Accelerating rendering by partial inpainting
BobRossNet

Individual Research Project

V. Hoveling
4591941

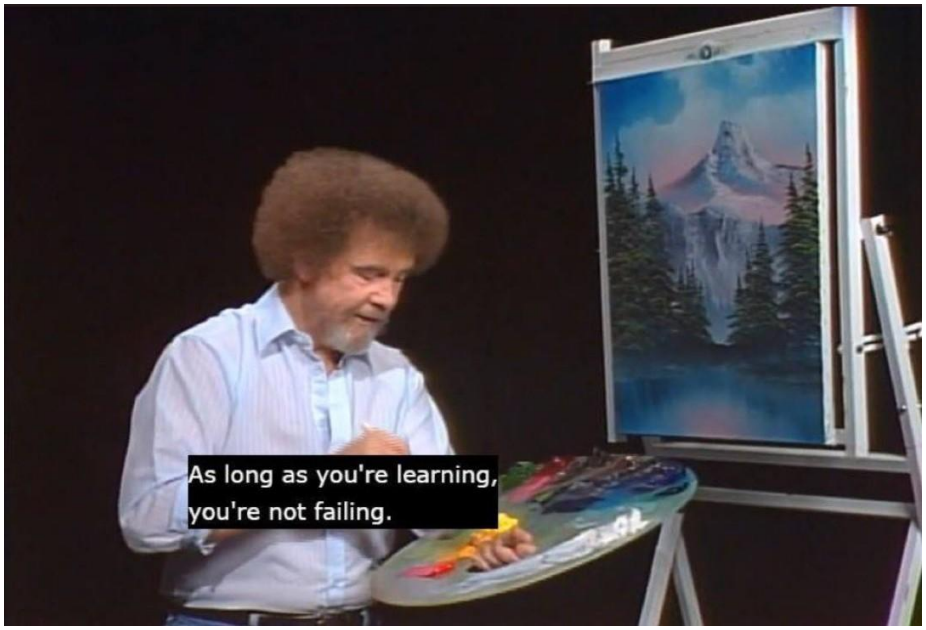
February 11, 2019

Abstract

Before you lies the report that resulted from the Individual Research Project of Vera Hoveling. For this project, a neural network has been developed to complete partially rendered images: BobRossNet. In this document you will find the details of the project, the architecture of BobRossNet, its training, results and a reflection on its future.

Contents

1 Introduction



This report presents and discusses the results of the Individual Research Project of Vera Hoveling: a neural network for image inpainting in partially rendered images to accelerate the image synthesis process of a resource-constrained ray-tracer. To achieve this, partially rendered images are completed with a neural network that follows a U-Net architecture. The final architecture is named BobRossNet.

This report will explain the background of the research problem and how we came to this solution (Chapter 2). Then it will detail key concepts regarding neural nets and their applications for denoising (Chapter 3), followed by descriptions of the design of BobRossNet (Chapter 4), the training of the network (Chapter 5) and its performance (Chapter 6). Finally, we conclude with a discussion of future work (Chapter 7).

2 The problem to solve

This section will provide the background of the project, elaborate on the problem to solve and explain the proposed solution.

2.1 Background

This research project has been conducted at Borges3D[1]. The Borges project has developed a rich language that can express any shape. To express the shapes, the language is not limited to triangle meshes for representation but also includes NURBS curves and implicit surfaces. For the web app that is part of the Borges project, it is required to visualize the shapes. As the language uses multiple representations, the rendering of the geometry can be cumbersome and requires expensive triangulation operations and painful encounters with WebGL frameworks. With the rise of cheap compute power and WebAssembly (Wasm), a new rendering solution came up: Integrate a raytracer in the heart of the Borges project and port it from C++ to Wasm so that it can be run in the browser!

2.2 Research problem

But Wasm is not *that* good yet, and compute power not *that* cheap yet, so that brings us to the particular problem that was investigated for this report: speeding up a raytracer that cannot render sufficient pixels for high quality images in time. To speed up the visual results, we propose the following: inpainting of missing pixels while the image converges by sampling random pixels, as depicted in Figure 2.1.

[1] www.borges.xyz

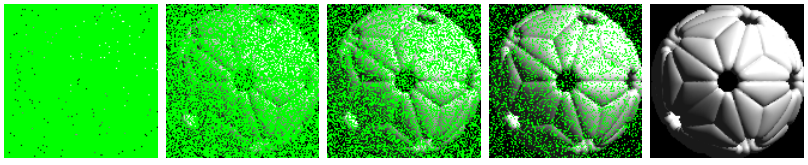


Figure 2.1: Converging to a complete image whilst sampling pixels over a uniform random distribution (green represents untraced pixels).

