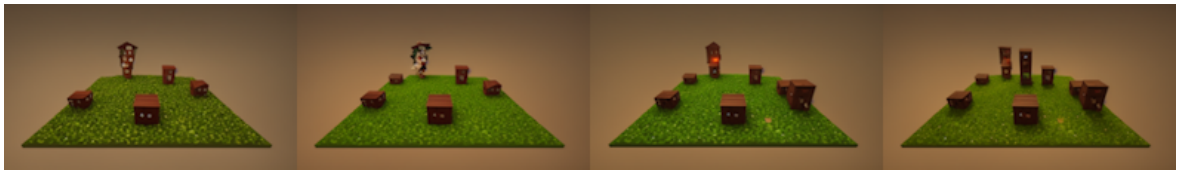

Figure 6.22: Four subsequent renders of the citybuild game. Prompt: ”A little village between fields, cartoon, low poly”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in depth mode with the depth maps from Figure 6.20, Resolution: 1280x720, Seed for all results: 2192436337

variation in the houses, but the overall images of the states stay very similar to the renders of the other states. Having a little variation from one state to the next state we will test how much we variation we have between two completely different states. For the render in Figure 6.23 we fill every still empty space with houses and create a dramatic change between the the previous and this state. It still has the same overall colour to it, we have again a green grass floor and brown wooden houses, but the change between this render and the other renders is a little bit bigger than the change between the other images among each other. If we compare the render with the full field with the last render of the previous rendering series we can still find

Figure 6.23: Left: The ingame state, Middle: The resulting depth map, Right: Generated render. Prompt: ”A little village between fields, cartoon, low poly”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows”, ControlNet in depth mode with the depth map in the middle, Resolution: 1280x720, Seed: 2192436337

similarities in the houses.

6.1.4 First results

Before we move on to the next game, we take a look at the results of what we found in this section. First of all, throughout the experiments I threw in some ideas what we could improve to get better results. These are not the only methods to take more control of the generation process. The goal of the citybuild game was not to get the best results, however. The goal was to get a good basis on how to use the add-on and how we can generate images for our game. We have seen that we can control the positioning of our objects with the depth mode and with the edge mode of the ControlNet. The depth mode is more practical for the actual placement of objects while the edge mode is more suitable to define the shapes of the objects. If we combine the two modes by using two ControlNets separately, we can use the best from both modes and we can control the influence of each ControlNet by setting the weights on them. Another thing we could notice is that the biggest skyscraper in our first experiment was a little bit like a flat image composited into 3D space. We could see this in other renders with the depth control as well, either appearing like a flat image or integrating into the background. This is most likely due to the way the preprocessor generates another depth image which then is used for the generation process. We can see in Figure 6.24 how the generated depth map looks like and we

can see that the skyscraper in the background is quite dark and mixes into the black background. In the same figure we have a render of the miniature wooden houses where this caused some problems and next to it with the same settings except for the preprocessor changed to ”invert” (because the depth map from the Godot engine uses black as the nearest value and white as the furthest away value). The skyscraper with the depth map from Godot is certainly more three dimensional than the one with the generated depth map

Figure 6.24: Left: The depth map that comes from the preprocessor, Middle: The result with the depth map, Right: The result with the inverted original depth map. Prompt: ”miniature houses made of wood on a table, hyperrealistic, photo, dslr, focus”, Negative prompt: ”ugly, disfigured, transparent, poorly drawn buildings, poorly drawn windows, bokeh”, ControlNet in depth mode with the depth map from Figure 6.1, Resolution: 1280x720, Seed (for both results): 1695448060

We have seen that we can change the style and look of our game by just changing the prompt and negative prompt to fit what we want to see in the final image. We have to be aware, though, that we need to find a good prompt depending on the style we want to get. With some styles we can simply ignore the surrounding and let Stable Diffusion work that out, with other styles we have to explicitly describe what we want to have and again other styles ignore the parts of the prompt completely. While we can describe objects using colour, Stable Diffusion does not restrict the colour to that object and uses it quite at will as it seems. This does not mean that we should not use colours inside of our prompts. We just need another way of telling Stable Diffusion where it should apply this colour.

Another important part of using AI to render video games is the continuity between two frames. We get different results if we just use random seeds for Stable Diffusion. Even if we were to describe in our prompt exactly how everything should look like we probably would get different results between frames. Moreover, this approach would be very time consuming as you would end up describing every single detail in the image. Therefore, we can specify one seed that is used the entire time. This works at least if the changes we make to the depth or edge map are relatively small. We still get some differences between two frames, but those differences are small as well. If we make bigger changes to our maps, we cannot assure the continuity between two frames anymore and need another approach here too.

Summing all up, we can say that we can come quite a long way with using only the txt2img mode with one or two ControlNets. We can control the positioning with the depth and edge modes, change the styles to our likings and introduce colour to our image. We also can play one or two moves. Some of the functions work more or less good. To create a complete game this is not enough, however. We have to find a way to constrain the colours better and we need a way to hold continuity between frames, even if there is a major change in the image.

6.2 Playing chess with AI rendering

Now that we have seen how we can use AI to render the images for our game, we transfer this knowledge to another game and see if we can apply it there as easily as in the other game. For this, we use chess as this is a small game that is easy to create, has a lot of different shapes we have to deal with and we need the coloured pieces to match the colours in the previous frame. The worst thing for a chess player that can happen is that his queen is suddenly the opposite colour. In Figure 6.25 we can see the images generated by the Godot engine for the chessfield only and in Figure 6.26 we can see the images generated by the Godot engine for the chessfield with pieces in starting order which we will need both for the experiments

with the chess game.

Figure 6.25: Left: The chessfield without pieces in game, Middle: The depth map of the chessfield, Right: The edge map of the chessfield

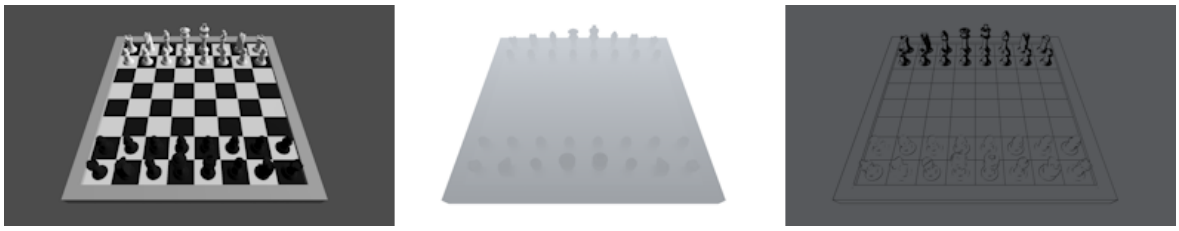

Figure 6.26: Left: The chessfield with pieces in game, Middle: The depth map of the chessfield and pieces, Right: The edge map of the chessfield and pieces

6.2.1 Trying to generate with txt2img only

First, we try to find a suitable prompt that generates a good looking chessfield without any pieces on it. For this we only use the txt2img mode without any ControlNet, a resolution for the rendered image of 640px by 360px and a sample count of 20. The easiest prompt that we try is ”chessfield” without any negative prompts. As we can see in Figure 6.27, Stable Diffusion does know a little bit how chess looks like, but these results are not even close to what we want.

Figure 6.27: Eight generated images to try to generate a simple chessfield. Prompt: ”chessfield”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 4233871134, 423387114135, 423387114136, 4233871137, 4233871138, 4233871139, 4233871140, 4233871141

Especially, as there are results that just have some houses and no chess related objects on them. Most of the images that have some kind of chessfield do not really have black and white objects. But, we give Stable Diffusion also very much room for interpretation with only one term in the prompt. So, we add some more terms and try ”chessfield on a table, b&w tiles, photo, dslr, hyperrealistic” as prompt. The results (see Figure 6.28) are a little bit better since we only have chess related objects in the images now.

Figure 6.28: Eight generated images to try to generate a simple chessfield. Prompt: ”chessfield, on a table, b&w field, photo, dslr, hyperrealistic”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 2555371816, 2555371817, 2555371818, 2555371819, 2555371820, 2555371821, 2555371822, 2555371823

We still have two major problems though: There are pieces on the chessboard, which we do not want right now, and the checkerboard pattern of the chessboard is not really a checkerboard pattern. We keep the prompt for now and add ”chess pieces, ugly, disfigured, distorted, poorly drawn” as negative prompt (see results in Figure 6.29).

Figure 6.29: Eight generated images to try to generate a simple chessfield. Prompt: ”chessfield, on a table, b&w field, photo, dslr, hyperrealistic”, Negative prompt: ”chess pieces, ugly, disfigured, distorted, poorly drawn”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 1864149337, 1864149338, 1864149339, 1864149340, 1864149341, 1864149342, 1864149343, 1864149344

Now, we do not get any chess related object anymore. It seems as if Stable Diffusion does not know how a chessfield without pieces looks like. If we remove the term ”chess pieces” again from the negative prompt, we get again better results as we can see in Figure 6.30.

Figure 6.30: Eight generated images to try to generate a simple chessfield. Prompt: ”chessfield, on a table, b&w field, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 701398354, 701398355, 701398356, 701398357, 701398358, 701398359, 701398360, 701398361

Since we are not in need of a complete empty chessfield this might not be a major problem for us. There is one more thing we can try before adding a ControlNet. Instead of telling Stable Diffusion that we want a chessfield we just tell it how we want the field to look like. This gives us ”checkerboard pattern on a wooden board, straight lines, on a table, b&w, photo, dslr, hyperrealistic, 3D” as prompt and ”ugly, disfigured, distorted, poorly drawn lines” as negative prompt. The results are shown in Figure 6.31.

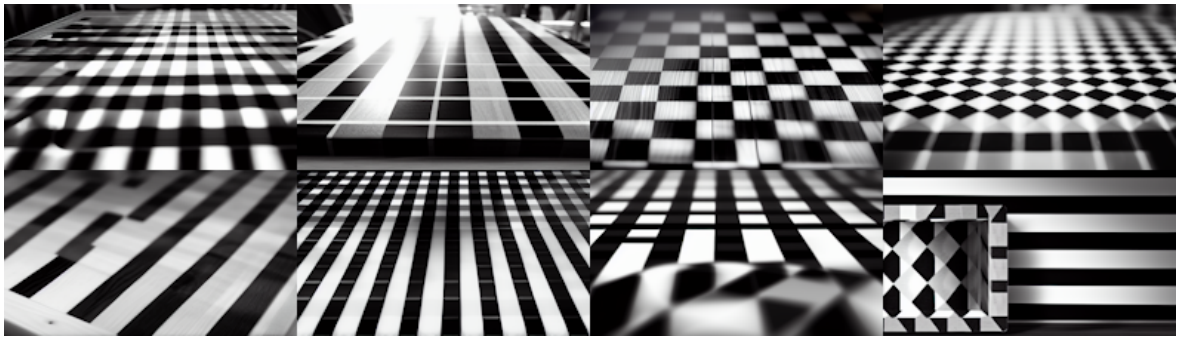

Figure 6.31: Eight generated images to try to generate a simple chessfield. Prompt: ”checkerboard pattern on a wooden board, straight lines, on a table, b&w, photo, dslr, hyperrealistic, 3D”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn lines”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 3836923279, 3836923280, 3836923281, 3836923282, 3836923283, 3836923284, 3836923285, 3836923286

While we certainly get a black and white field in some sense, we only got two checkerboard patterns out of eight seeds and only one of them is not rotated.

