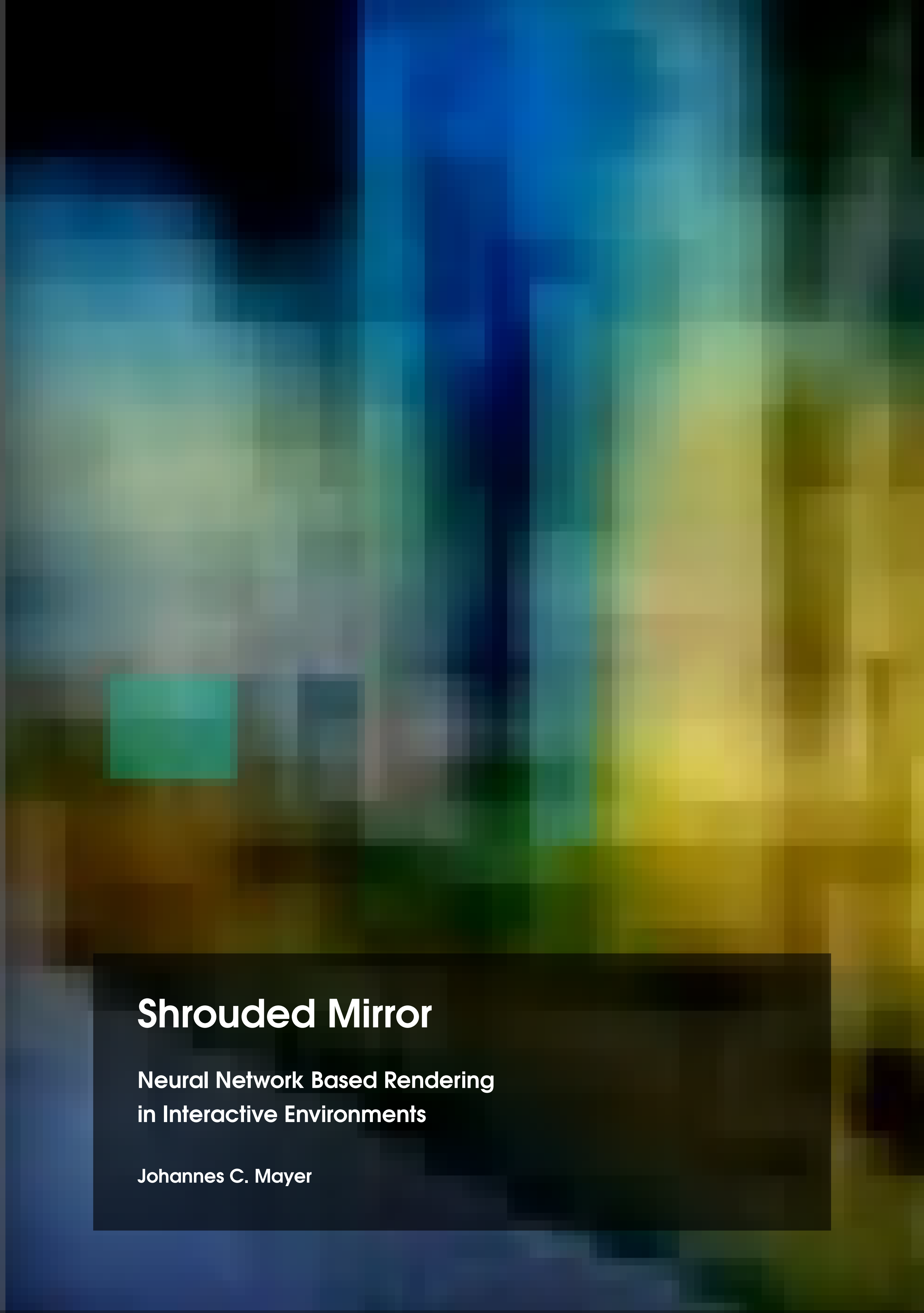


Shrouded Mirror

Neural Network Based Rendering in Interactive Environments

Johannes C. Mayer

Bachelor thesis booklet in the studies of GAME DESIGN
The work constitutes a technical design work piece

Shrouded Mirror

Neural Network Based Rendering in Interactive Environments

Submitted by
Johannes C. Mayer, 553087
on January 30, 2019

First Examiner: Prof. Thomas Bremer
Second Examiner: Prof. Susanne Brandhorst

Hochschule für Technik und Wirtschaft Berlin
Fachbereich Gestaltung und Kultur

Exposé

In my bachelor thesis I want to evaluate how neural networks can be used in an interactive context to visualize an environments state. For this the project is structured into three phases. First create a simple implementation of the algorithms required, then explore possible applications, third develop the most promising application further.

Neural networks have the ability to learn an abstract representations of the data they are being trained on. This representation can then be used to generate new output. In this work a network is used that is trained on data pairs that consist of an Image and the corresponding coordinates where the image was taken. The network can now, given a set of scene coordinates, render an image that approximates what a camera would render when placed at the provided coordinates. This means that this technique can be used to create a second visual representation of an environment inside a game engine.

Interesting possibilities open when integrating this network into an interactive environment. One example would be, that the player only sees the output of the network while the underlying "real" state of the environment is hidden from him.

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Scope 2

2 Background **3**
- 2.1 Neural networks 3
  - 2.1.1 Neuron 3
  - 2.1.2 Multilayer network 4
  - 2.1.3 Training 6
- 2.2 Differential calculus 7
  - 2.2.1 Univariate 7
  - 2.2.2 Multivariate 8
- 2.3 GQN Network 9
  - 2.3.1 Definition 9
  - 2.3.2 Usage 10

3 Workflow **12**
- 3.1 Functional 12
- 3.2 Top down 13
- 3.3 Walking sim 13
- 3.4 Object morphing 16

4 Maze Game Systems **24**
- 4.1 Audio 24
- 4.2 Sound spectrum analyzing materials 24
- 4.3 Checkpoints 25
- 4.4 Enemies 25
  - 4.4.1 Speeder 28

4.4.2 Invisible champer 28
4.4.3 DumDum 28
4.5 Player interaction 33
4.5.1 Laser 33
4.5.2 Smoke balls 33

5 Neural Network Systems 37
5.1 Data generation 37
5.2 Model 38
5.2.1 Saving and loading of network 38
5.2.2 Data Preprocessing 39
5.3 IPC 39
5.4 Rendering 40
5.4.1 Smoke ball rendering 43

6 Further work 44
6.1 Interactive data generation 44
6.2 Network state blending 45
6.3 Model improvements 45
6.3.1 Rendering nonstatic objects 45
6.3.2 Interactive environment modeling 46

7 Reflection 47
7.1 Project 47
7.2 Workflow Evaluation 48
7.3 Other directions 49
7.3.1 GQN and reinforcement learning 49
7.3.2 Synthesized game 50

8 Resources 52
8.1 Software 52
8.2 Unity packages 52
8.3 Python packages 53
8.4 Textures 54

1 Introduction

Shrouded mirror is an experimental experience, where the player sees the world through the eyes of a neural network. The player has to move through an environment, collect checkpoints and avoids enemies. Each type of entity emits a unique audio signal, creating a distinctive soundscape.

1.1 Motivation

Using the representation a neural network has learned of an environment to present the environment state to the player, in an interesting way, is a nearly unexplored avenue. In this work I explore one way neural networks can be utilized in interactive experiences. I use a neural network that has learned how to render an image of the environment out of a certain position and angle. The rendered images are then used as the main information stream presented to the player.

1.2 Scope

The following items where developed during the bachelor project:

- Data generation environment
- Data processing pipeline
- Scene rendering neural network model and training pipeline
- System to send network output to Unity
- Merging of network output with unity rendered objects
- Several exploratory prototypes
- One extended Prototype (Maze game)

2 Background

2.1 Neural networks

To give a better intuition for how neural networks work and how they are used in this project, a brief overview follows describing the components necessary to build a simple neural network.

In brief a neural network is computational structure made up of multiple units called neurons that can adapt their output based on data.

2.1.1 Neuron

A simple neuron can be defined as:

$$\begin{aligned} N_{\boldsymbol{w},b}(\boldsymbol{x}) &= \sigma\left(\sum_{i=1}^{n}\Big(w_i x_i\Big) + b\right) \\ &= \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) \end{aligned} \tag{2.1}$$

Where $\boldsymbol{x} \in \mathbb{R}^n$ is a vector containing all inputs, $\boldsymbol{w} \in \mathbb{R}^n$ a vector containing the neurons weights, $b \in \mathbb{R}$ is the bias, $n \in \mathbb{Z}$ is the number of inputs and σ is a nonlinear function refereed to as the activation function of the neuron. σ is often set to be the rectified linear unit $ReLU(x) = \max(0, x)$.

Intuitively a neuron performs a weighted sum over its inputs adds a bias and applies an activation function to the result to

calculate the final output. The bias can be thought of as a threshold, determining—if $\sigma = ReLU$—how high the sum of inputs has to be, before the output becomes non 0. Figure 2.1 shows, how we can think of a neuron as having connections, where the entirety of the connections form the input vector.

2.1.2 Multilayer network

As seen in fig. 2.2 a neural network work can be formed out of neurons by first organizing multiple neurons into a layers[1] and then connecting multiple layers together. To get a dense network each neuron in a layer gets as inputs the outputs of all the neurons in the previous[2] layer. The first layer is used to provide the input to the network. We can simply set the output values of the neurons to what we like to have as inputs to the network. To get the output of the network we read of what values the neurons in the final layer have after the network has been evaluated. The layers between the first and last layer are called hidden layers. This is because it is normally not necessary to inspect directly what values or parameters these neurons have.

Normally in an implementations of a densely connected neural network the computer performs multiple matrix operations to compute the output. This is usually more efficient because many matrix operations, in implementations we would use, are heavily optimized. Equation (2.2) shows how to compute the values for the first hidden layer of the network in fig. 2.2 this way. The σ is applied to each element of the vector.

[1] In fig. 2.2 a layer is a collection of neurons that line up vertically.
[2] The layer to the left

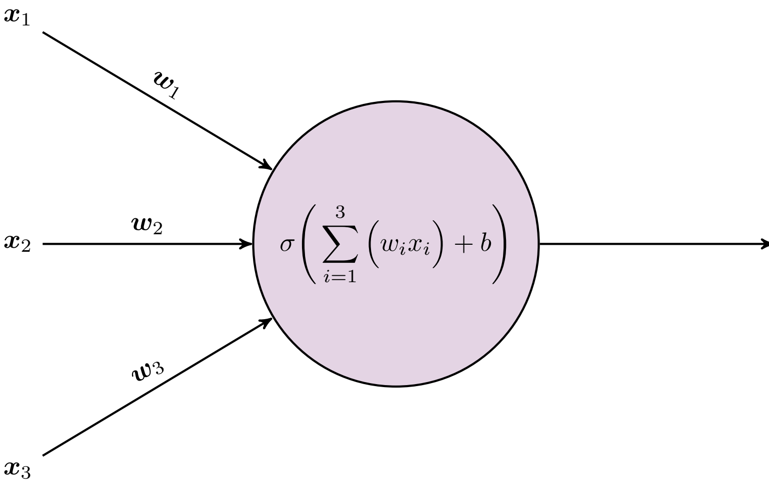


Figure 2.1: Graphical representation of a neuron with 3 inputs

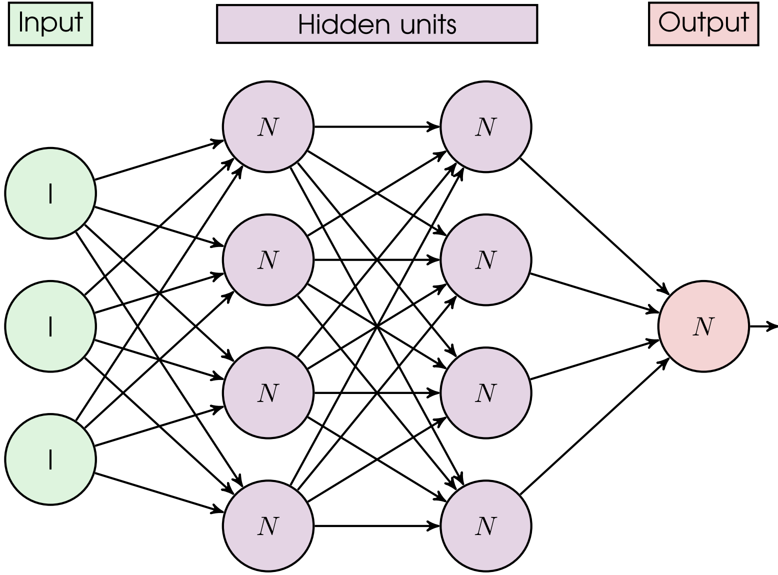


Figure 2.2: Multilayer neural network structure

$$Hidden_1(\boldsymbol{x}) = \sigma\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \\ w_{30} & w_{31} & w_{32} \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) \tag{2.2}$$

2.1.3 Training

Before we start training we need to set our weights and biases to some start values. The bias is usually initialized to 0 and the weights can be initialized to small random values if the network is small enough. More complicated initialization strategies are needed for the weights if we are dealing with a big network (Glorot and Bengio 2010; Ioffe and Szegedy 2015).