2.3 Approach

Inpainting relies on existing pixels to conclude on those that have not been captured. What makes inpainting problems usually so hard, is that the regions of missing pixels are irregularly defined. This is why we have chosen to sample the pixels randomly over the image; given a collection of randomized and noisy input pixels, the setup resembles a denoising problem. Denoising assumes that the existing pixels are corrupted by noise, which is to be removed, but the noise distribution can often be modeled, which helps the solution. Visually, the randomly sampled images are mostly noisy. Neural nets have been very successful at solving denoising problems recently. Inspired by those successes, we dediced to apply a neural net based on existing architectures that have proven success in denoising applications to our particular noisy inpainting problem.

3 Key Machine Learning Concepts

This chapter will shortly introduce key concepts in machine learning that are important for image restoration.

Convolutional Neural Net (CNN): A convolutional neural network (Fukushima & Miyake, 1982) is a neural net of which the layers perform convolutions on the input. The weights for the convolutional filters are learned by training the neural net and backpropagating the results, such that 'hand crafted' filters can be avoided. Convolutional nets have seen stunning results and applications in the field of computer vision, and in more recent years also in image processing.

Denoising CNN: The application of deep neural networks for image denoising arose from stacking a lot of convolutional layers and training for the particular application of denoising. This works well, but as convolutions are relatively expensive operations, a more optimized architecture was required for our purpose.

Autoencoder: An autoencoder (Rumelhart, Hinton, & Williams, 1986) is a convolutional neural net of which the first n layers of the network reduce the x and y dimension and increase the number of channels, usually by a combination of convolution and pooling. This way a sparse representation of the original image is obtained. From this sparse representation the end result is reconstructed, by increasingly expanding the x and y dimensions and decreasing the number of channels.

Skip Connections: In the early layers of a CNN, information is captured that can be used in later stage of the network. Skip connections (Long, Shelhamer, & Darrell, 2014) transfer this information to later stages of the network. This information would otherwise would have been lost. Skip connections can help with reconstruction and also prevent blurriness that is otherwise a typical result of CCN's.

U-net: A U-net (Ronneberger, Fischer, & Brox, 2015) is a neural net that follows the autoencoder architecture and has additional skip connections. The encoder-layers have skip connections to the matching decoder-layers, such that important information can be preserved between the layers.

4 Design

This chapter will explain the architecture of BobRossNet, optimization decisions and the implementation.

4.1 Structure of the network

The model is a small (10346 parameters) and fully convolutional network. It is important that the network is fully convolutional, because it makes the network applicable to images of arbitrary resolution. The architecture follows a U-net architecture. As U-net architectures have shown very promising results for denoising, this structure seemed a natural candidate. The network has two layers of encoding (3x3 convolution and 2x2 downsampling) and two layers of decoding (3x3 convolution and 2x2 upsampling). For the consecutive activation layers, we've chosen to apply the ReLu function (Nair & Hinton, 2010), for the non-saturation of its gradient accelerates the convergence of gradient descent (Krizhevsky, Sutskever, & Hinton, 2012). For the input of the network 5 channels were used, conveying information on missing pixels, grayscale color, depth and screen-space normals. The architecture is illustrated in Figure 4.3. The output of the network is a grayscale image.

4.2 Optimization

To keep the execution time of the network as short as possible, the model was stripped down to just 5 layers (7 including input and output layers) and several optimizations were implemented. The optimizations are discussed in this section.

4.2.1 Convolutional groups

Convolutional groups (Krizhevsky et al., 2012) were implemented in the model to decrease the number of parameters and speed up compute time. An example of grouped convolutions is illustrated in Figure 4.1. The size of a convolutional filter is defined by its width, height and the *number of channels* of the input it filters. As the network gets deeper, the number of channels increases, while the spatial dimensions decrease (with pooling). In a convolutional layer with `n` filter groups, each of the filters in each filter group are convolved with only `1/n` of the previous layer's channels. Had the network not applied convolutional groups in the network, it would have had 31946 parameters (as opposed to the 10346 it has now), roughly tripling its size.