Hopefully, we can generate better fields later on with more influence on the scene. As for now, we will try to get at least the right pieces onto the field. We have seen that Stable Diffusion does know how chess looks like. Therefore, we build a prompt using simply ”chess” as its base. I use ”chess on a table, b&w pieces, photo, dslr, hyperrealistic” as prompt and ”ugly, disfigured, distorted, poorly drawn, out of frame” as negative prompt. Every other setting stays the same for now. The results we get (see Figure 6.32) look like some kind of chess game with black pieces and white pieces on the board.

Figure 6.32: Eight generated images to try to generate a chessfield with pieces. Prompt: ”chess on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 580249172, 580249173, 580249174, 580249175, 580249176, 580249177, 580249178, 580249179

Unfortunately, some of the results have only black pieces in them, the positioning of the pieces is not always great and the shapes of some pieces, especially of the knights, are questionable. To fix the ratio of black to white pieces the next approach is adding numbers to the prompt. Instead of using only ”b&w pieces” we now specify the amount of each colour with ”16 black pieces, 16 white pieces” in the prompt. The results of which we can see in Figure 6.33.

Figure 6.33: Eight generated images to try to generate a chessfield with pieces. Prompt: ”chess on a table, 16 black pieces, 16 white pieces, photo, dslr, hyper-realistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 1971478009, 1971478010, 1971478011, 1971478012, 1971478013, 1971478014, 1971478015, 1971478016

This does not work very well. Instead of improving the ratio it gets even worse and in most images there are only black pieces on the field. As we have seen in the previous section controlling colour only with prompts does not work very well, so it is not surprising that it does not work here as well. We will stick with ”b&w pieces” for now and will try one more thing. Right now the pieces are placed all over the field. The question is whether Stable Diffusion can do some kind of starting formation without any control of positioning. For this test we add ”starting order” to my prompt, while the negative prompt still stays the same. We can see the results in Figure 6.34.

Figure 6.34: Eight generated images to try to generate a chessfield with pieces. Prompt: ”chess on a table, b&w pieces, starting order, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, Resolution: 640x360, Seeds (from left to right and from top to bottom): 3996181033, 3996181034, 3996181035, 3996181036, 3996181037, 3996181038, 3996181039, 3996181040

The answer is that Stable Diffusion does not know the starting order of a chess game. We do see some indications of a chess game in starting order in some images but no clear distinction between them and the images where the chess pieces are again on the complete field. This leads to the result that Stable Diffusion does know how the look of a chess game feels like, but it does not have the knowledge how it really looks or behaves.

6.2.2 Adding the ControlNet

With only the txt2img mode we could not generate a good looking chess game with the correct amount of white pieces and the correct amount of black pieces. We had problems with the checkerboard pattern on the field and also the shapes of the pieces were not satisfying in style. For this section we add one ControlNet in depth mode and try to generate again only the board. We start with ”chessfield on a table, b&w, photo, dslr, hyperrealistic” as prompt and ”ugly, disfigured, distorted, poorly drawn, out of frame” as negative prompt. The size of the generated images is still 640px by 360px. The results are shown in Figure 6.35.

Figure 6.35: Four generated images to try to generate a simple chessfield. Prompt: ”chessfield on a table, b&w, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in depth mode with the depth map from Figure 6.25 as input, Resolution: 640x360, Seeds (from left to right): 787135112, 3513073253, 4103105522, 404248958

The board is on a table, has the correct size and has more of a checkerboard on it than without the ControlNet. However, we want the board to have a checkerboard pattern with a size of eight by eight squares that are not rotated. For that we will try to use the edge mode of our ControlNet with otherwise the same settings. The results can be seen in Figure 6.36.

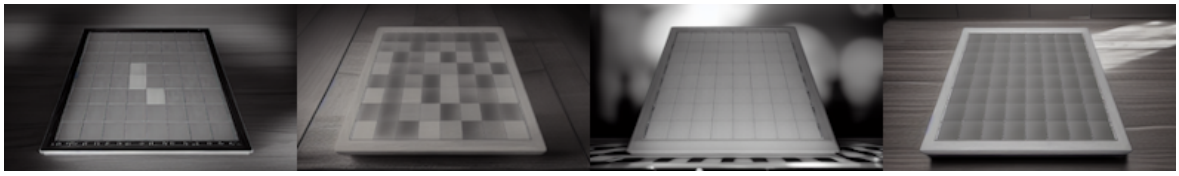

Figure 6.36: Four generated images to try to generate a simple chessfield. Prompt: ”chessfield on a table, b&w, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge map from Figure 6.25 as input, Resolution: 640x360, Seeds (from left to right): 3778436397, 2604710534, 1394272220, 1375113646

While we have now an eight by eight field with straight squares, the checkerboard pattern is gone again as well as the black and white distinction in some images. We are left with gray squares, some darker or brighter than others but otherwise no differences in colour. Now, we try if we get better results if we combine the two modes. Therefore, I use the depth mode with a weight of 1.0 and the edge mode

with a weight of 0.5. The results are shown in Figure 6.37.

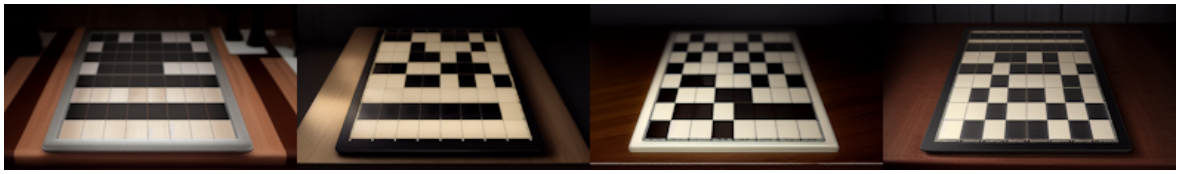

Figure 6.37: Four generated images to try to generate a simple chessfield. Prompt: ”chessfield on a table, b&w, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge map from Figure 6.25 as input and weight 0.5, ControlNet in depth mode with the depth map from Figure 6.25 as input and weight 1, Resolution: 640x360, Seeds (from left to right): 2967458447, 2967458448, 2967458449, 2967458450

We clearly have again our differences in colour between the tiles but it is still not a checkerboard pattern. In the next step we add the pieces again.

We start again with the depth mode of our ControlNet and change the ”b&w” term in our prompt to ”b&w pieces”. There are no further changes. With this we get the results in Figure 6.38.

Figure 6.38: Four generated images to try to generate a chessfield with pieces. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in depth mode with the depth map from Figure 6.25 as input, Resolution: 640x360, Seeds (from left to right): 1583874397, 2395028380, 3626070975, 3301945769

While the depth mode of the ControlNet is good in positioning things it certainly is not capable of capturing more complex shapes. We can also see that it does not

recognise every piece on the chessfield, especially in the back of the image it has problems with the correct amount of pieces. As we are dealing with complex shapes in this case maybe the edge mode can capture the pieces better than the depth mode. With equal settings and the edge mode on one ControlNet we receive the results in Figure 6.39.

Figure 6.39: Four generated images to try to generate a chessfield with pieces. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge map from Figure 6.25 as input, Resolution: 640x360, Seeds (from left to right): 852053898, 3668972323, 650011762, 4150199923

Now, we receive the correct amount of pieces on the correct position. The shapes are still not very accurate and the colours are not yet correct as well but with this we could be able to move the pieces around. Before we do that we try if we can get even better results with a combination of both ControlNet modes. As the edge mode returned better results we use this time a weight of 0.5 for the ControlNet in depth mode and a weight of 1 for the ControlNet in edge mode. The results are shown in Figure 6.40.

Figure 6.40: Four generated images to try to generate a chessfield with pieces. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge map from Figure 6.25 as input and a weight of 0.5, ControlNet in depth mode with the depth map from Figure 6.25 as input and a weight of 1, Resolution: 640x360, Seeds (from left to right): 3624316404, 3624316405, 3624316406, 3624316407

Other than that the tiles of the board have again more differences between each other the quality of the results does not improve. Furthermore, we can see that the result gets a more artificial style to it as if it is a painting or handdrawing.

Until now we have only rendered the chess game in its initial state and with the edge mode received sufficient results in terms of the amount of pieces and their positioning. Now, we can try to move some of the pieces around and see what happens when we change the scene and the edge map. For this, we will generate the first image with a random seed again and use that seed for the next renders. The render of the initial state is shown in Figure 6.41.

Figure 6.41: Render of the initial state of a gameplay. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge map from Figure 6.25 as input, Resolution: 640x360, Seed: 2954826565

In this initial state we have unfortunately only white pieces. The field does not look too bad, though. If I refer to the black pieces in this experiment I refer to the black pieces in the Godot scene as these will not change their colours. So, for the next four moves we move the left white knight, then the left black knight, then the white pawn in front of the white queen and then the black pawn in front of the black king. The next four moves in the game are seen in Figure 6.42. The rendered results are seen in Figure 6.44.

Figure 6.42: Four subsequent ingame states of the first four moves of a chess gameplay

Figure 6.43: Four subsequent depth maps of the first four moves of a chess gameplay

Figure 6.44: Four subsequent renders of the first four moves of a gameplay. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge maps from Figure 6.43 as input, Resolution: 640x360, Seed (for all images): 2954826565

The colours of the pieces change a little bit and the chessfield also changes its colours but overall the style stays consistent and the colours of every piece is always the same. We can also see that the amount of pieces is still correct and no additional pieces are generated. We move a few steps forward and see if there is still no change in the pieces after we mixed the field up a bit. In Figure 6.45 we can see the field in the game engine and in Figure 6.47 we can see the AI rendered results.

Figure 6.45: Two ingame states of a chess gameplay where the pieces are mixed up

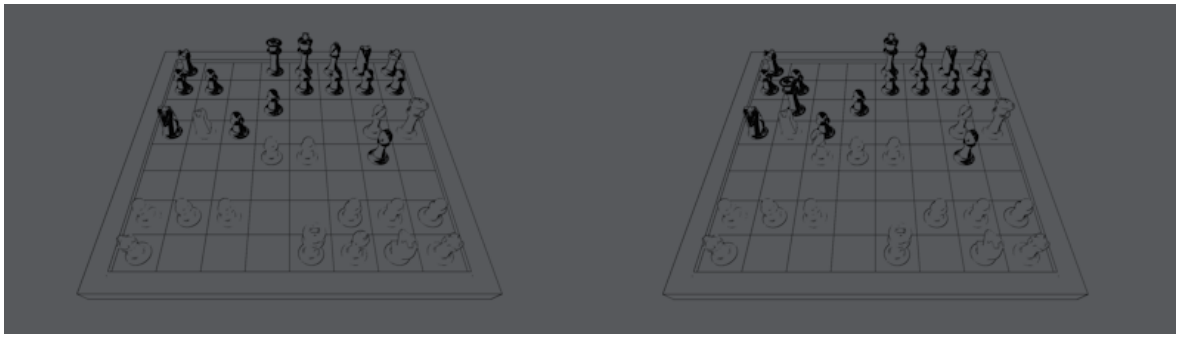

Figure 6.46: Two edge maps of two states of a chess gameplay where the pieces are mixed up

Figure 6.47: Two renders of two states of a chess gameplay where the pieces are mixed up. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge maps from Figure 6.46 as input, Resolution: 640x360, Seed (for all images): 2954826565

We get no changes in the results. The pieces stay one colour and the tiles on the chessfield are also not correctly sorted now but overall, we still have the same amount of pieces on the field. Some pieces have been merged a little bit as they conceal parts of other pieces or are too close together. The shapes of the pieces are not really recognisable as well.