Normally we are interested in finding specific weight vectors and biases for neurons that make the network perform some specific task like image classification.

We might want to know if an image pictures an orange or an apple. For this task we can use a network with one output neuron, where that neurons value should be one if we feed in a picture of an orange and zero if we feed in a picture of an apple. If we now feed in a picture, the network gives back some arbitrary numerical value that will only by chance predict the correct fruit. This is because the network is not trained yet.

To train the network we first need to define a loss function. This function returns a numerical value that represents how bad the network is. For images we may use the means square error function as shown in eq. (2.3). There $\boldsymbol{x}$ is the output of the network and $\boldsymbol{y}$ is our desired output.

$$MSE(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x}_i - \boldsymbol{y}_i)^2 \tag{2.3}$$

To train the network we need a dataset of input and desired output pairs. To create a dataset we first need a bunch of images of apples and oranges. Then we pair each image with the number one if the image depicts an orange and the number zero if the image depicts an apple.

Now if we feed in an image into the network we can measure how bad the network is using the loss function. To improve the output of the network, we train it by adjusting our weights and biases proportionally to the negative gradient of the loss function with respect to the weights and biases using calculus.

2.2 Differential calculus

Differential Calculus is an area of mathematics that studies how a functions output changes with regard to tiny nudges to its inputs.

2.2.1 Univariate

The derivative can intuitively be interpreted as giving the value of how fast the output of a function changes as well as the direction of change[3] at a specific point if we start to increase x. It is defined as:

$$\frac{df(x)}{dx} = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx} \tag{2.4}$$

The $\lim_{dx \to 0}$ means, that we don't use a specific value for dx but take the value the equation approaches when dx approaches 0 without dx ever becoming 0.

This means that the derivative can be used to find out how a function changes locally. If the derivative is positive and we increase the input to the function, at the point the derivative was

[3] The output either increases or decreases if we increase x.

evaluated, the output increases in size. This increase is proportional to the magnitude of the derivative. The same logic applies when the derivative is negative.

2.2.2 Multivariate

The same can be applied to functions with multiple parameters:

$$\begin{aligned} \frac{\partial f(x_1, x_2)}{\partial x_1} &= \lim_{\partial x_1 \to 0} \frac{f(x_1 + \partial x_1, x_2) - f(x_1, x_2)}{\partial x_1} \\ \frac{\partial f(x_1, x_2)}{\partial x_2} &= \lim_{\partial x_2 \to 0} \frac{f(x_1, x_2 + \partial x_2) - f(x_1, x_2)}{\partial x_2} \end{aligned} \tag{2.5}$$

Here each equation defines as how much the output of the function changes locally when the corresponding input parameter to the function increases. The gradient of a function is a vector containing the derivative information with respect to each parameter. The gradient of $f(x_1, x_2)$ is defined as:

$$\nabla f(x_1, x_2) = \begin{bmatrix} \frac{\partial f(x_1, x_2)}{\partial x_1} \\ \frac{\partial f(x_1, x_2)}{\partial x_2} \end{bmatrix} \tag{2.6}$$

Calculating the gradient of the loss gives us the information of how we should adjust our weights and biases to improve the loss i.e. improve the output of the network. Normally we scale the gradient by some learning rate $\eta \in \mathbb{R}$.

2.3 GQN Network

2.3.1 Definition

A Generative Query Network (Eslami et al. 2018) is a type of neural network architecture that can learn to render an image from a queried viewpoint in an environment. The output of the network is then, in the ideal case, identical to an image a camera would produce when placed at the same queried viewpoint in the same environment.

The network can perform this rendering even for environments that where not seen during training. This works because the network constructs a representation of the current environment for each render (fig. 2.3). To generate this representation we need to input a few[4] image-coordinate pairs into the network. These image-coordinate pairs need to be of the environment that we want the network to render. The coordinate that is paired with an image denotes where[5] that image was taken.

During training the network has learned how to encode these image-coordinate pairs into an abstract representation of the environment. Also, the network learned how to use this representation to render an image from a queried viewpoint. The closer the unseen environment is to environments seen during training, in terms of objects in the environment and their properties, the better the network is expected to perform. One thing the network is very good at is to learn where objects, that appeared in the training set, are placed in an environment, even if an object was never observed to be positioned in this particular way in any environment. To some extend the network is able to correctly render objects not present in any training environment (Eslami et al. 2018).

[4]How many images we use is a hyperparameter.
[5]The coordinate includes the position and rotation.

The network architecture is shown in fig. 2.3, where I_1 denotes an image of an environment and C_1 the position and rotation where the image was taken. The $\bigcup$ here means concatenation. All R_i are, as tensors encoded, representations of the environment. These tensors are summed up element wise over all tensors so that R has the same dimensionality as R_i. C_q is the coordinate from which the network should render an Image, z is a vector of latent variables and I_y is the ground truth image of what a render form C_q looks like. The encoder and decoder are both neural networks.

To train the network we feed image-coordinate pairs and compute the gradient of the loss. For the target output and the query position we use an image-coordinate pair from the same environment. With the gradient of the loss we can then update the network parameters. All (I_i, C_i) and (I_y, C_q) have to be drawn from the same environment in one evaluation of the network. If there are multiple (I_i, C_i) input pairs, the same encoder network is used to encode each of them. The update procedure has to be repeated many times until the network output reaches the desired quality.

2.3.2 Usage

This work uses a simplified implementation of the Generative Query Network.

In this work the architecture of the used neural network differs from the implementation of the GQN used in the original paper. Here I use simple dense models for the encoder and decoder that don't use random latent variables. This circumvents the necessity of having to use the evidence lower bound as an optimization target.

This however limits the networks abilities. It does not take into

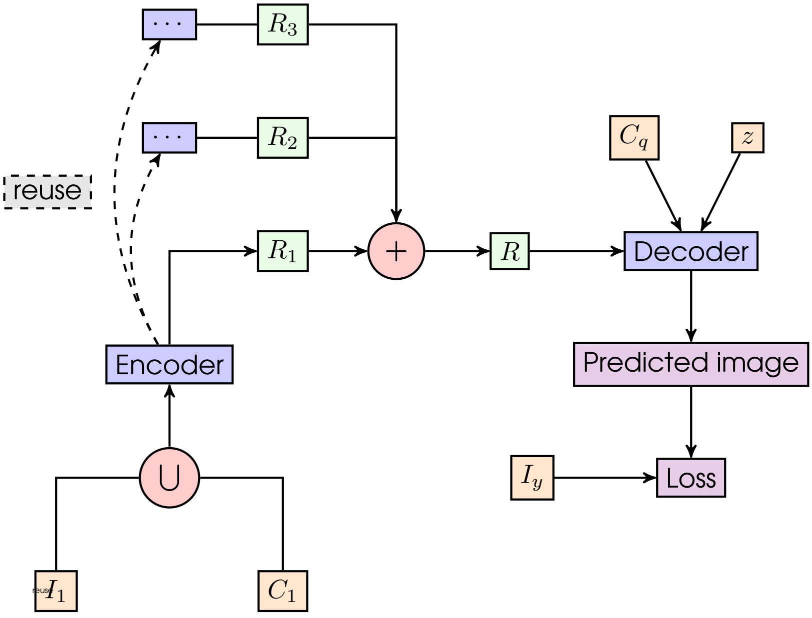


Figure 2.3: GQN architecture

account the inputs, given during inference[6], to the same extent as in the original experiments described in Eslami et al. 2018. The only meaningful considerations of the inputs of the network, that where given during inference, were found to be the coloring of the sky, floor and walls (fig. 3.1). These only worked on static objects. In the original implementation the network can correctly infer more properties like object position, rotation, color and texture on nonstatic[7] objects.

Because interesting use cases where found that do not depend on these properties of a GQN network, no more effort was put into recreating the abilities of the original implementation.

[6]Producing output without training a network.

[7]Nonstatic referees to objects that change their position between environments. An any given environment objects are still static.

3 Workflow

In this project I fallowed an iterative prototype workflow, creating multiple prototypes to evaluate specific ideas. As for implementing different systems needed for the application, I followed the workflow of researching a single component[1] and then implementing it, before beginning further researching efforts.

What fallows is a description of the prototypes developed over the course of the project. The main prototype, called "maze game", became the focus of the project after the direction had been set with the help of the results obtained from previous prototypes. The maze game is described in detail in chapter 4.

3.1 Functional

The goal of the first prototype was to developed the core systems. The entire data generation- and preprocessing pipeline and the neural network model where implemented and evaluated. A small level with checkerboard textures was created for this prototype (fig. 3.1 and fig. 3.2). The sky and wall colors are variable between environments. An environment is a variation of a level. In this case, everything but the sky and wall colors stays constant for each environment, e.g. the walls are always at the same position in each environment. As fig. 3.1 shows, the network can adjust its output based on the information that is being fed in. The network is able to this without retraining. All that needs to be changed is the inputs to the encoder (fig. 2.3). More information on the systems developed in this prototype can be found in

[1]Such as the Kears functional API, UDP sockets, etc.

chapter 5.

3.2 Top down

This prototype was used to evaluate the suitability of the network for a top town game. In this prototype the player has to go from one colored platform to another while avoiding the red walls. The challenge should be created by only training the network on data that is captured by pointing the camera at one of the colored platforms. Training the network on this data resulted only in making the network output blurry, when the player was not positioned on a platform (fig. 3.4 and fig. 3.3). Because the prototype did not present an interesting direction, it was abandoned.

3.3 Walking sim

Here the idea was to present the level to the player through the neural network to create an interesting visual appearance. I also experimented with having certain objects only be visible if the player is at a certain position in the environment (fig. 3.5). This idea was further explored in the next prototype.