To try to improve the shapes of the pieces we try to make the size of the generated image bigger. Instead of a size of 640px by 360px we double it to 1280px by 720px. With this adjustment Stable Diffusion will now need more time to render the image. With the same seed and otherwise the same settings we will redo the chess game. The render of the initial state is shown in Figure 6.48.

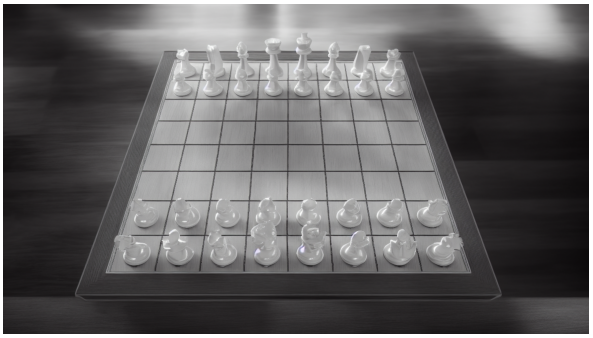

Figure 6.48: Render of the initial state of a gameplay. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge map from Figure 6.25 as input, Resolution: 1280x720, Seed: 2954826565

The first thing we can see is that with a change in resolution our result changes even though we use the same seed as before. We still have white pieces on both sides but the pieces are far more recognisable than with a lower resolution and in the initial state of the game no pieces are merged together. We take a look at the next four steps again which will be the same as with the lower resolution. The rendered results are shown in Figure 6.49.

Figure 6.49: Four subsequent renders of the first four moves of a gameplay. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge maps from Figure 6.43 as input, Resolution: 1280x720, Seed (for all images): 2954826565

The shapes of the moved pieces change a little bit, probably due to the change in the preprocessed canny edge image of the ControlNet. Otherwise, the pieces are

clearly separated and do not merge together. If we go again a few steps further we receive the results in Figure 6.50.

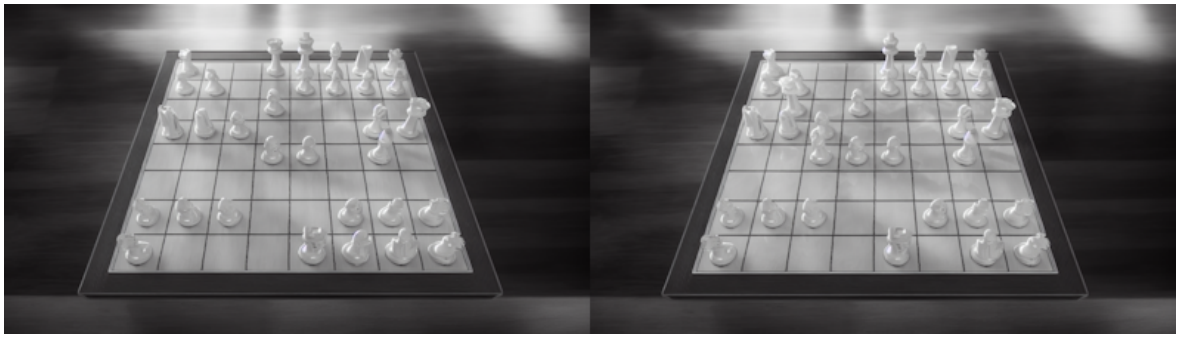

Figure 6.50: Two renders of two states of a chess gameplay where the pieces are mixed up. Prompt: ”chessfield on a table, b&w pieces, photo, dslr, hyperrealistic”, Negative prompt: ”ugly, disfigured, distorted, poorly drawn, out of frame”, ControlNet in edge mode with the edge maps from Figure 6.46 as input, Resolution: 1280x720, Seed (for all images): 2954826565

This time no pieces are merged together and the pieces all have a distinct shape. While we cannot play chess with it (even if the colours would be correct) as we are not able to tell which piece is a knight or which one is a bishop, we have clearer shapes as in the lower resolution renders. We might be able to improve this by increasing the resolution even further but we probably will not get the shapes to be perfectly clear and recognisable and as Stable Diffusion needs a lot of memory even now with the current resolution of 1280px by 720px we cannot increase the resolution too much.

In summary, we have found that while the depth mode of the ControlNet is useful to position objects in an image it has its limits when there are too many, too small and too complex things in it. On the other hand, while the edge mode was not that great in positioning of fewer, bigger and simpler objects in the citybuild game it works very good in finding the correct positions and shapes of the pieces. The resolution of the image takes an important part as well for more detailed objects. Maybe the shapes

could be further improved if the ControlNet would generate more precise lines. We can see in 6.51 that the canny edge map, which is made by the preprocessor of the ControlNet with the edge map from the game as input, is far less detailed than the Sobel's edge map which we get from Godot.

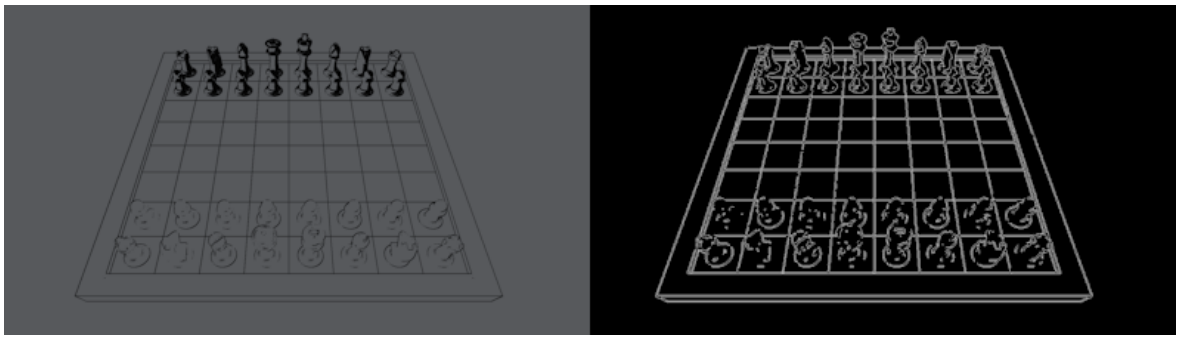

Figure 6.51: Comparison of the edge map generated in Godot and the edge map generated from the ControlNet preprocessor. Left: Sobel's edge map generated in Godot. Right: Canny edge map generated by the ControlNet preprocessor

6.2.3 Using img2img mode alone

In the Related Work chapter I mentioned a few approaches which used reference images to influence the appearance of colour. This can be done using the img2img mode of Stable Diffusion. In this section we try this out for ourselves by using the chessfield of the chess game as reference image and input to the img2img mode. With simply "chessfield" as prompt and no negative prompt we did not get great results with the txt2img mode. Using it now looks a bit different. We still use a resolution of 1280px by 720px for the generated image and no ControlNet. The results are shown in Figure 6.52.

Figure 6.52: Four generated images to try to generate a simple chessfield in the img2img mode. Prompt: ”chessfield”, Resolution: 1280x720, Seeds (from left to right): 2574691216, 2574691217, 2574691218, 2574691219

This looks exactly like a perfect chessfield. It also looks very much like our reference image. Therefore, the question might come up whether we need an exact reference image and will not have any possibilities changing for example the background. Adding a few more terms to the prompt and ending up with ”chessfield, on a table, in the woods” gives us the results in Figure 6.53.

Figure 6.53: Four generated images to try to generate a simple chessfield with background in the img2img mode. Prompt: ”chessfield, on a table, in the woods”, Resolution: 1280x720, Seeds (from left to right): 842646503, 842646504, 842646505, 842646506

The answer is: Fortunately not. It would be advised to give Stable Diffusion some directions on how the background should look like through the reference image, though. In the last generated image there is for example no forest in the background. We can also see that Stable Diffusion took the gray background of our reference image and used it as colour for the forest backgrounds. This means that we do not need a fully thought surrounding but only a rough outline of it using the colours we want it to have.

If generating a correct chessfield without any pieces from a reference image is this easy, this could work for the pieces as well. With the same prompt and the exact same settings but a reference image with the chess pieces we achieve the renders in Figure 6.54.

Figure 6.54: Four generated images to try to generate a chessfield with pieces and a background with the img2img mode. Prompt: ”chessfield, on a table, in the woods”, Resolution: 1280x720, Seeds (from left to right): 2379758637, 2379758638, 2379758639, 2379758640

The quality of the chessfield decreases very slightly in the first image where Stable Diffusion thinks that the black rook is a black tile. The pieces on the other hand are not in their correct position and neither have the correct colour. But maybe with the reference image we can tell Stable Diffusion the amount of black pieces and the amount of white pieces on the image. With ”chessfield, 16 black pieces, 16 white pieces” Stable Diffusion generates the images seen in Figure 6.55.

Figure 6.55: Four generated images to try to generate a chessfield with pieces with the correct amount of each colour with the img2img mode. Prompt: ”chessfield, 16 black pieces, 16 white pieces”, Resolution: 1280x720, Seeds (from left to right): 1950693596, 1950693597, 1950693598, 1950693599

We do not get the correct amount of pieces and the black pieces are also where white pieces should be and the other way around. We do have, however, white pieces and

black pieces in every image. This means that it is of no use telling Stable Diffusion how many pieces we have, but now it is very helpful to tell Stable Diffusion which colours we want to have without having the colour all over the image. There is still one problem to face. Some of the pieces are black and white. After removing the numbers in the prompt and adding ”two-coloured pieces” as negative prompt this problem is solved to the most part (see Figure 6.56). With a bit more sophisticated prompts we could probably improve the image further and get rid of the two-coloured pieces completely.

Figure 6.56: Four generated images to try to generate a chessfield with pieces with the img2img mode and a negative prompt. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, Resolution: 1280x720, Seeds (from left to right): 970008429, 970008430, 970008431, 970008432

6.2.4 Adding the ControlNet again

The img2img mode of Stable Diffusion gives us very promising results in terms of colour and reproducability of a given style. If we now add the ControlNet we might be able to tell Stable Diffusion exactly where we want the black pieces to be and where we want the white pieces to be. Since we already achieved a satisfying chessfield without any ControlNet we will simply move on to the next step and render the initial state of the chess game with the img2img mode and a ControlNet in edge mode, since this gave us the best results in the txt2img mode. For the prompt we will use ”chessfield, black pieces, white pieces” again and for the negative prompt we will also use ”two-coloured pieces” again. The results are shown in Figure 6.57.

Figure 6.57: Four generated images to try to generate a chessfield with pieces with the img2img mode. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in Figure 6.26 as input, Resolution: 1280x720, Seeds (from left to right): 4070255767, 1755636343, 2796249461, 1874466644

Due to the ControlNet in edge mode we achieve exactly 32 pieces again, with the reference image together with the ControlNet we are able to guess which piece is the king, the queen, a rook, any of the other pieces, and for the most part pieces that should be black are black and pieces that should be white are white. Only in one generated image the pieces do not match their colour. The chessfield, however, is not consistent anymore whenever a piece is on top of it. Unfortunately, we also cannot fix this by adding another ControlNet in edge mode with the edge map of only the chessfield. Using a weight of 1 for the ControlNet that uses the complete scene for the edge map and a weight of 0.5 for the ControlNet that only uses the chessfield for the edge map we cannot get the issue fixed (see Figure 6.58). Using a weight of 1 for both ControlNets even introduces us to more problems (see Figure 6.59). So, using weights of 1 for both ControlNets or a weight of 1 for the ControlNet that uses the complete scene for the edge map and a weight of 0.5 for the ControlNet that only uses the chessfield for the edge map we are not able to fix the issue.