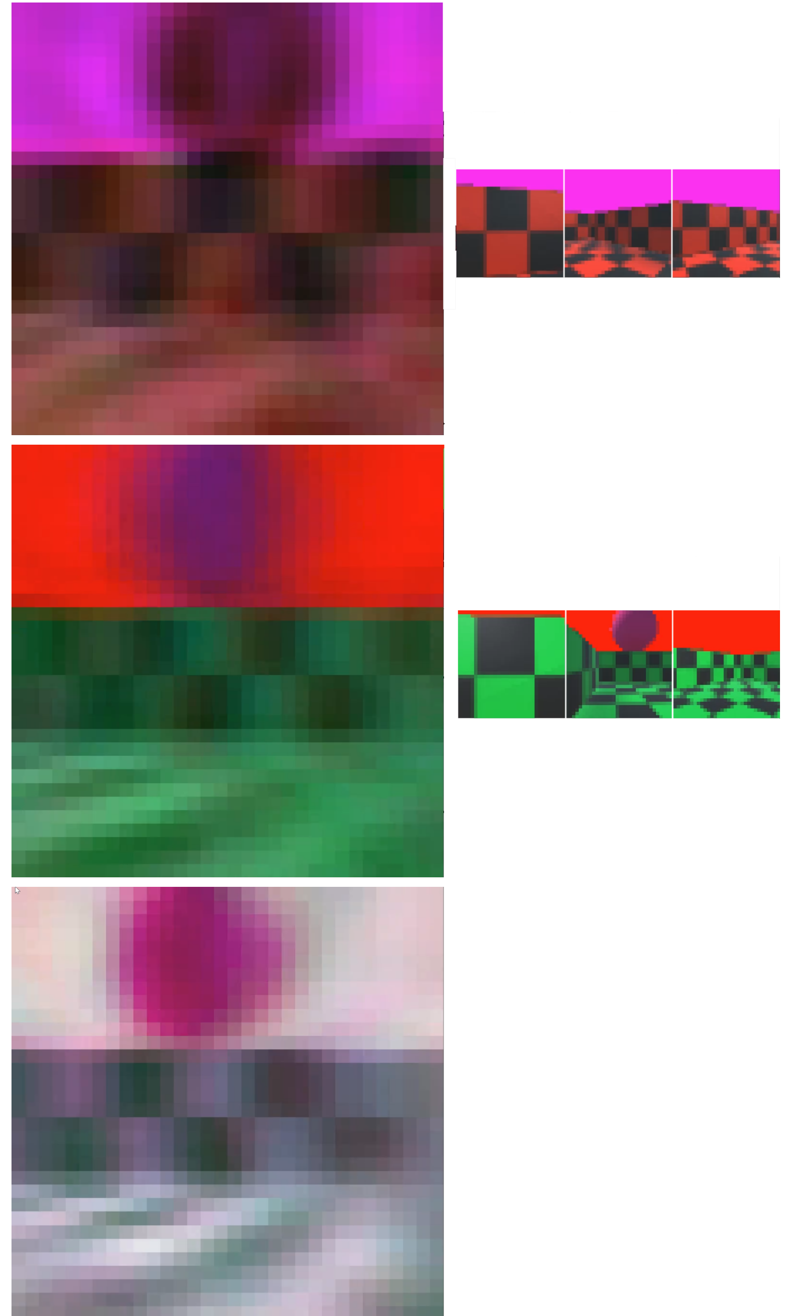


Figure 3.1: The neural network predicts different colors based on the input data that makes up R. The input pictures are shown to the right. The network gives output even if no images are fed in as seen in the last image.

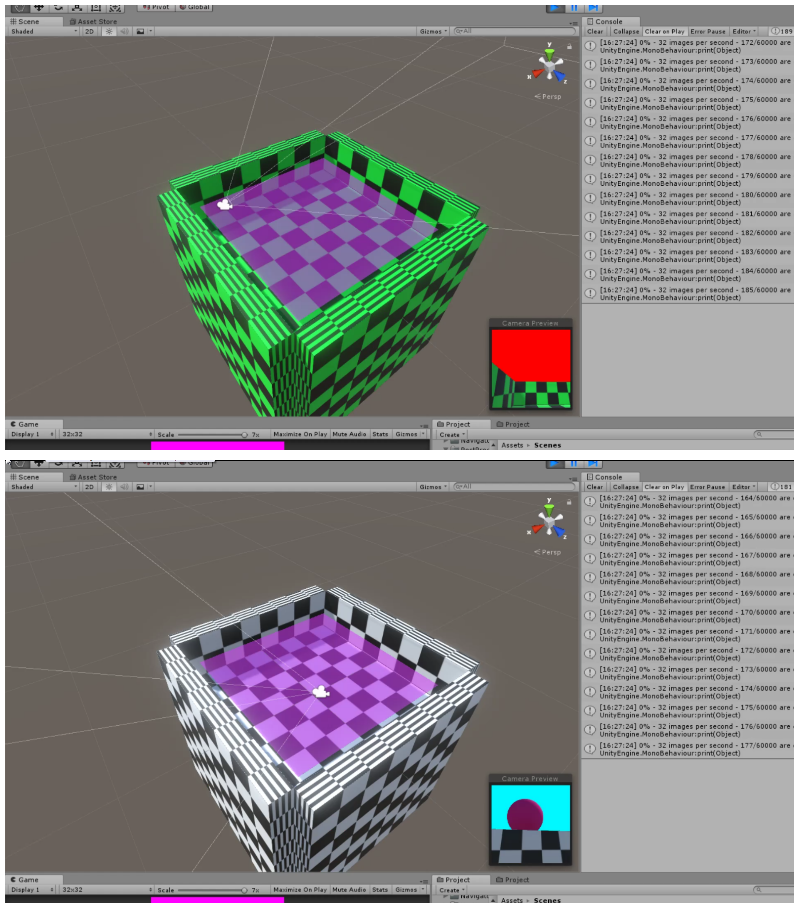


Figure 3.2: Training data is being captured in the functional prototype level.

3.4 Object morphing

This prototype has a level that is divided into four differently colored platforms (fig. 3.7) that are surrounded by tall pillars. The coloring of the platforms and the pillars help the player to orient himself. In the center four different objects are placed (fig. 3.8). Each of the objects is linked with a different marker group as described in section 5.1. The marker groups are shown in fig. 3.6. There is a marker placed on top of each platform. This means that if an observation is taken from a specific platform only one of the four objects is displayed in the center. When the player crosses from one platform to another the center object is morphed from one object to another as seen in fig. 3.9. This is probably because a neural network with a limited amount of parameters cannot model an instantaneous change in "pixel space".

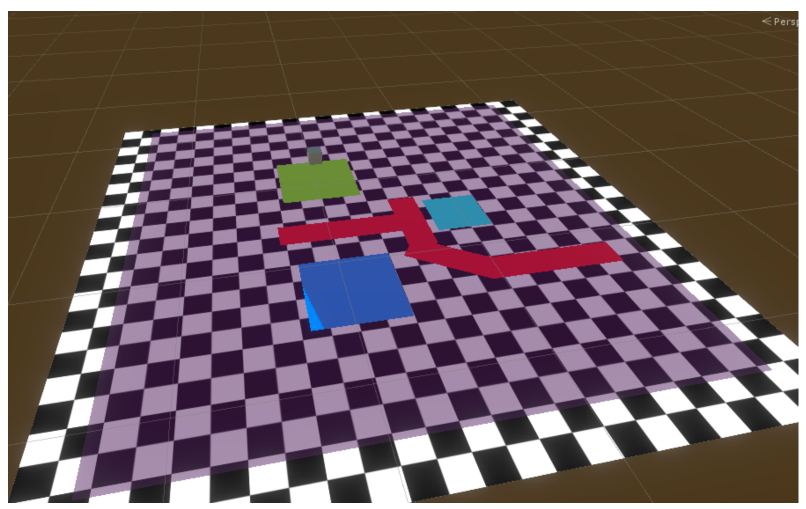


Figure 3.3: Top down prototype level as seen in the editor

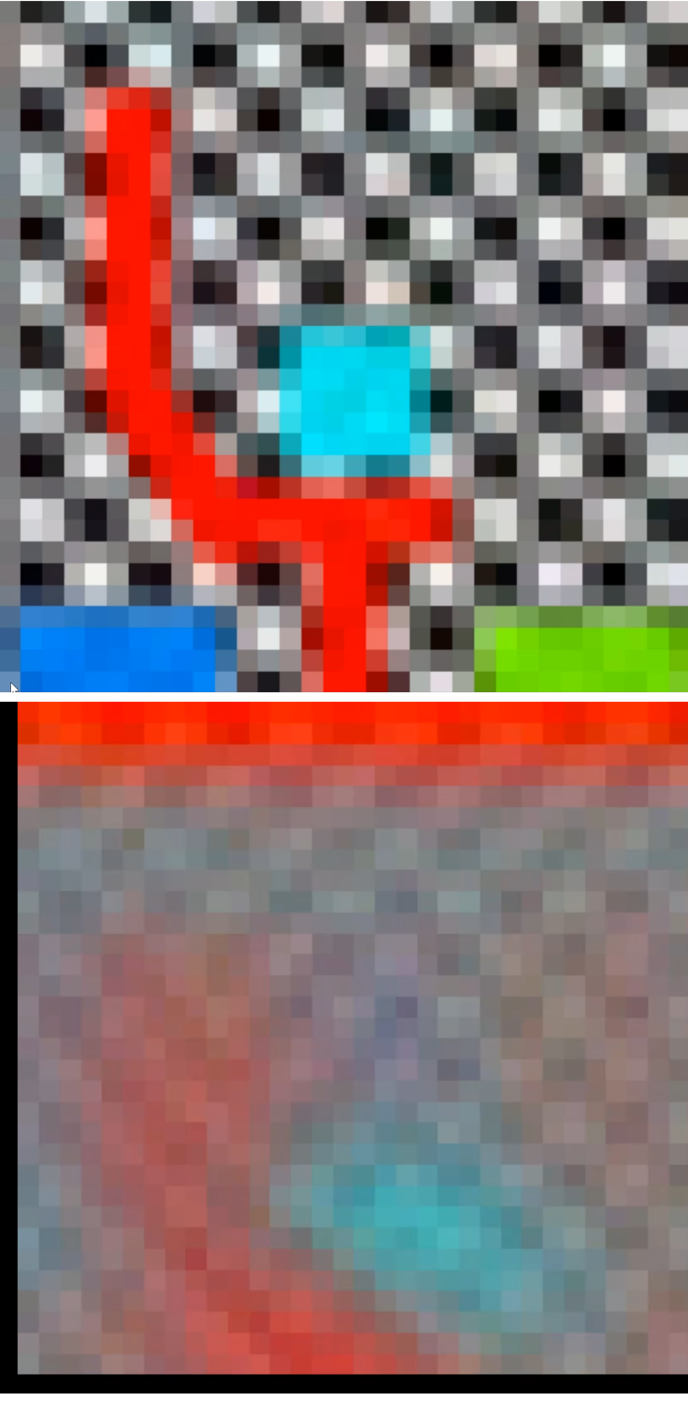


Figure 3.4: In the top down prototype, the network output gets blurry when the player walks off a platform.

Figure 3.5: Certain objects can only be seen, when the player stands at certain locations.

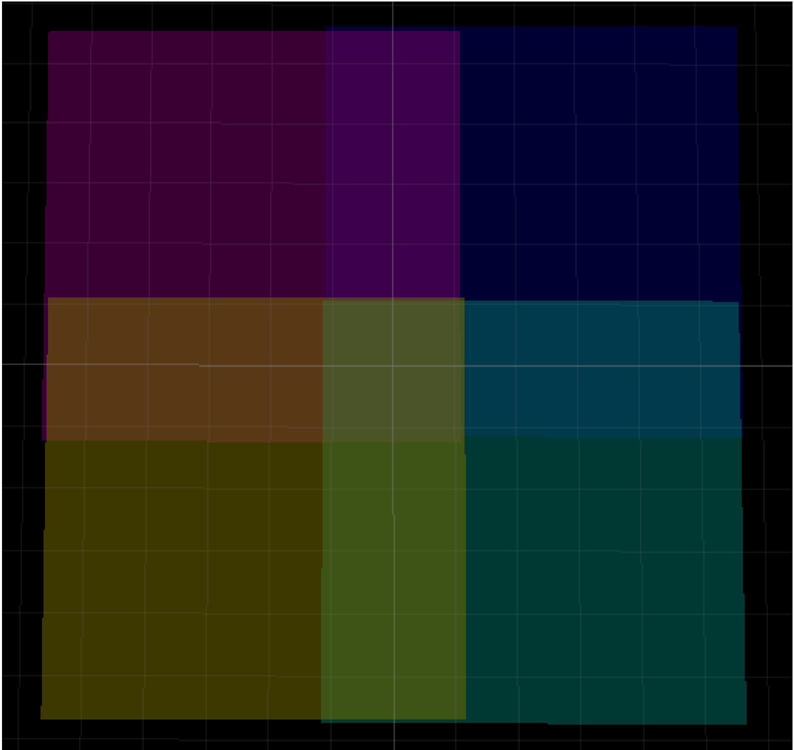


Figure 3.6: These are the 4 markers (explained in section 5.1) of the morphing environment. Overlapping areas are differently colored.

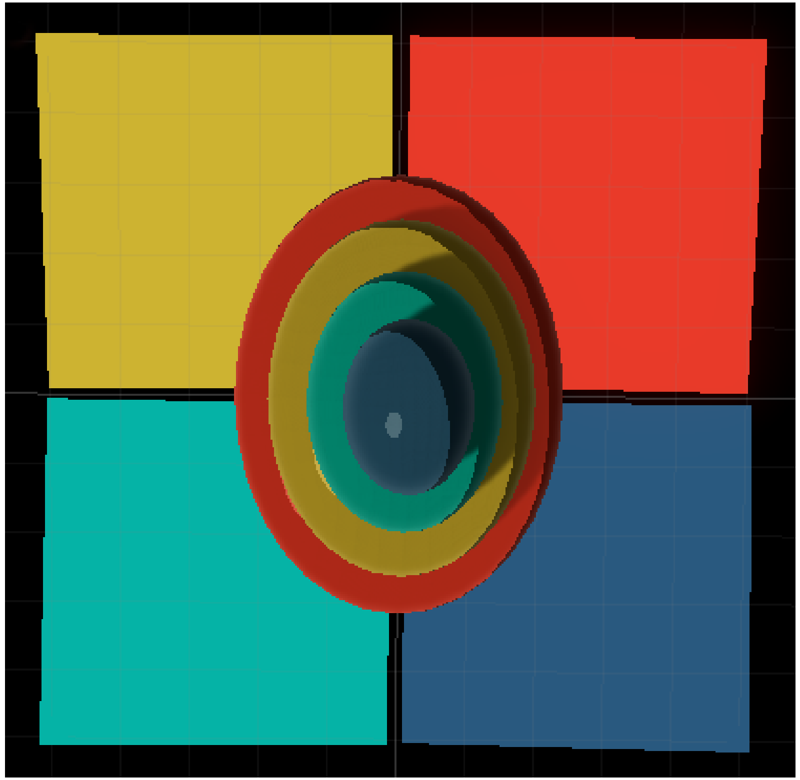


Figure 3.7: Top down view of the level with only one object activated in the middle

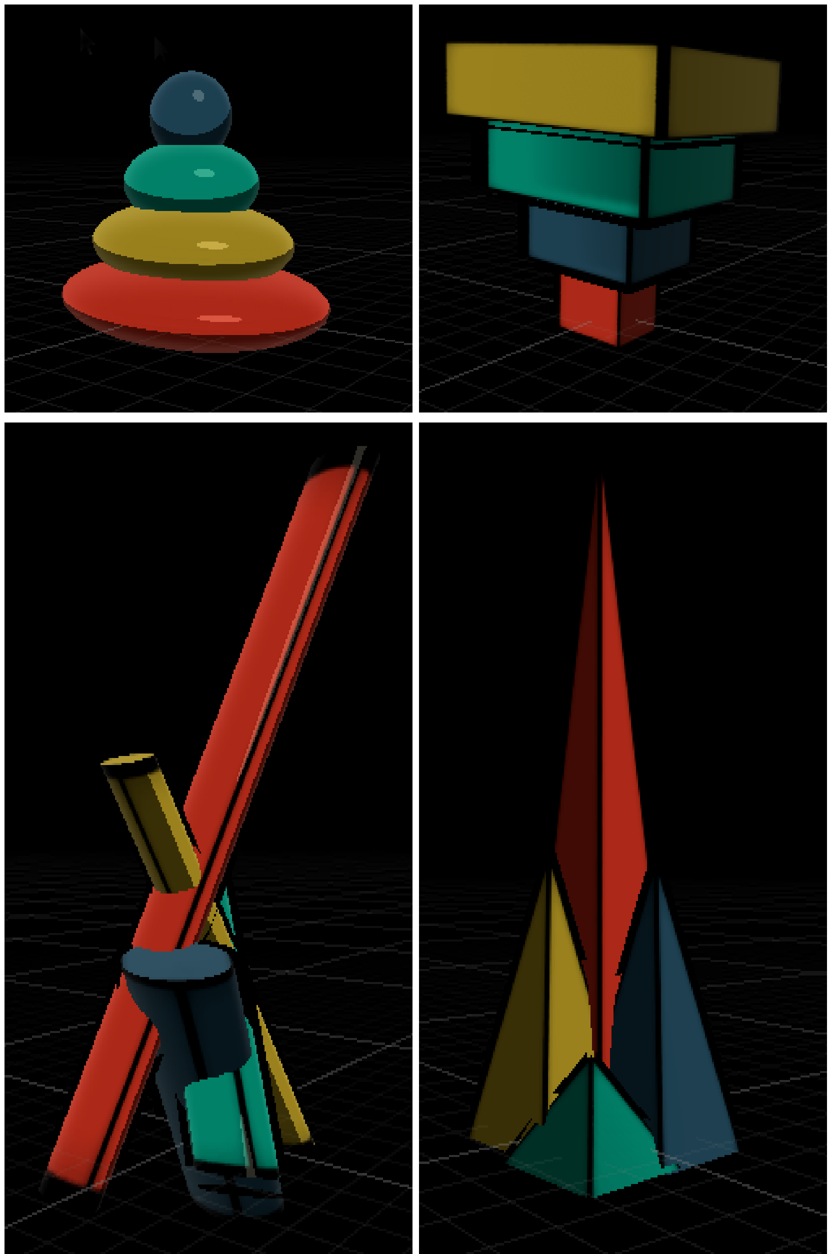


Figure 3.8: The 4 differnt objects in the center of the level

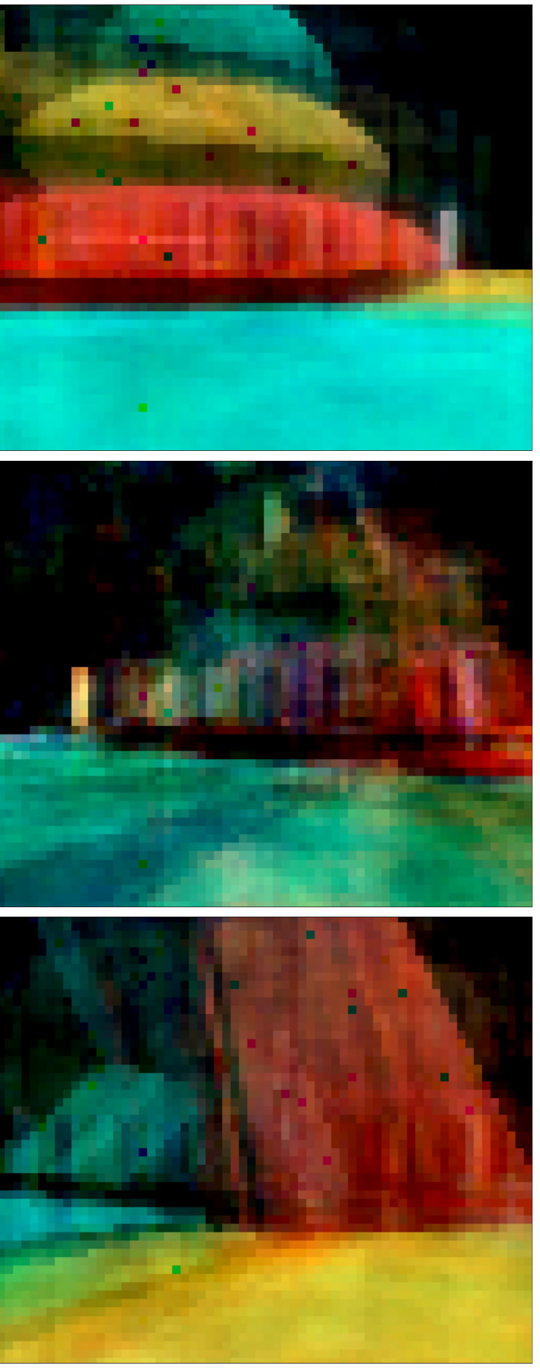


Figure 3.9: The center object morphs into a different object when the player crosses from one marker into another.

4 Maze Game Systems

In this Prototype the player has to navigate though a maze. All static objects in the environment are rendered with a neural network. The player has to use the output of the neural network to navigate through it. There are several enemy types that have to be avoided or destroyed. Multiple checkpoints have to be collected before the goal is unlocked. By collecting the goal the player wins the level.

4.1 Audio

In this project I use the Resonance Audio plug-in to simulate spatial sound. This facilitates player orientation. All important objects and events such as enemies and checkpoints emit a unique sound, that helps to spatially locate and identify them. A Doppler effect is used on all audio sources, so that the player perceives how entities are moving relative to him.

4.2 Sound spectrum analyzing materials

I created a simple sound spectrum analyzing script, that when attached to an object automatically varies the intensity of the objects material color based on the current spectrum of the objects audio source. Parameters are exposed to define ranges for valid color intensities and to set how the spectrum should be filtered before its intensity is determined.

4.3 Checkpoints

To beat a level the player has to collect all checkpoints. They are visualized as green rectangular cuboids that vary their material using the spectrum analyzing script (fig. 4.1). Periodically a checkpoint emits a high pitched, slowly decaying ping sound. Because the sound is specialized with Resonance Audio, the player can use this audio signal to determine the direction and distance to a checkpoint. Only one checkpoint is active at a time, and a new checkpoint only activates if the current one has been collected. This makes it easier for the player to home in on the sound. The final checkpoint, refereed to as goal, has the same audio signal but different visuals (fig. 4.2) and triggers the game won state when reached.

4.4 Enemies

Enemies in this prototype all follow a simple pattern. They spawn out of a for the player visible spawn point and move in a straight line across the map. How they move is different for each enemy type. Each of them emits a sound that helps to identifies them. They are all pyramid shaped with the apex pointing in the flight direction, to give them the association that they are dangerous.