Figure 6.58: Four generated images to try to generate a chessfield with pieces with the img2img mode. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in Figure 6.26 as input and a weight of 1, ControlNet in edge mode with the edge map in Figure 6.26 as input and a weight of 0.5, Resolution: 1280x720, Seeds (from left to right): 4197102029, 4197102030, 4197102031, 4197102032

Figure 6.59: Four generated images to try to generate a chessfield with pieces with the img2img mode. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, Two ControlNets in edge mode with the edge map in Figure 6.26 as input and a weight of 1, Resolution: 1280x720, Seeds (from left to right): 2748013571, 2748013572, 2748013573, 2748013574

Now, we will try to play a round of chess again. We have a fairly good field, we have black pieces where they should be and we have white pieces where they should be. The shapes of our pieces resemble the chess pieces close enough to be able to guess what they are. We start again with an initial render (see Figure 6.60) that will decide our seed throughout the rest of the game.

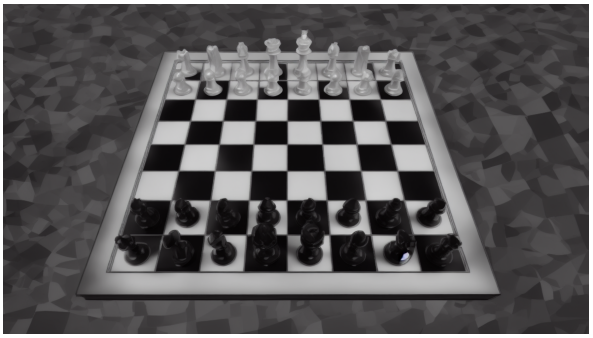

Figure 6.60: Render of the initial state of a gameplay. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map from Figure 6.25 as input, Resolution: 1280x720, Seed: 2704779737

With this seed we will again do the same four moves as in the game with the txt2img mode and ControlNet. We start by moving the left white knight, then we move the left black knight, then the white pawn in front of the white queen and then the black pawn in front of the black king. The results are shown in Figure 6.61.

Figure 6.61: Four subsequent renders of the first four moves of a gameplay. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge maps from Figure 6.43 as input, Resolution: 1280x720, Seed (for all images): 2704779737

Multiple things happen here. First of all, we can see again that the black knight is about to change to a white colour in between. In the end the knight is black again. The second thing we can see is that, although we use the same seed for every step, the results look quite different each time. Sometimes the white is like a silver-gray

colour, in other steps it is a perfect white. The surrounding is sometimes more structured and in other renders it is less detailed. To get more results, we play another four steps. This time we move the left white bishop, then the black queen, then the white pawn in front of the right white knight and lastly, the left black knight again. The ingame states for the moves are shown in Figure 6.62 and the results in Figure 6.64.

Figure 6.62: Four subsequent ingame states of the second four moves of a chess gameplay

Figure 6.63: Four subsequent depth maps of the second four moves of a chess gameplay

Figure 6.64: Four subsequent renders of the second four moves of a gameplay. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge maps from Figure 6.63 as input, Resolution: 1280x720, Seed (for all images): 2704779737

We can see that, as we mix the two colours more and more, Stable Diffusion cannot really decide anymore which pieces have to be black and which ones have to be white. This time the style of the overall image stayed relatively consistent which means that the surrounding is now also a kind of checkerboard pattern with mixed chess pieces on it and the white pieces and tiles are again the bright white colour. If we were to mix the pieces up even more we are not able anymore to tell which ones are white and which ones are black. After a few more steps we get to the render in Figure 6.65.

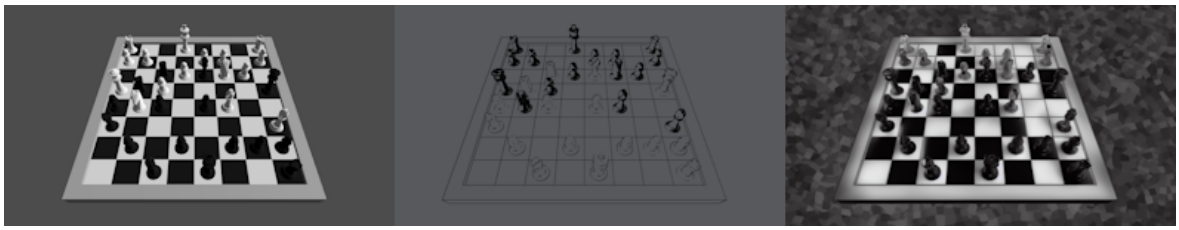

Figure 6.65: One render of a states of a chess gameplay where the pieces are mixed up. Left: the ingame state. Middle: The edge map. Right: The rendered result. Prompt: ”chessfield, black pieces, white pieces”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in the middle as input, Resolution: 1280x720, Seed: 2704779737

This is the problem with the reference image. It is only a reference. With the initial states it works very well as we have black pieces grouped together and white pieces grouped together. As the game advances we mix them up more and more until we do not have a group of white pieces and group of black pieces anymore but only a group of mixed pieces. The reference image is, however, a good step towards AI-rendered graphics.

6.2.5 Weighting terms

We can add something more to the ControlNets and reference images. Stable Diffusion has the possibility not only to weight ControlNets but also to weight terms inside

of the prompt and negative prompt. In Figure 6.64 we can see how the black queen started to switch the colour from black to white. Using the same prompt but with the terms ”black pieces” and ”white pieces” having a weight of 1.3 (we use three bracket pairs in the prompt for that) we can see that we have again a black queen standing there (see 6.66).

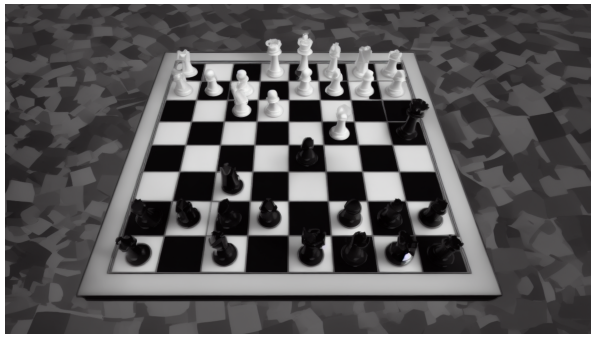

Figure 6.66: A render of the chess gameplay with the black queen’s colour restored. Prompt: ”chessfield, (((black pieces))), (((white pieces)))”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the third edge map in Figure 6.63 as input, Resolution: 1280x720, Seed: 2704779737

The problem is that with the next move of the white pawn the pawn now starts to switch its colours from white to black. (see Figure 6.67). Adding another three brackets and having therefore a weight of 1.6 on both weighted terms we do not get the pawn to turn back into white anymore. In addition, we have a step backwards since the pieces do not look great anymore and the colours get mixed further (see Figure 6.68).

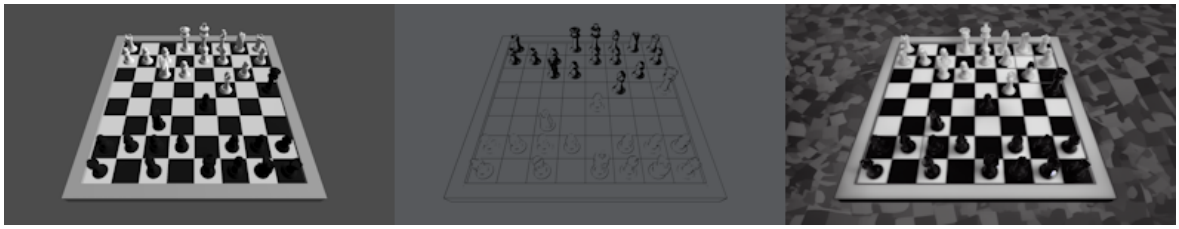

Figure 6.67: A render of the eighth step of the chess gameplay. Left: The ingame state. Middle: The resulting edge map. Right: The rendered image. Prompt: ”chessfield, (((black pieces))), (((white pieces)))”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in the middle as input, Resolution: 1280x720, Seed: 2704779737

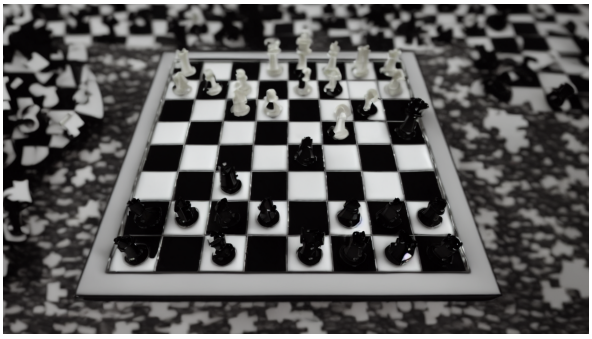

Figure 6.68: A render of the eighth step of the chess gameplay. Left: The ingame state. Middle: The resulting edge map. Right: The rendered image. Prompt: ”chessfield, ((((((black pieces)))))), ((((((white pieces))))))”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in Figure 6.67 as input, Resolution: 1280x720, Seed: 2704779737

It is also not necessarily possible to solve this question with more brackets. We can try using three brackets for every piece on the field. That equals 48 brackets per colour. We round that to 50 bracket pairs per colour and have 100 bracket pairs in the end. In Figure 6.69 we can see what this does to our image. We gain too much control over Stable Diffusion and the backwards diffusion process does not work anymore. If we use instead the prompt ”chessfield on a table, 16 black pieces,

16 white pieces”, however, we can play our game of chess further without much problems (see Figure 6.70). There are some pieces who get two-coloured again but they do not stay that way and turn to their correct colour in the next render. And even after taking some pieces from the field I still get the correct amount of white and black pieces. Further research needs to be conducted to find the reason why we can get promising results with one prompt but not with the other and if highly weighted terms are an option to generate better results.