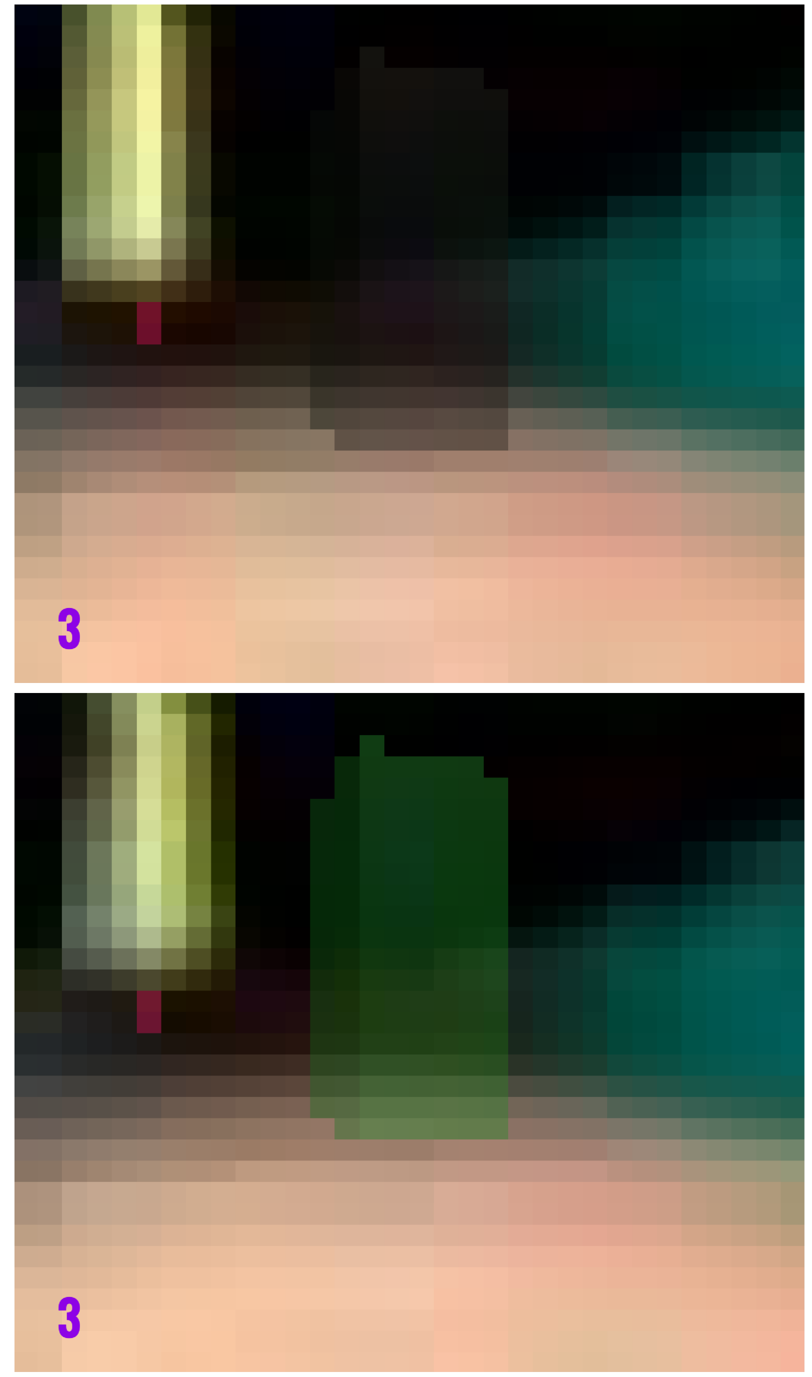


Figure 4.1: Checkpoint cuboid varying material color based on the current sound

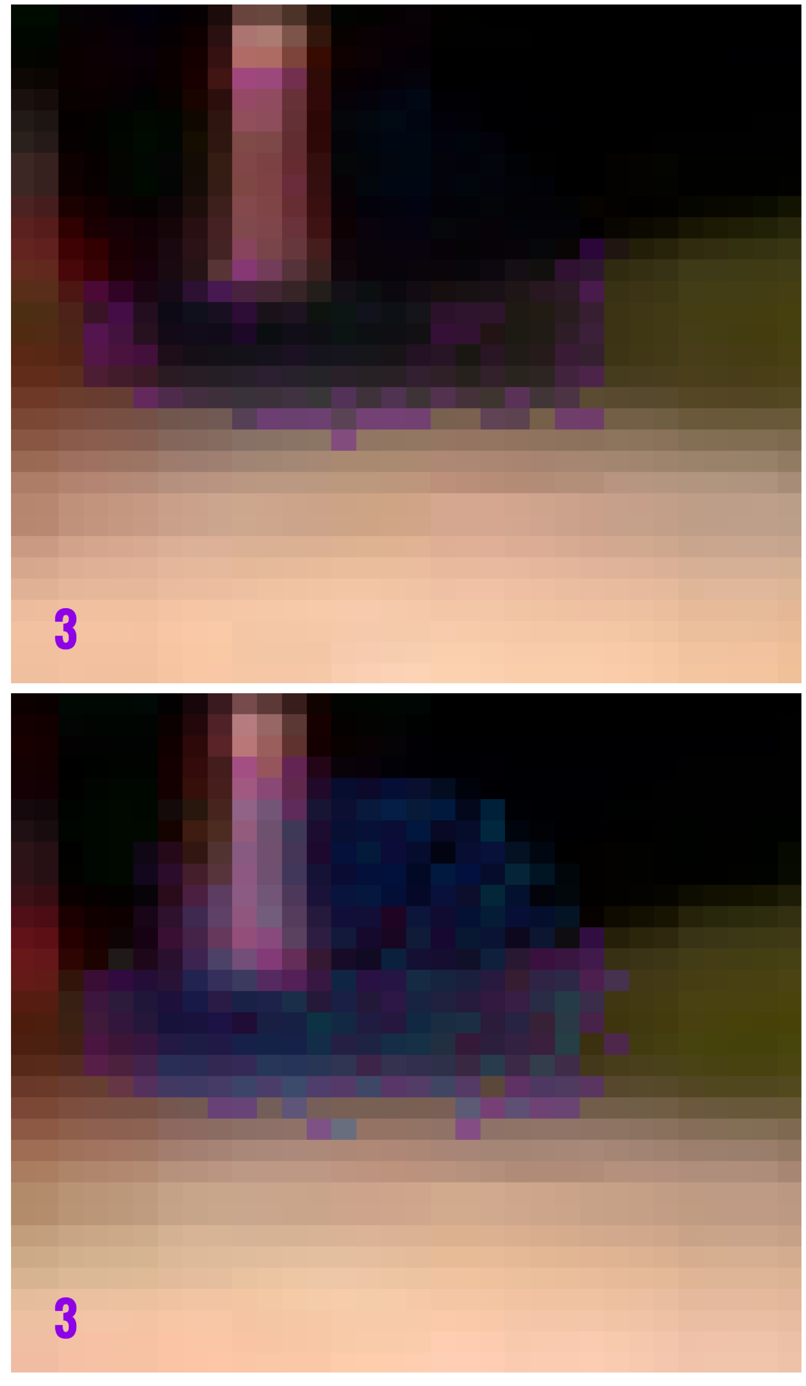


Figure 4.2: Goal varies its material color based on the current sound

4.4.1 Speeder

Figure 4.3 shows the simplest enemy, the speeder. He moves forward with a constant velocity. This enemy emits a two sounds. One can only be heard by the player if the enemy is moving directly at him. A variation of the speeder exists that moves way faster and looks different (fig. 4.4). Because of the Doppler effect applied to the audio sources, the sound emitted feels more dangerous. This is probably because with the Doppler sound you also hear that it goes really fast.

4.4.2 Invisible champer

This enemy moves just like the speeder. However the champer is a much harder enemy because he is invisible most of the time. Periodically he makes a champing sound. Using the spectrum analyzing script the material of the champer is adjusted to make him visible for a brief period, every time he emits a sound (fig. 4.5).

4.4.3 DumDum

The DumDum uses a script that analyzes the sound spectrum of the attached audio source to determine his movement speed. The audio source loop of the DumDum consists of four short pulses of sound and a long pause. Each time one of the pulses is played the DumDum starts to move until the pulse stops. This leads to the DumDum rapidly jumping forward four times before becoming stationary for brief period of time.