Figure 6.69: Test to use 100 bracket pairs inside the prompt. Prompt: ”chessfield, ((black pieces)), ((white pieces)))”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in Figure 6.67 as input, Resolution: 1280x720, Seed: 2704779737

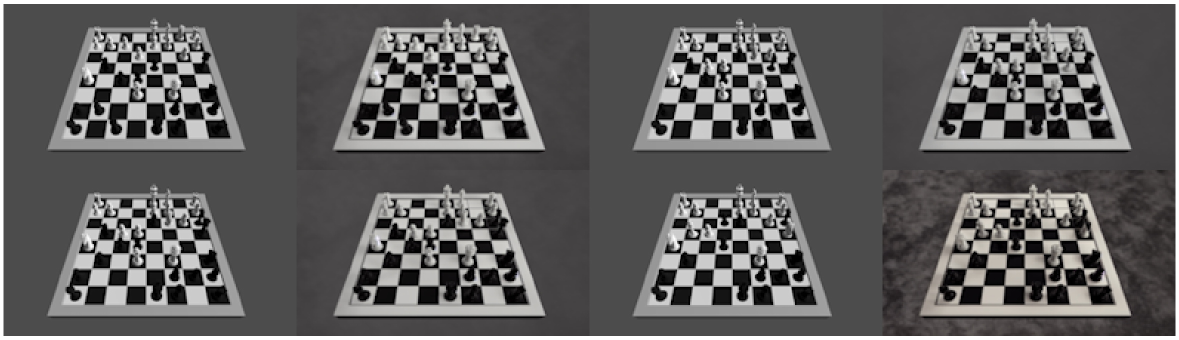

Figure 6.70: Four ingame states and their corresponding renders with 100 bracket pairs. Left is always the ingame state. Right is always the rendered image. Prompt: ”chessfield, ((black pieces)), ((white pieces))”, Negative prompt: ”two-coloured pieces”, ControlNet in edge mode with the edge map in Figure 6.67 as input, Resolution: 1280x720, Seed: 2704779737

6.3 Interpretation of the results

There is a lot we can take from these experiments. From finding good prompts over using a ControlNet for positioning and shapes to using reference images to give Stable Diffusion information about colour and style. Firstly, we have to think about the way we use prompts for video games. When using Stable Diffusion only with the txt2img mode prompts are the only way we can tell Stable Diffusion what we want to have in the generated image or what we do not want to have. Therefore, it is very important to put as much detail as possible inside of the prompt and the negative prompt. When using additional methods like a ControlNet or the img2img mode we do not need to explain every single detail inside of our prompt. In fact, the more layers of information we add, the less information we have to give through the prompt. In the end, we simply had the three main concepts of chess that we needed: a chessfield, black pieces and white pieces.

The next result we could see is that ControlNets are a great way to position the objects inside of the generated image. For larger and simpler objects we should use a ControlNet in depth mode and for smaller and more complex objects we should use a ControlNet in edge mode. Furthermore, the edge mode of the ControlNet is better suited if we want a more flexible surrounding. To get the best out of both worlds we can use a combination of two or more ControlNets and adjust their weights to fit our needs. If we want to give Stable Diffusion more freedom in positioning and more freedom for the surrounding we can give the depth mode a lower weight. If we want to have higher positioning precision we give the weight of the depth mode a higher value.

Depending on our style we have to explicitly bring the surrounding into the edge or depth map. As seen with the miniature wood house experiment we need to provide Stable Diffusion with more information on what we want for some styles while Stable Diffusion generates a surrounding by its own for other styles. Other than that we can use any style Stable Diffusion has had training on by specifying it in the prompt.

For motion pictures, video games and any other form that uses subsequent frames we need continuity between frames. Using one seed for every frame is a great way to ensure the overall feel of the image. The objects inside the image, however, will still change their appearance and the bigger the change in the scene is the bigger also is the change in the look of the rendered image. It is better to use a specific seed than to use random seeds, though.

To receive a more accurate result in terms of painting objects in a certain colour, using patterns (e.g. checkerboard pattern) or defining areas (e.g. the background in the chess game) we should use the img2img mode of Stable Diffusion with a reference image as input. The more we can incorporate objects used in the prompt to the reference image the better. Stable Diffusion understands a prompt better if it is also described in the reference image.

Lastly, we can use weighted terms to give Stable Diffusion more guidance in what it sees inside of a reference image. While it certainly is not advised to use a lot of bracket pairs inside of handcrafted prompts, as it can get messy very fast, this could be a helpful tool in more automated approaches.

To summarise the result interpretation:

1. The more layers of information the less information we need in our prompt
2. ControlNets are a great way for positioning objects
3. For better results we should use one ControlNet in depth mode for positioning and one in edge mode for details
4. For results that fit our needs we should incorporate everything into our information layers (like add in some lines into an edge map for a ControlNet).
5. For more continuity between frames find one seed that fits your needs best and use it for every render
6. Reference images are useful to define shapes and patterns, colours and areas
7. We should incorporate as much objects from the prompt as possible inside the reference image
8. Weighting terms are a possible way to get more accurate results in combination with a reference image

In the end, we can see that we have to adjust certain parameters depending on other parameters. While we get great surroundings with one style and approach we might get not so variant surroundings in other styles with the same approach. Using a ControlNet in depth mode for big and simple objects result in great positioning of these objects while it can get very chaotic with more, smaller and more complex objects.

6.4 Towards AI rendered adventure games

We have seen in this last chapter how we can use AI to render video games but there is one more pending question. We used round based games because we cannot render images faster than five seconds per frame and this would not be the best experience in a realtime environment, like in a shooter or adventure game. To get the best results we found that we need reference images with which we can tell Stable Diffusion where we want colours to be and how we want the image to look like in general. The question is how detailed this reference image needs to be? For the reference images of the chess game we simply used the rendered images from the Godot game engine. For a chess game this is no big deal as we only have a few 3D models we have to create. But if we would need to create the complete scene and all objects and elements in our world only to use it as a reference image when we could render them with the existing rendering pipeline, why should we bother with this approach that is not yet realtime ready, cannot yet generate perfect frames for a game and is not completely continuous between frames? The answer is that we do not need fully thought out reference images, we do not need the complete world and all the objects in it premade to use it as a reference. We can think of a reference image as something we can tell Stable Diffusion what is in this image. For this experiment we take some screenshots of landscapes and buildings from a Minecraft (Studios, 2009) world and use them as reference images.

6.4.1 The portal

The first screenshot is the structure pictured in Figure 6.71. We can imagine this to work as a sci-fi portal. We want it to be a red portal and therefore use ”a sci-fi portal, red portal” as prompt, no negative prompt, a ControlNet in depth mode with a weight of 1 with the same image as input and a ControlNet in edge mode with a weight of 0.3 and we get the results in Figure 6.72. We can still clearly see the block structure of Minecraft and we will not be able to get completely rid of it in this example as the reference image is made very close to the block structure. With a weight of 0.5 on

the ControlNet in edge mode and a weight of 0.3 of the ControlNet in depth mode we can get a little bit more details and we can decrease a little bit of the block structure in the surrounding. The results are shown in Figure 6.73

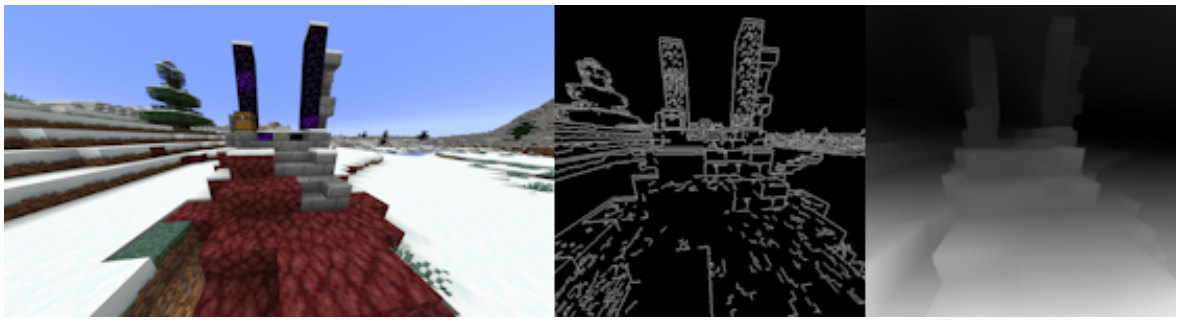

Figure 6.71: Left: The screenshot of a building structure in Minecraft. Middle: Generated edge map from the screenshot. Right: Generated depth map from the screenshot

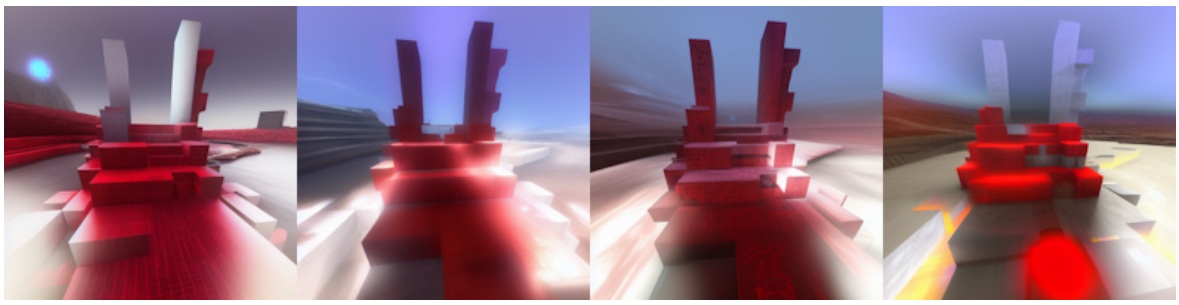

Figure 6.72: Four portals generated with the img2img mode. Prompt: ”a sci-fi portal, red portal”, ControlNet in edge mode and the screenshot in Figure 6.71 as input with weight 0.3, ControlNet in depth mode and the screenshot in Figure 6.71 as input with weight 1, resolution: 512x512, seeds (from left to right): 3158781618, 3158781619, 3158781620, 3158781621

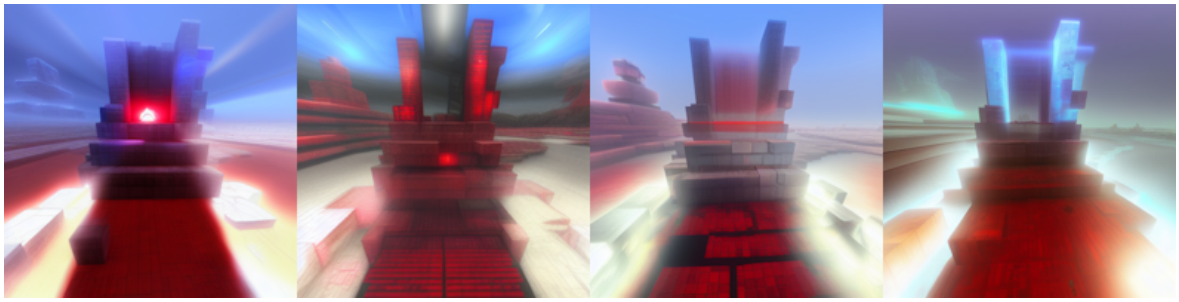

Figure 6.73: Four portals generated with the img2img mode. Prompt: ”a sci-fi portal, red portal”, ControlNet in edge mode and the screenshot in Figure 6.71 as input with weight 0.5, ControlNet in depth mode and the screenshot in Figure 6.71 as input with weight 0.3, resolution: 512x512, seeds (from left to right): 3334137632, 3334137633, 3334137634, 3334137635

6.4.2 The igloo

For the next test we use a custom built igloo on a snow field (see Figure 6.74) and we simply want it to be exactly the same thing but not in Minecraft but in a cartoon style. We use ”an igloo, on a mountain, cartoon” as the prompt, again no negative prompt and a weight of 0.5 on both ControlNets. The results are shown in Figure 6.75. Since we have no blocky structures in the foreground we do not have them in the image as well except for the igloo itself. If we suddenly want it to be a boat we can change the prompt to ”a boat on very bright water” and get a blue boat on very bright water where we had an igloo before. Figure 6.76 show the results.