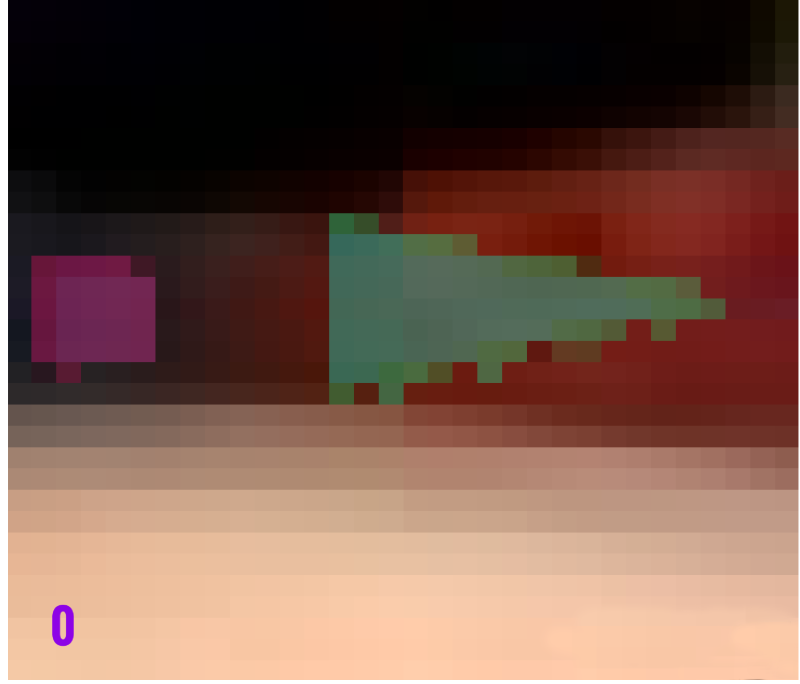


Figure 4.3: Speeder moves away from his pink spawn point

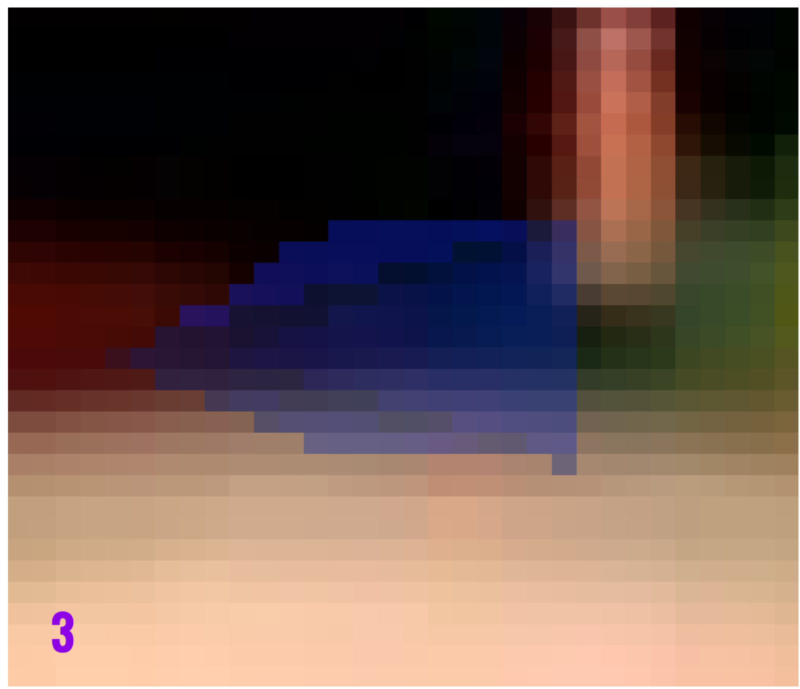


Figure 4.4: A variation of the speeder that moves faster

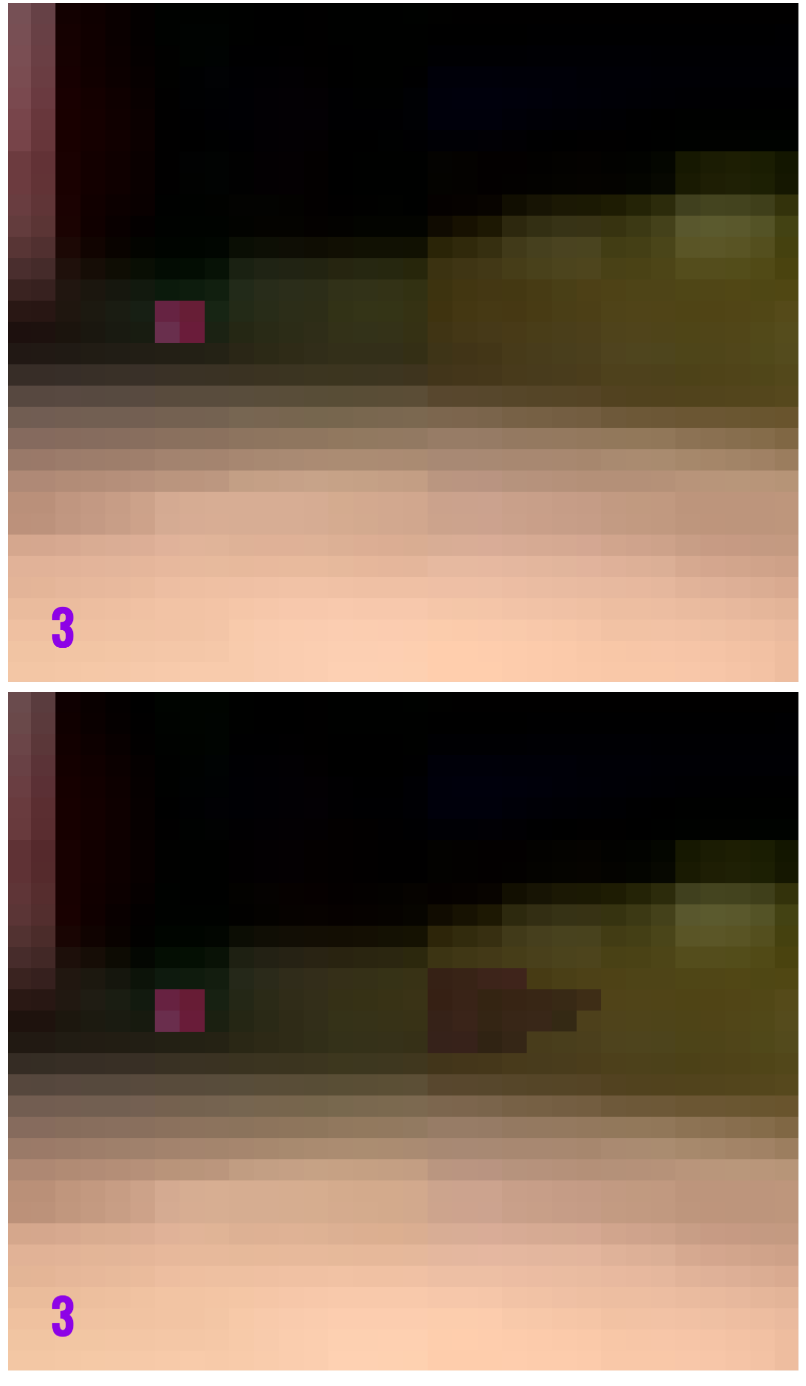


Figure 4.5: A champer becomes visible because he emits a sound

Figure 4.6: Two DumDums standing still because the silent part in their loop is played

4.5 Player interaction

4.5.1 Laser

The player has a laser that he can shoot to destroy enemies. Ammunition is limited but can be refilled by reaching a checkpoint. If the player attempts to shoot without ammunition a failure sound is played, and a puny particle effect is displayed. If the player hits an enemy, a death sound and -particle-system are played as seen in fig. 4.7.

4.5.2 Smoke balls

Smoke emitting balls can be thrown by the player. As shown in fig. 4.8, smoke balls can reveal where structural objects, like walls, are in the Unity environment. This is useful because where a wall is in the unity environment and where it is displayed by the network might vary considerably in places. Smoke balls can also be used to uncover the invisible champer as seen in fig. 4.9. More information on how the smoke ball works can be found in section 5.4.1.

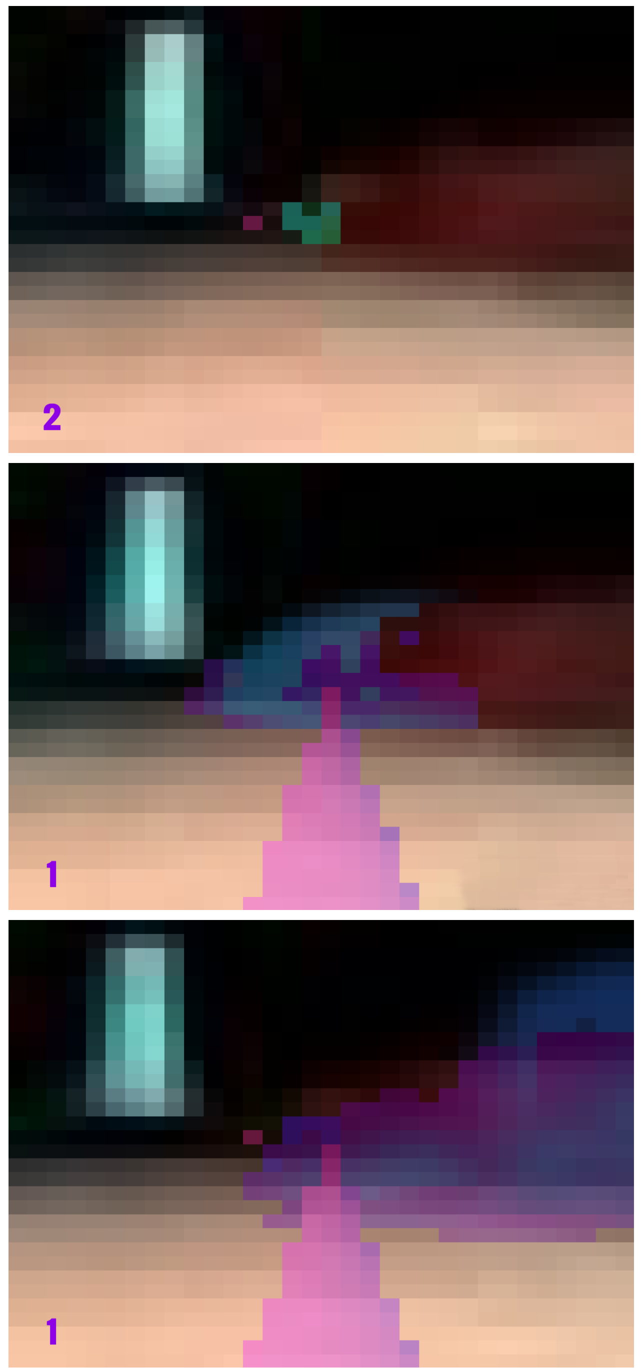


Figure 4.7: The player destroys an enemy with the laser.

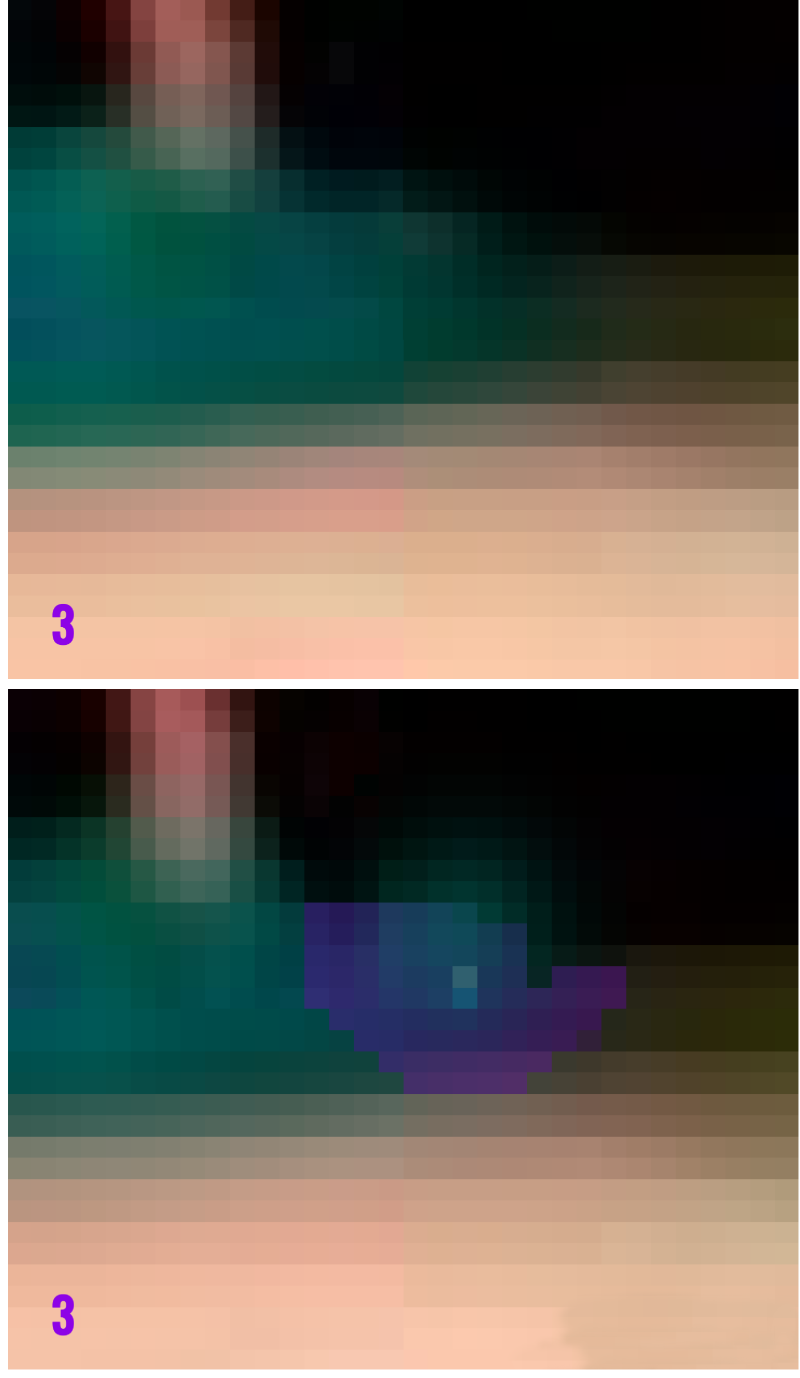


Figure 4.8: The player reveals where the walls are in the unity environment using smoke.

Figure 4.9: The player reveals a champer using smoke.

5 Neural Network Systems

5.1 Data generation

The first step is to place markers in the environment where observations should be taken. These markers are scaled Unity cubes where the volume denotes all the valid observation positions, while allowed camera rotations are set in the capturing script. It is possible to group markers and link groups with objects in the environment. Linked objects are then only visible to the capturing camera if the camera is placed in a marker that is in the group the object is linked to.

During data generation, a script moves the camera to a random position and rotation. The camera is then programmatically rendered to a texture and then saved to a file. Camera positioning and capturing can be done multiple times per frame to speed up the generation process. When the file is saved, the position and rotation where the image was taken is saved in the filename of the image. Depending on in which unity scene the image was captured and at what resolution it was set it is sorted automatically into a corresponding folder structure that is created on the fly.

The data generation process can easily be customized in the inspector of the capturing script. In a list, entries can be added that define a capturing resolution and the number of data points to be captured at that resolution. Once the application is run in capturing mode, all data points specified in this list are sequentially generated. On completion a notification sound is played and the environment is automatically shut down.

In the Maze game all structural objects are cubes, where a raised cube represents a wall. This makes it easy to find all valid player positions and generate the markers for the environment automatically.

5.2 Model

To create the neural network model used in this project the Kears functional API is used. Keras is the official front end to Tensorflow and simplifies the creation of neural networks. With the functional API predefined callable class instances, which represent layers, can be used to define a models architecture. For this a class of a layer type is set up and called with the input to the layer. This then returns the output[1] of the layer. The outputs can now be feed into another layer. After the model has been described in this manner, Keras can receive optimization parameters e.g. the loss function and then compile the computation graph.

The model created corresponds to the model described in section 2.3, with the exception that no latent variables z are used. As encoder and decoder dense networks are used.

5.2.1 Saving and loading of network

To make it easier to identify and reuse models a simple versioning system has been developed. If no specific model is specified to be used, a new one of the specified architecture is created. The model is then given a unique file name consisting of the following: date; time; observation environment; name-ID; version; numeric-ID.

If a request is received to load a specific model, an automatic

[1] The Keras classes don't execute statements immediately, but construct a computation graph for later execution. The output by the layer can be thought of as a pointer that is resolved later, when the model is executed.

search is performed for the newest model of the requested id. If a model is found that is newer than the one specified that model is automatically loaded instead of the old one. The versioning of a model is set up so that it increments the version based on the version of the model that is currently being trained.