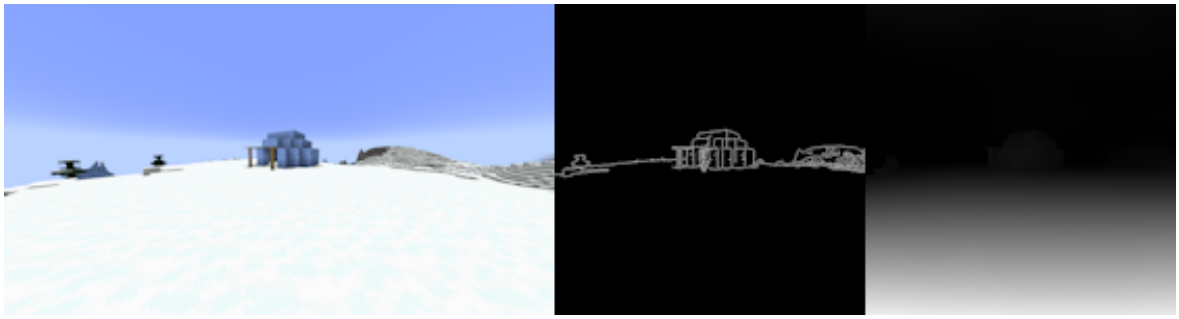

Figure 6.74: Left: The screenshot of an igloo in Minecraft. Middle: Generated edge map from the screenshot. Right: Generated depth map from the screenshot

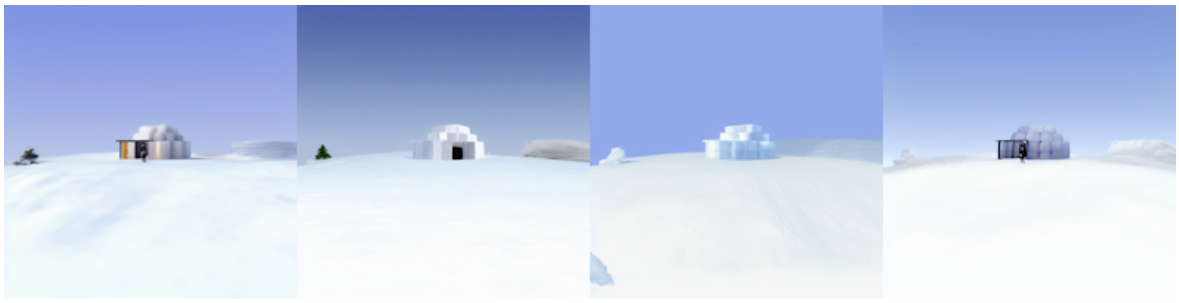

Figure 6.75: Four igloos generated with the img2img mode. Prompt: ”an igloo, on a mountain, cartoon”, ControlNet in edge mode and the screenshot in Figure 6.74 as input with weight 0.5, ControlNet in depth mode and the screenshot in Figure 6.74 as input with weight 0.5, resolution: 512x512, seeds (from left to right): 2810828231, 2810828232, 2810828233, 2810828234

Figure 6.76: Four boats generated with the img2img mode with an igloo building as reference. Prompt: ”a boat on very bright water”, ControlNet in edge mode and the screenshot in Figure 6.74 as input with weight 0.5, ControlNet in depth mode and the screenshot in Figure 6.74 as input with weight 0.5, resolution: 512x512, seeds (from left to right): 250899574, 250899575, 250899576, 250899577

6.4.3 The dungeon entrance

We have also a screenshot of an entrance to a cave in the Minecraft world (see Figure 6.77) and we can imagine this to be the an entrance to a dungeon in the style of Ghibli. So, we use the reference image with ”a dungeon in the style of ghibli” as prompt, no negative prompt, a weight of 0.5 for the ControlNet in edge mode and a weight of 1 for the ControlNet in depth mode and get the results in Figure 6.78. As we have a lot going on in this scene we do not get the blocky structures of Minecraft and the entrance is exactly where the entrance to the cave was. We can try to change the colour of the glow of the entrance in the first image. Therefore, we take this as the new reference image and leave everything as it is. Moreover, we add ”red glowing entrance” to the prompt and get the results in Figure 6.79. Not all entrances glow red now. To improve this we could use the inpaint and inpaint sketch functions of the WebUI and paint with the red colour over it. For another image we can try to change the entrance to have a green door and paint with green over the entrance in the reference image. In Figure 6.80 we can see the result.

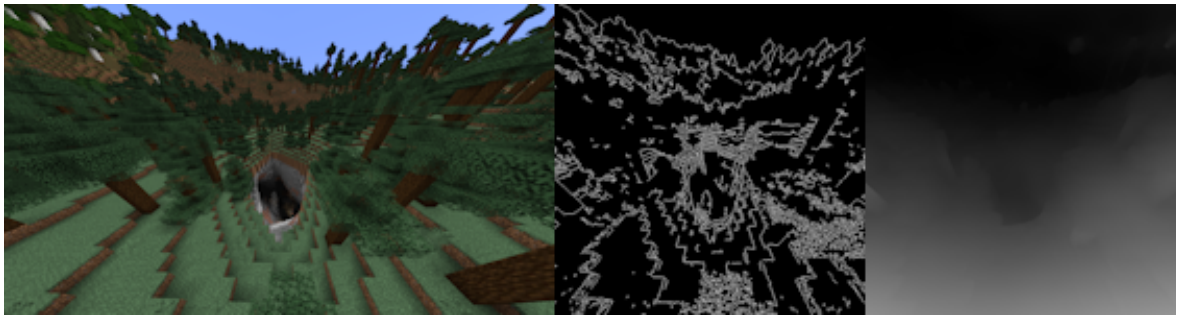

Figure 6.77: Left: The screenshot of an entrance to a cave in Minecraft. Middle: Generated edge map from the screenshot. Right: Generated depth map from the screenshot

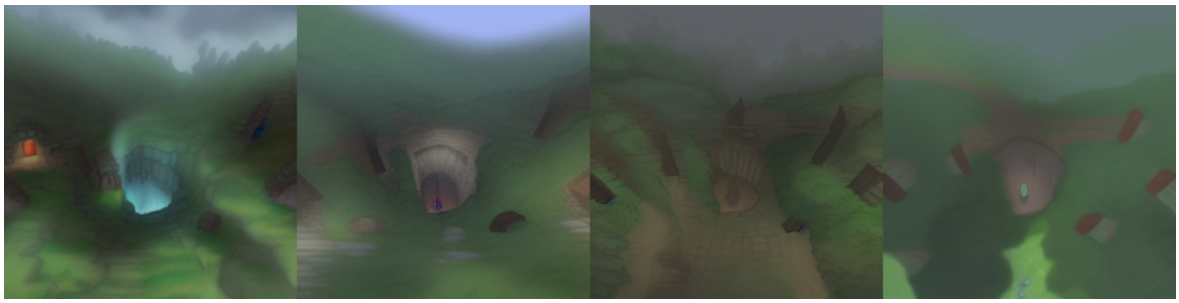

Figure 6.78: Four dungeons generated with the img2img mode. Prompt: ”a dungeon in the style of ghibli”, ControlNet in edge mode and the screenshot in Figure 6.77 as input with weight 0.5, ControlNet in depth mode and the screenshot in Figure 6.77 as input with weight 1, resolution: 512x512, seeds (from left to right): 4020720994, 4020720995, 4020720996, 4020720997

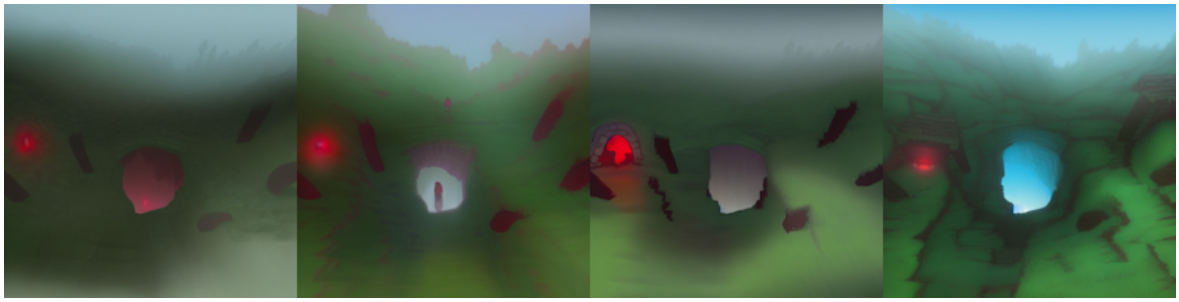

Figure 6.79: Four dungeons with a red glow generated with the img2img mode. Prompt: ”a dungeon in the style of ghibli, red glowing entrance”, ControlNet in edge mode and the screenshot in Figure 6.77 as input with weight 0.5, ControlNet in depth mode and the screenshot in Figure 6.77 as input with weight 1, resolution: 512x512, seeds (from left to right): 3252043576, 3252043577, 3252043578, 3252043579

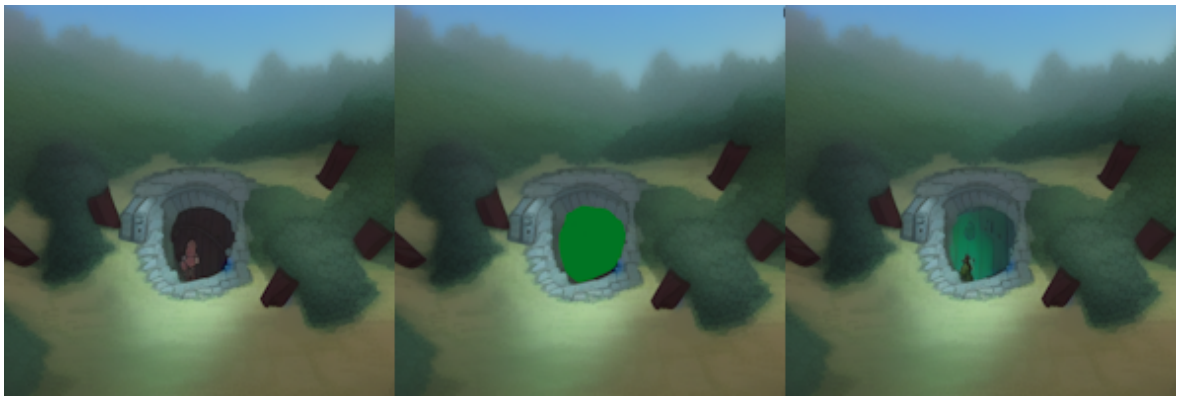

Figure 6.80: Left: The reference image which I got when using the screenshot in Figure 6.77. Middle: The reference image with a green inpaint sketch on it used as the new reference image with the inpaint function. Right: The resulting entrance with green door

6.4.4 AI rendered adventure games summary

We can see we do not need exact reference images for the AI, in fact anything can do as a reference image. To get the best results we have to lead the AI towards

our goal. The idea with the ship was very far fetched for example. The approach to change the colour of the light of the dungeon entrance is a far more useful idea. There is still a lot to do to find how the best reference images look like, but as a start we can say anything that has some relation to the topic is a good reference.

Chapter 7

Conclusion

In this thesis I proposed a new way of generating images for video games by incorporation generative AI. The current state of the art is using complex 3D models and calculating for every pixel how they should look like. Using AI we would not need the ever growing rendering pipeline anymore and could replace it by add-ons for game engines that can access the generative AI. There are a lot of challenges to overcome before we can utilise AI for the rendering of video games. At the time of writing generating images using AI is computationally extremely expensive and instead of being able to render images with a framerate of 30 or more frames per second we have to calculate the render time in seconds per frame. We also need a sufficient way of telling the AI what it should generate on the frame. In this thesis we used several functions of the Stable Diffusion AI. We found that while Stable Diffusion does not know every concept from a prompt only, we can use one or more ControlNets to control the positioning of objects and we can use reference images to give the AI a more complete idea what we want to see in the generated image. We also found that the prompt is not the most important part to influence the generation process. While the prompt sets the overall topic of what should be seen in the generated image, the ControlNet can control the positions of the objects and the reference image can control the shapes, patterns and colours. To be able to run a complete image sequence we have to make sure that the continuity between the frames does not get broken. While we can use seeds to keep the overall tone of the

image comparatively equal we were not able to always keep the same surroundings or shapes of the objects in the scene. While we have the term ”engineering” inside of prompt engineering we do not build prompts based on given formulas or something similar. Instead, we try to find the best prompt by trial and error and this is also the way we treated the approaches to find ways to improve the images. In the end, we could find an approach that lets us play a game of chess with AI-generated renders for the individual states. Since the field of generative AI is relatively new, and not much research was made into using it for video game rendering, we can consider this as a quite successful series of experiments which showed a lot of the strengths and weaknesses of the different parameters we can change.

Chapter 8

Future Research

There is so much more you can try to get better results out of Stable Diffusion. From conducting more research about prompt engineering over trying new features up to using image manipulation techniques to computationally composite multiple renders or images for ControlNet modes. One future research topic could for example be to use regional prompting to associate a colour to a specific region inside of the image. With regional prompting we could also describe objects in that region more detailed. Instead of describing chess as a chessfield with pieces we could for example describe every single piece. Another approach that can lead to better images is using a ControlNet in segmentation mode and using a segmentation map that segments the chess pieces by colour. In addition, with a ControlNet in edge mode it could control the positions while the segmentation mode could control the colours of the pieces. Another idea could be to train a model specifically with chess images for the chess game or with specific images that we can use in a game. We could then try if the results we get from the self-trained model is better than the results from a more general model. Cameras in video games are not static as well and one further research topic could be to use the inpainting functionality of Stable Diffusion for a moving background. You could create for example a 2D side scroller where the AI inpaints new areas as we walk to the left or right. One last idea is whether it is possible to use a ControlNet with the pose estimation mode to generate characters into the images or even animate them. There are probably much more approaches for

future research topics and as this is a very new field of research it will still take a few years until we see the first games with fully AI generated graphics but we can use these tools today to create video games and improve them on the go.

Bibliography

Douglas, A. S. (1952). OXO. Retrieved August 18, 2023, from https://www.dcs.warwick.ac.uk/~edsac/ (cit. on p. 3).

Russell, S., Graetz, M., Wiitanen, W., Saunders, B., & Piner, S. (1962). Spacewar! Retrieved August 18, 2023, from https://www.computerspacefan.com/SpaceWarSim.htm (cit. on p. 3).

Alcorn, A. (1972). Pong. Retrieved August 18, 2023, from https://en.wikipedia.org/wiki/Pong (cit. on p. 3).

Whitted, T. (1980). An Improved Illumination Model for Shaded Display. *Commun. ACM*, *23*(6), 343–349. https://doi.org/10.1145/358876.358882 (cit. on p. 4).

Nintendo. (1986). The Legend of Zelda. Retrieved August 18, 2023, from https://www.nintendo.de/Spiele/NES/The-Legend-of-Zelda-796345.html (cit. on p. 4).

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533–536 (cit. on p. 6).

LLC, L. G. (1990). The Secret of Monkey Island. Retrieved August 18, 2023, from https://store.steampowered.com/app/32360/The_Secret_of_Monkey_Island_Special_Edition/ (cit. on p. 4).

Jarzynski, C. (1997). Equilibrium free-energy differences from nonequilibrium measurements: A master-equation approach. *Phys. Rev. E*, *56*, 5018–5035. https://doi.org/10.1103/PhysRevE.56.5018 (cit. on p. 7).

Studios, M. (2009). Minecraft. Retrieved September 12, 2023, from https://www.minecraft.net/ (cit. on p. 73).

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Networks. (Cit. on pp. 5, 6).

Dinh, L., Krueger, D., & Bengio, Y. (2015). NICE: Non-linear Independent Components Estimation. (Cit. on p. 6).

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., & Ganguli, S. (2015, July). Deep Unsupervised Learning using Nonequilibrium Thermodynamics. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (pp. 2256–2265, Vol. 37). PMLR. https://proceedings.mlr.press/v37/sohl-dickstein15.html (cit. on p. 7).

Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2017). Density estimation using Real NVP. (Cit. on p. 6).

Boden, M. A. (2018, August). *Artificial Intelligence: A Very Short Introduction*. Oxford University Press. https://doi.org/10.1093/actrade/9780199602919.001.0001 (cit. on p. 5).

Kingma, D. P., & Dhariwal, P. (2018). Glow: Generative Flow with Invertible 1x1 Convolutions. (Cit. on p. 6).

Wang, T.-C., Liu, M.-Y., Zhu, J.-Y., Liu, G., Tao, A., Kautz, J., & Catanzaro, B. (2018a). Researrch at NVIDIA: The First Interactive AI Rendered Virtual World. https://www.youtube.com/watch?v=ayPqjPekn7g (cit. on p. 12).

Wang, T.-C., Liu, M.-Y., Zhu, J.-Y., Liu, G., Tao, A., Kautz, J., & Catanzaro, B. (2018b). Video-to-Video Synthesis. *Advances in Neural Information Processing Systems (NeurIPS)*. https://doi.org/https://doi.org/10.48550/arXiv.1808.06601 (cit. on p. 12).

Nvidia. (2019). Quake 2 RTX. Retrieved August 18, 2023, from https://www.nvidia.com/en-us/geforce/news/quake-ii-rtx-ray-traced-remaster-out-now-for-free/ (cit. on p. 4).

Child, R. (2021). Very Deep VAEs Generalize Autoregressive Models and Can Outperform Them on Images. (Cit. on p. 6).

Dhariwal, P., & Nichol, A. (2021). Diffusion Models Beat GANs on Image Synthesis. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, & J. W. Vaughan (Eds.),

Advances in neural information processing systems (pp. 8780–8794, Vol. 34). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2021/file/49ad23d1ec9fa4bd8d77d02681df5cfa-Paper.pdf (cit. on p. 7).

Kingma, D., Salimans, T., Poole, B., & Ho, J. (2021). Variational Diffusion Models. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, & J. W. Vaughan (Eds.), *Advances in Neural Information Processing Systems* (pp. 21696–21707, Vol. 34). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2021/file/b578f2a52a0229873fefc2a4b06377fa-Paper.pdf (cit. on p. 7).

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021, 18–24 Jul). Learning Transferable Visual Models From Natural Language Supervision. In M. Meila & T. Zhang (Eds.), *Proceedings of the 38th international conference on machine learning* (pp. 8748–8763, Vol. 139). PMLR. https://proceedings.mlr.press/v139/radford21a.html (cit. on p. 8).

Reynolds, L., & McDonell, K. (2021). Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. (Cit. on p. 13).

AUTOMATIC1111. (2022). Stable Diffusion Web UI. https://github.com/AUTOMATIC1111/stable-diffusion-webui (cit. on pp. 10, 15).

Dev, S. (2022). Sygil WebUI. https://github.com/Sygil-Dev/Sygil-WebUI (cit. on p. 10).

Kingma, D. P., & Welling, M. (2022). Auto-Encoding Variational Bayes. (Cit. on p. 6).

Liu, V., & Chilton, L. B. (2022). Design Guidelines for Prompt Engineering Text-to-Image Generative Models. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3491102.3501825 (cit. on p. 13).

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). High-Resolution Image Synthesis with Latent Diffusion Models. https://doi.org/10.48550/arXiv.2112.10752 (cit. on pp. 6–9, 15).

Schuhmann, C., Beaumont, R., Vencu, R., Gordon, C., Wightman, R., Cherti, M., Coombes, T., Katta, A., Mullis, C., Wortsman, M., Schramowski, P., Kundurthy, S., Crowson, K., Schmidt, L., Kaczmarczyk, R., & Jitsev, J. (2022). LAION-5B:

An open large-scale dataset for training next generation image-text models. (Cit. on p. 9).

Bozesan, A. (2023). How to use ControlNet in your AI Art - Stable Diffusion Tutorial 2023. Retrieved September 8, 2023, from https://www.youtube.com/watch?v=dLM2Gz7GR44 (cit. on p. 14).

comfyanonymous. (2023). ComfyUI. https://github.com/comfyanonymous/ComfyUI (cit. on p. 10).

Hofmann, N., Hasselgren, J., & Munkberg, J. (2023). Joint Neural Denoising of Surfaces and Volumes. *Proc. ACM Comput. Graph. Interact. Tech.*, *6*(1). https://doi.org/10.1145/3585497 (cit. on p. 5).

Juan Linietsky, G. E. c., Ariel Manzur. (2023a). Advanced post-processing. Retrieved September 7, 2023, from https://docs.godotengine.org/en/stable/tutorials/shaders/advanced_postprocessing.html (cit. on p. 19).

Juan Linietsky, G. E. c., Ariel Manzur. (2023b). Godot Game Engine. https://godotengine.org/ (cit. on p. 15).

Kamph, S. (2023). Control Light in AI Images. Retrieved September 8, 2023, from https://www.youtube.com/watch?v=_xHC3bT5GBU (cit. on p. 13).

Katri, C. (2023). Dream Textures. https://github.com/carson-katri/dream-textures (cit. on p. 1).

Mikubill. (2023). ControlNet for Stable Diffusion WebUI. https://github.com/Mikubill/sd-webui-controlnet (cit. on p. 15).

Sarikas, O. (2023). COLOR CONTROL! - NEW ControlNET Method for A1111. Retrieved September 8, 2023, from https://www.youtube.com/watch?v=u-jvjP2k-RU (cit. on p. 13).

Steinberg, S., Ramamoorthi, R., Bitterli, B., d'Eon, E., Yan, L.-Q., & Pharr, M. (2023). A Generalized Ray Formulation For Wave-Optics Rendering. https://doi.org/https://doi.org/10.48550/arXiv.2303.15762 (cit. on p. 5).

Zhang, L., & Agrawala, M. (2023). Adding Conditional Control to Text-to-Image Diffusion Models. (Cit. on p. 11).

Nintendo. (n.d.). Super Nintendo Entertainment System. Retrieved August 18, 2023, from https://www.nintendo.de/Hardware/Unternehmensgeschichte/Super-Nintendo/Super-Nintendo-627040.html (cit. on p. 4).

runwayml. (n.d.). Stable Diffusion v1.5. Retrieved September 12, 2023, from https://huggingface.co/runwayml/stable-diffusion-v1-5 (cit. on p. 15).

Vijn, J. (n.d.). *Mode 7 Part 1*. Retrieved August 18, 2023, from https://www.coranac.com/tonc/text/mode7.htm (cit. on p. 4).

Winter, D. (n.d.). *Arcade Pong*. Retrieved August 18, 2023, from https://www.pong-story.com/arcade.htm (cit. on p. 3).