Keras is used to automatically save the model during training in intervals, preventing total data loss in case of a critical system failure.

5.2.2 Data Preprocessing

First the data is loaded from from disk. Then the data points are normalized. This means that we make all inputs of the data have a range between zero and one. This is required for the used architecture to enable efficient training. The next step is to create a pairing of input data and output labels[2]. This is simply done by choosing all the inputs and output labels from the same environment. These pairings constitute the training data for the model. The output of the model is just a vector, so that we need to rearrange it into a tensor of the original image dimensions. Then we can encode the tensor into a JPEG image.

5.3 Inter process communication

Because Python runs outside the Unity environment we need a way to send the network output to Unity. This is done by using a python UDP socket to send the JPEG images to a listening UDP socket in Unity. Because UDP is a connectionless protocol, and doesn't give any guaranties that the send data will be received, it is a very fast way to send the data. The chance of data getting lost on a local server is very slim. We are also sending a continuous stream of information so that we don't have to care if an image

[2] A label denotes what output we would like to have for a set of inputs.

is lost. A new image will be received within a few milliseconds.

5.4 Rendering

Now the data received by the Unity UDP socket gets loaded into a texture. This texture is then rendered to screen. Because our simplified model is unable to render nonstatic objects we need to merge the network output with rendered output from a Unity camera if we want to use moving objects. For this we change the material of objects, we don't want to see in the merged output, to an unlit material of a specific color, referred from here on as the key-out color. This can automatically be done when Unity enters play mode using a script that filters objects based on their layer. Figure 5.1 shows the application of the material form the editor perspective.

Now we render a Unity camera and bit blit the image into a texture using a custom shader that sets all pixels that are equal to the key-out color to be transparent. Then we apply a pixelation shader to the image to reduce it to the same resolution as the network output. Finally we blit the pixelated version onto the screen over the network rendered output (fig. 5.2).

This whole procedure is necessary to hide objects that are blit onto the network output, when they are behind an object like a wall in the unity environment.

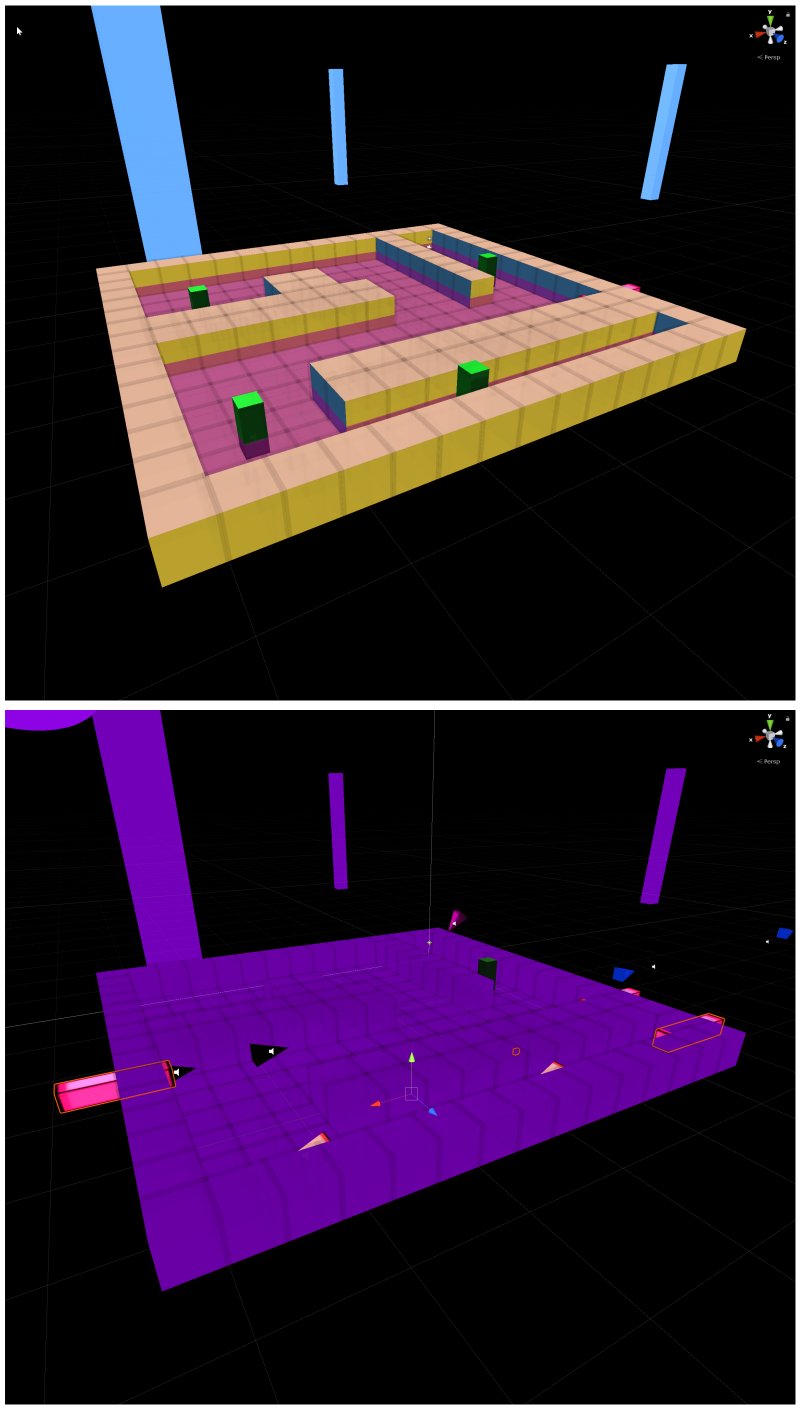


Figure 5.1: The keyout material is applied to everything that should not be visible.

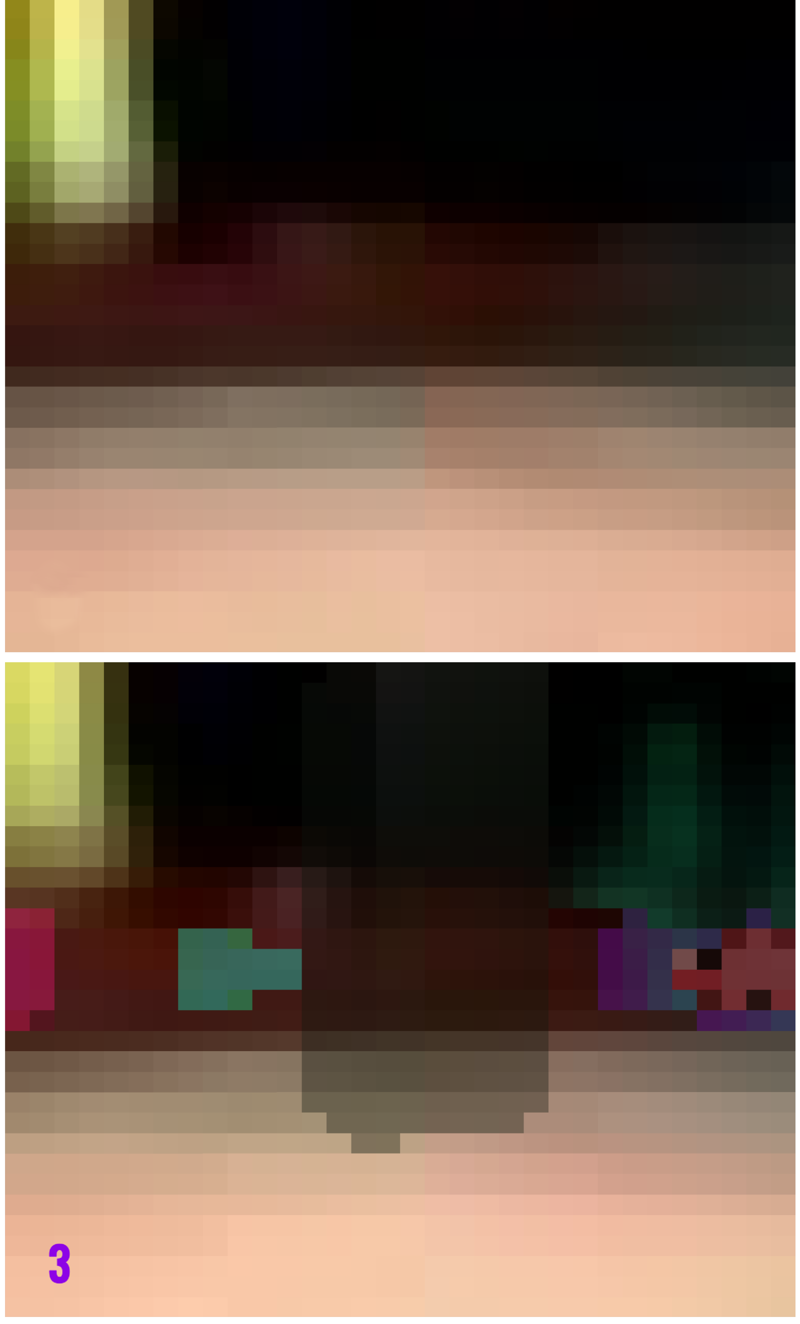


Figure 5.2: The network output is merged with the output of a Unity camera.

5.4.1 Smoke ball rendering

The ability of smoke balls to reveal structural environment objects is created by having the smoke be additive particles. When the additive particles are rendered in front of objects that have the key-out color material, the color gets offset slightly so that it is no longer keyed out by the shader. The objects behind the smoke appear to the player as flat shaded surfaces with the key-out material.

Smoke balls can also be used to reveal the Invisible champer that is described in section 4.4.2. This works because the champer is keyed out by the same shader that keys out the environment. In unity the champer is not transparent but opaque. This means that if the champer is in front of smoke, he overwrites the smoke by getting his pixels in the image set to transparent. The player then sees a cutout in the smoke where only the network output is visible in the shape of the champer.

6 Further work

This chapter presents possible future directions of how the project can be expanded upon.

6.1 Interactive data generation

One additional layer of interactivity could be created by giving the player control over how the network trains. Specifically the player could control the generation of the training data. One way to achieve this is by allowing the player to traverse the unity environment and take screenshots of it via a keypress [1]. The captured images would then be used to train a neural network. After training the player would have to fulfill some task in the environment, but now he has to use the output of the network he trained to navigate the environment.

This could be used to create a competitive multiplayer game where both players first have to train a network and then use the output of their trained network to beat the other player at some task that has to be performed in the environment such as collecting objects, catching the other player or shooting the other player.

[1] or alternatively we could record what the player sees each moment as a series of images

6.2 Network state blending

One way to make the output of the neural network more interesting is by parameterizing the internal learned parameters. During training they would not have be variable, but after training the parameters can be manipulated to morph the output of the neural network. This morphing might be dependent on some dynamic gameplay values i.e. if the player shoots the laser the morphing could be set to height value and then gradually lerp back.

6.3 Model improvements

To enable more prototypes, the capability of the GQN network should be expanded so that it is as capable as the original implementation. The data generation process and preprocessing are capable of supplying such an implementation with the required data. This means that only the architecture of the model needs to be updated. Updating the network in this would would enable the development of the prototypes in the following subsections.

6.3.1 Rendering nonstatic objects

After the update the network would be able to directly render nonstatic objects. To capture the movement of objects, each frame a new observation from all observation points needs to be send to the encoder to update the environment representation, that is then used to render the output. Updating the representation each frame means that the output of the network would always represent where objects are in the environment at the present time.

6.3.2 Interactive environment modeling

In this prototype the player manually updates the inputs to the GQN. For this he places some object in the environment, where the observations should be taken. To create more gameplay depth, the number of possible observations that can be taken per level and the total number of active observations should be limited in some way. This forces the player to strategize about when and where to take observations. Placing observations at a position would make the output of the network more accurate in the region the capturing object points in.

7 Reflection

7.1 Project

The original idea of integrating a GQN into an interactive experience was only partly successful. The time needed to get accustomed with the intricacies of the methods required to use the GQN to its full potential was greater that the time available for research in this project. Only a simplified GQN architecture could be utilized in an interactive prototype.

For the prototypes that where developed, it might be argued that simpler methods exist that could have been used to achieve the same design goal. In the maze game a post processing filter might be applied to make the environment more confounding instead of using a neural network. This is a valid critique. If one wants to create a game similar to the maze game and wants do so quickly one should not use a neural network. However, it is probably the case that the specific style that was achieved in this work, using neural networks, is very hard or practically impossible to recreate using simpler methods.

This work also represents only a first exploration of the possibility space of how neural networks might be used to process visual information before it is presented to the player and the improvements described in chapter 6 would make it possible to create even more unique experiences that utilize the power of the GQN to its full potential.

The final version of the maze game prototype, can stand on its own as a playable, challenging game. Only one level is available in the final quality but with the systems in place, adding more lev-

els would be easy. The goal of the maze game prototype of creating a visually simple but appealing game, where the player has to use sound to navigate effectively through a maze, avoid enemies and collect checkpoints, all while deciphering the output of the network has been achieved.

The project greatly improved my understanding about how neural networks function at the core. The model in the project uses synthetic data. This enabled me to learn how to construct a complete pipeline for an unsupervised machine learning system. I implemented data generated and preprocessing, created a simplified version of the GQN architecture and improved my intuitions about how to train neural networks.

7.2 Workflow Evaluation

When working with neural networks it is often necessary to try out different architectures and parameters for a model that should perform a specific task. For this it is very useful to have programmatic utilities, that automatically search the space of parameters for some time and then returns the best. I realized to late into the project how useful these utilities would have been. The technical debt I collected this far into the project was to high and remaining time of the project to short, so that I decided that the implementation would not be worth the effort anymore.

Another mistake I made, was to make the model too soon too complex. Very early on I started to train the model on data of a camera rotating on multiple axis that produces 64x64—instead of 32x32—resolution images as training data. Both these things significantly increased the time necessary to train the network to a reasonable level. This made iteration time slow. Besides the training time issue, the implementation of parameterized capturing and higher resolution output took too soon an unnecessarily large chunk of time out of my schedule, time that would have

been better spend improving the existing systems.

Furthermore I sunk some time into researching how to train neural network models on multiple machines. After 1–2 days of research I realized, that the endeavor is outside the scope of this project.

Structuring the project around creating prototypes to evaluate specific ideas worked well. This approach encouraged me to finding out early on all the ways an idea might break. I knew that if an idea did not work out I could move on to developing the next prototype reusing what worked and discarding the rest. This allowed me to have a short iteration cycle when exploring directions for the project. A further advantage was that I started to combine ideas from different prototypes that I would not have thought about without having the prototypes as reference. An example is to reuse the object morphing, that was used in a basic form in the walking simulator prototype, and build a whole prototype just around that idea. This prototype then became then the object morphing prototype.

7.3 Other directions

The GQN is a versatile neural network architecture. There are many directions, different from the ones explored in this work, one might want to go. Some that came to my mind are listed below.

7.3.1 GQN and reinforcement learning

One promising way to apply neural networks to interactive environments is through reinforcement learning. Reinforcement learning allows machine learning agents to learn from experience. They might in the future be used to create more realistic NPCs, generate content such as levels and music, or to open up the

world of machine learning to players by letting them train their own agents. The ability of the GQN to create a compact encoding of an environment can be used in conjunction with reinforcement learning.

Eslami et al. 2018 show that it is possible to utilize the representation of a GQN as input to a reinforcement learning algorithm to greatly improve the algorithms data efficiency. This in turn makes an agent learn faster with fewer data points[1] making it more practical to use reinforcement learning in interactive environments. In the paper they train an agent to control a robot arm to reach for a randomly positioned sphere. They also show that the agent can learn how to control the arm, even when the camera is positioned at a random position on a sphere around the robot arm each frame.

The invariance to where the camera is positioned could potentially be used to give control of the camera to a player. The player could then, by moving the camera, determine what kind of observations an agent gets from the environment. This is interesting both as a means of giving the player control of the learning process and manipulating inference. During training the player can focus on different objects the agent can interact with. This would enable the player to determine which skills the agent learns by controlling the generation of the training data. During inference the player can influence the already learned behavior by changing the inputs the agent receives by moving the camera e.g. to observe specific objects.

7.3.2 Synthesized game

One might be able to expand the capabilities of the GQN to include sequence modeling. This would mean, that the model is able to not only predict the current environment state but also how this state evolves over time. The model would then be able

[1] The paper reported that 75% less data was required.

to predict how a mobile object would move on screen given only images of the environment, the coordinates of where the images where taken, and a reference from which time step each image stems. This might even allow the network to predict game states such as game over and game state changes caused by player input similar to what was shown to be possible by Ha and Schmidhuber 2018. This would allow us to train a model on multiple game environments[2] and then smoothly blend between the different environments, creating a nearly infinite amount of possible gameplay variations.

As Ha and Schmidhuber 2018 show, this approach might also be combined with reinforcement learning. An agent could be trained on many of the generated gameplay variations. This might result in a more robust agent, which could be useful if the agent uses a virtual environment to train a task that is supposed to be executed in the real world. A related approach to making agents, that train in virtual environments for real world tasks, more robust is discussed by Peng et al. 2017.

[2]By a game environment is meant that the environment also has a specific rule set attached to it.

8 Resources

8.1 Software

- Unity3D
- Git
- Visual Studio
- Python
- PyCharm
- Blender
- Krita

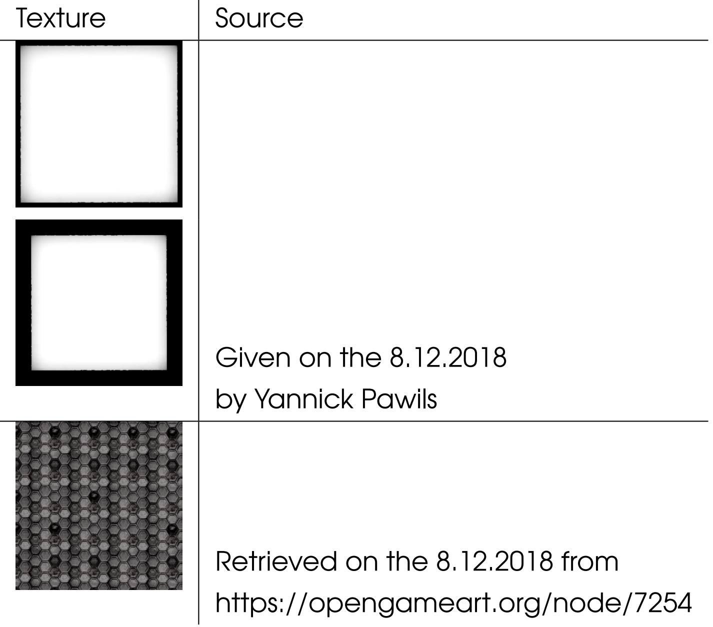
8.2 Unity packages

- MK Toon Free (Toon shader)
- Post Processing Stack
- Universal Sound FX
- Standard Assets
- Ultimate Game Music Collection
- Resonance Audio SDK for Unity v1.2.1

8.3 Python packages

absl-py	0.6.1	altgraph	0.16.1
astor	0.7.1	certifi	2018.10.15
Click	7.0	cycler	0.10.0
Cython	0.29	dist-keras	0.2.1
Flask	1.0.2	future	0.17.1
gast	0.2.0	grpcio	1.12.1
h5py	2.8.0	itsdangerous	1.1.0
Jinja2	2.10	Keras	2.2.4
Keras-Applications	1.0.6	Keras-Preprocessing	1.0.5
keyboard	0.13.2	kiwisolver	1.0.1
macholib	1.11	Markdown	3.0.1
MarkupSafe	1.0	matplotlib	3.0.1
mkl-fft	1.0.6	mkl-random	1.0.1
names	0.3.0	numpy	1.15.3
olefile	0.46	pandas	0.23.4
pefile	2018.8.8	Pillow	5.3.0
pip	10.0.1	protobuf	3.6.1
pygame	1.9.4	PyInstaller	3.4
pyparsing	2.2.2	PyQt5	5.11.2
PyQt5-sip	4.19.12	python-dateutil	2.7.5
pytz	2018.7	pywin32-ctypes	0.2.0
PyYAML	3.13	scipy	1.1.0
setuptools	39.1.0	six	1.11.0
tensorboard	1.11.0	tensorflow	1.11.0
termcolor	1.1.0	Theano	1.0.3
tornado	5.1.1	Werkzeug	0.14.1
wheel	0.32.2	wincertstore	0.2

8.4 Textures

Texture	Source
	Given on the 8.12.2018 by Yannick Pawils
	Retrieved on the 8.12.2018 from https://opengameart.org/node/7254

Bibliography

Eslami, S M Ali et al. (June 2018). "Neural scene representation and rendering". en. In: *Science* 360.6394, pp. 1204–1210.

Glorot, X and Y Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference*.

Ha, David and Jürgen Schmidhuber (Mar. 2018). "World Models". In: arXiv: `1803.10122 [cs.LG]`.

Ioffe, Sergey and Christian Szegedy (Feb. 2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: arXiv: `1502.03167 [cs.LG]`.

Peng, Xue Bin et al. (Oct. 2017). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization". In: arXiv: `1710.06537 [cs.RO]`.

List of Figures

2.1 Graphical representation of a neuron with 3 inputs . 5
Own graphic

2.2 Multilayer neural network structure 5
Own graphic

2.3 GQN architecture 11
own graphic

3.1 The neural network predicts different colors based on the input data that makes up R. The input pictures are shown to the right. The network gives output even if no images are fed in as seen in the last image. 14
own graphic

3.2 Training data is being captured in the functional prototype level. 15
own graphic

3.3 Top down prototype level as seen in the editor 17
own graphic

3.4 In the top down prototype, the network output gets blurry when the player walks off a platform. 18
own graphic

3.5 Certain objects can only be seen, when the player stands at certain locations. 19
own graphic

3.6 These are the 4 markers (explained in section 5.1) of the morphing environment. Overlapping areas are differently colored. 20
own graphic

3.7 Top down view of the level with only one object activated in the middle 21

own graphic
3.8 The 4 differnt objects in the center of the level 22
own graphic
3.9 The center object morphs into a different object when the player crosses from one marker into another. 23
own graphic

4.1 Checkpoint cuboid varying material color based on the current sound 26
own graphic
4.2 Goal varies its material color based on the current sound 27
own graphic
4.3 Speeder moves away from his pink spawn point . . . 29
own graphic
4.4 A variation of the speeder that moves faster 30
own graphic
4.5 A champer becomes visible because he emits a sound 31
own graphic
4.6 Two DumDums standing still because the silent part in their loop is played 32
own graphic
4.7 The player destroys an enemy with the laser. 34
own graphic
4.8 The player reveals where the walls are in the unity environment using smoke. 35
own graphic
4.9 The player reveals a champer using smoke. 36
own graphic

5.1 The keyout material is applied to everything that should not be visible. 41
own graphic
5.2 The network output is merged with the output of a Unity camera. 42
own graphic

Acknowledgments

I like to thank Yannick Pawils for our daily insightful discussions and for his feedback which was a great guidance to me.
Also I like to thank my professor Thomas Bremer for the feedback and guidance he provided.
Furthermore I like to thank Susanne Brandhorst, Jules Pommier and Matthias Mayer.

Statement of Authorship

I hereby declare that I am the sole author of this bachelor thesis and that I have not used any sources other than those listed in the bibliography- and resources section. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

....
(Place, Date) (Signature)

DEHIVE