List of Figures

Appendices

Appendix A

Godot addon

A.1 Base64 decoder

```
func decodeBase64(data: String):
    var final_buffer = []
    var counter = 0
    for i in range(data.length() / 4):
        var sub_data = data.substr(counter, 4)
        var paddingCounter = sub_data.count("=")

        # get binary representation of base64 encoded values
        var bin_buffer = []
        for char in sub_data:
            if char != "=":
                bin_buffer.append(SD_Renderer.
                    from_b64_database[char])
            else:
                bin_buffer.append(0b000000)

        # create a number with lenght of 24 (4 * 6 or 3 * 8)
```

```
        var bin_number = 0b0
        for idx in range(bin_buffer.size()):
            var b = (bin_buffer[idx] << (18 - (idx * 6)))
            bin_number = bin_number | b

        # get decoded values in binary
        var v1 = ((bin_number & (0b11111111 << 16)) >> 16)
        var v2 = ((bin_number & (0b11111111 << 8)) >> 8)
        var v3 = bin_number & 0b11111111

        # add values to final buffer (keep base64 padding in
          mind)
        final_buffer.append(v1)
        if paddingCounter < 2:
            final_buffer.append(v2)
        if paddingCounter < 1:
            final_buffer.append(v3)

        counter += 4

    return PackedByteArray(final_buffer)
```

A.2 Base64 encoder

```
func encodeBase64(data: Image):
  var buffer_data = data.save_png_to_buffer()
  var encoded_string = ""
  var encoded_array = []

  var counter = 0
```

```
for i in range(buffer_data.size() / 3):

  # create a 24 bit number from three values
  var long_byte = 0b0
  var padding = 0
  if buffer_data.size() < (counter + 1):
    long_byte = long_byte | (buffer_data[counter] << 16)
    padding = 2
  elif buffer_data.size() < (counter + 2):
    long_byte = long_byte | (buffer_data[counter] << 16) |
      (buffer_data[counter + 1] << 8)
    padding = 1
  else:
    long_byte = long_byte | (buffer_data[counter] << 16) |
      (buffer_data[counter + 1] << 8) | (buffer_data[
      counter + 2])

  # Add the four chars to the encoded string
  encoded_array.append(SD_Renderer.to_b64_database[((
     long_byte & (0b111111 << 18)) >> 18)])
  encoded_array.append(SD_Renderer.to_b64_database[((
     long_byte & (0b111111 << 12)) >> 12)])
  if padding == 2:
    #encoded_string += "=="
    encoded_array.append("=")
    encoded_array.append("=")
  else:
    encoded_array.append(SD_Renderer.to_b64_database[((
       long_byte & (0b111111 << 6)) >> 6)])
    if padding == 1:
      #encoded_string += "="
```

```
            encoded_array.append("=")
        else:
            encoded_array.append(SD_Renderer.to_b64_database[(
                long_byte & 0b111111)])

    counter += 3

  encoded_string = "".join(PackedStringArray(encoded_array))
  return encoded_string
```

A.3 API call

```
# create request to the stable diffusion api
var api_url = url + "/sdapi/v1/txt2img"
var json = generateJsonFromData()
var headers = ["Content-Type: application/json"]
requestNode.request(api_url, headers, HTTPClient.METHOD_POST,
    json)
```

A.4 Creating the JSON object with our data

```
func generateJsonFromData(scene_texture, model):
  var dataAsJson = null
  if model == SD_Renderer.Model_Options.txt2img || scene.
     number_of_renders == 0:
    dataAsJson = JSON.new().stringify({
      "prompt": scene.sd_prompt,
      "negative_prompt": scene.sd_negative_prompt,
      "steps": steps,
```

```
    "width": 1280,
    "height": 720,
    "seed": 2954826565,
    "alwayson_scripts": {
      "controlnet": {
        "args": [
          {
            "input_image": encodeBase64(scene_texture),
            "module": SD_Renderer.cn_options_values[scene.
               controlnet_option][0],
            "model": SD_Renderer.cn_options_values[scene.
               controlnet_option][1],
            "weight": 1.0,
          }
        ]
      }
    }
  })

else:
  dataAsJson = JSON.new().stringify({
    "init_images": [
      encodeBase64(scene.get_image())
    ],
    "prompt": scene.sd_prompt,
    "negative_prompt": scene.sd_negative_prompt,
    "steps": steps,
    "width": 1280,
    "height": 720,
    "seed": 2704779737,
    "alwayson_scripts": {
```

```
        "controlnet": {
          "args": [
            {
              "input_image": encodeBase64(scene_texture),
              "module": SD_Renderer.cn_options_values[scene.
                 controlnet_option][0],
              "model": SD_Renderer.cn_options_values[scene.
                 controlnet_option][1],
              "weight": 1.0,
            }
          ]
        }
      }
    })

  return dataAsJson
```

A.5 API request complete

```
func _on_request_completed(result, response_code, headers,
    body):
  # create json object from body
  var json_object = JSON.new()
  json_object.parse(body.get_string_from_utf8())
  var response = json_object.get_data()

  # decode base64 encoded image
  var data = decodeBase64(response["images"][0])

  # create image from decoded buffer
```

```
  var image = Image.new()
  var error = image.load_png_from_buffer(data)
  if error != OK:
    push_error("Couldn't load image")

  # create texture from image and add it to root
  var texture = ImageTexture.create_from_image(image)
  texture_rect.texture = texture
```

A.6 Initialization for rendering the control images

```
func _ready():
  if not Engine.is_editor_hint():
    subviewport = SubViewport.new()
    sv_camera = Camera3D.new()
    sv_mesh = MeshInstance3D.new()
    sv_texture = TextureRect.new()

    sv_mesh.mesh = QuadMesh.new()
    sv_mesh.mesh.size = Vector2(2.0, 2.0)
    sv_mesh.mesh.material = ShaderMaterial.new()
    sv_mesh.mesh.material.shader = depth_shader if (
       controlnet_option == SD_Renderer.CN_Options.DEPTH)
       else sobel_shader if (controlnet_option == SD_Renderer
       .CN_Options.EDGE) else thresh_shader
    sv_mesh.mesh.flip_faces = true
    for i in range(20):
      sv_mesh.set_layer_mask_value(i, false)
    sv_mesh.set_layer_mask_value(19, true)
    sv_mesh.set_layer_mask_value(20, true)
```

```
    print(depth_shader)

    sv_camera.add_child(sv_mesh)
    for i in range(20):
      sv_camera.set_cull_mask_value(i, false)
    sv_camera.set_cull_mask_value(19, true)
    sv_camera.set_cull_mask_value(20, true)
    sv_camera.environment = camera_env
    sv_camera.current = true
    sv_camera.position = camera.position
    sv_camera.rotation_degrees = camera.rotation_degrees
    sv_camera.fov = camera.fov

    subviewport.add_child(sv_camera)
    subviewport.size = Vector2i(1920, 1080)
    subviewport.render_target_update_mode = SubViewport.
      UPDATE_ALWAYS
    subviewport.transparent_bg = true

    self.add_child(subviewport)

func get_texture(current_render):
  return subviewport.get_texture().get_image()
```

A.7 Depth shader

```
shader_type spatial;

uniform sampler2D depth_texture : source_color,
  hint_depth_texture;
```

```
varying mat4 CAMERA;

void vertex() {
  POSITION = vec4(VERTEX.xyz, 1.0);
  CAMERA = INV_VIEW_MATRIX;
}

void fragment() {
  float depth = texture(depth_texture, SCREEN_UV).x;
  vec3 ndc = vec3(SCREEN_UV * 2.0 - 1.0, depth);
  vec4 view = INV_PROJECTION_MATRIX * vec4(ndc, 1.0);
  view.xyz /= view.w;
  float linear_depth = -view.z;
  ALBEDO = vec3(linear_depth / 2.0);
}
```

A.8 Edge shader

```
 shader_type spatial;

uniform sampler2D depth_texture : source_color,
   hint_depth_texture;
uniform sampler2D screen_texture : hint_screen_texture,
   repeat_disable, filter_nearest;
uniform vec2 resolution = vec2(384, 216);

void vertex() {
  POSITION = vec4(VERTEX.xyz, 1.0);
}
```

```
void fragment() {

  vec2 SCREEN_PIXEL_SIZE = 1.0 / VIEWPORT_SIZE;

  vec3 col = vec3(0.5);
  col += texture(screen_texture, SCREEN_UV + vec2(-
    SCREEN_PIXEL_SIZE.x, -SCREEN_PIXEL_SIZE.y)).xyz * -1.0;
  col += texture(screen_texture, SCREEN_UV + vec2(-
    SCREEN_PIXEL_SIZE.x, 0.0)).xyz * -2.0;
  col += texture(screen_texture, SCREEN_UV + vec2(-
    SCREEN_PIXEL_SIZE.x, SCREEN_PIXEL_SIZE.y)).xyz * -1.0;

  col += texture(screen_texture, SCREEN_UV + vec2(
    SCREEN_PIXEL_SIZE.x, -SCREEN_PIXEL_SIZE.y)).xyz * 1.0;
  col += texture(screen_texture, SCREEN_UV + vec2(
    SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 2.0;
  col += texture(screen_texture, SCREEN_UV + vec2(
    SCREEN_PIXEL_SIZE.x, SCREEN_PIXEL_SIZE.y)).xyz * 1.0;


  col += texture(screen_texture, SCREEN_UV + vec2(-
    SCREEN_PIXEL_SIZE.x, -SCREEN_PIXEL_SIZE.y)).xyz * -1.0;
  col += texture(screen_texture, SCREEN_UV + vec2(0.0, -
    SCREEN_PIXEL_SIZE.y)).xyz * -2.0;
  col += texture(screen_texture, SCREEN_UV + vec2(
    SCREEN_PIXEL_SIZE.x, -SCREEN_PIXEL_SIZE.y)).xyz * -1.0;

  col += texture(screen_texture, SCREEN_UV + vec2(-
    SCREEN_PIXEL_SIZE.x, SCREEN_PIXEL_SIZE.y)).xyz * 1.0;
  col += texture(screen_texture, SCREEN_UV + vec2(0.0,
```

```
    SCREEN_PIXEL_SIZE.y)).xyz * 2.0;
  col += texture(screen_texture, SCREEN_UV + vec2(
    SCREEN_PIXEL_SIZE.x, SCREEN_PIXEL_SIZE.y)).xyz * 1.0;

  col = vec3((col.x + col.y + col.z) / 3.0);

  if (col.x > 0.48 && col.x < 0.52) ALBEDO = vec3(1.0);
  else ALBEDO = vec3(0.0);
}
```

Appendix B

Chess

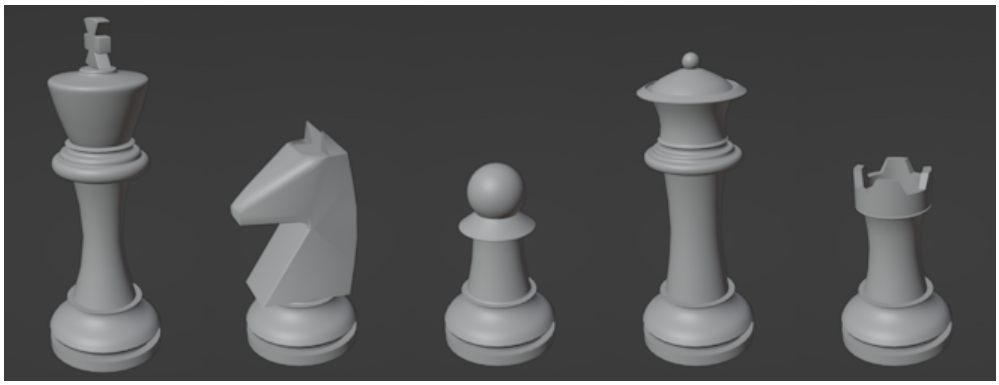

Figure B.1: 3D models for the chess game