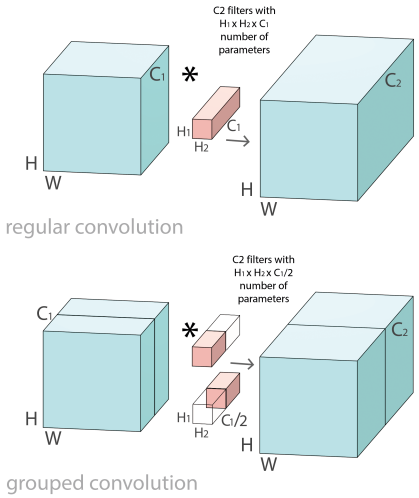


Figure 4.1: Grouped convolution filters are much smaller than their normal counterparts. With two filter groups (as depicted), each filter has exactly half the number of parameters of the equivalent normal convolutional layer.

4.2.2 Shuffling channels

In earlier stages of the design of BobRossNet, the convolution over the encoded representation (layer 3) was not done with grouped convolutions. To restrain the size of the network, the convolutional layer with 14420 parameters was replaced by grouped convolution as well, reducing the number of parameters to 3620. But stacking grouped convolution layers blocks the information flow between channel groups and that weakened the information gathered by the network. Shuffling the channels of the network restores the flow of information, as illustrated in Figure 4.2. In our model the shuffling is implemented in the encoded layer. Shuffling grouped convolution channels had been successfully applied in very small networks to increase accuracy previously (Zhang, Zhou, Lin, & Sun, 2017) and allowed in our network for better flow of information and more accurate predictions.

4.3 Implementation

The model was built with the Keras framework, using a TensorFlow backend. Keras is very suitable for rapid prototyping and provides a high level of abstraction. After training (Chapter 5), the models were loaded in a TensorFlow.js implementation that provided WebGL accelerated convolutional passes and hardware supported max-pooling.

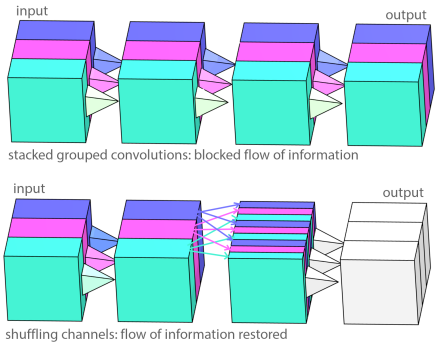


Figure 4.2: Stacking grouped convolution layers (indicated by coloring) keeps the information in the same channel groups. The flow of information can be restored by shuffling the channels.

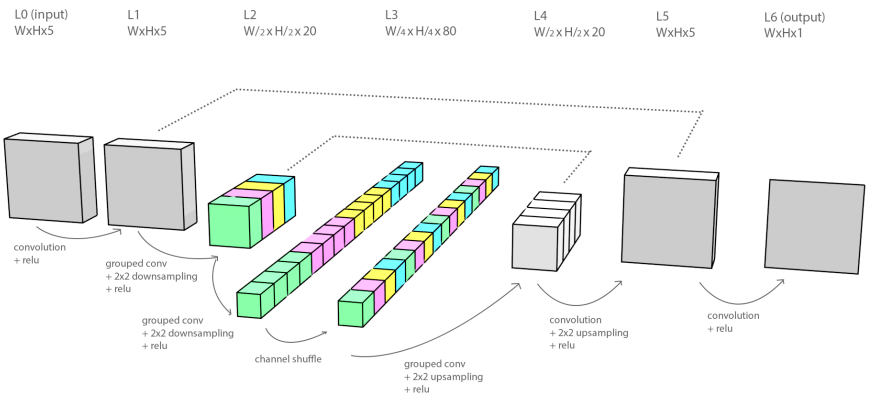


Figure 4.3: Illustration of the architecture of the model. All convolutions are padded 3x3 convolutions. The stacked convolution groups are color coded to show the flow of information in the network. Dotted lines represent skip connections.

5 Training

This chapter will elaborate on the dataset used for training, the loss function used and hightlight some details of the process of training the network.

5.1 Dataset

To obtain a meaningful model, the dataset used for training had to resemble the data it would be applied on as close as possible. Some observations on the application: The renderer is very limited in terms of shading: geometry has neither any materials nor colors, so standard Blinn-Phong shading suffices. Light positioning depends on the camera position. The camera rotates around the object and is orthogonal, so there are no changes in perspective. The pixels will be sampled over a uniform random distribution. The biggest variant is the geometry depicted. Some shapes will be very complex, some will be very simple. To mimick the application as close as possible and to encompass the variety of geometry, a collection of 479 3d models with varying complexity was used to construct the dataset.

The network was constrained to 10346 trainable parameters so overfitting was not an immediate concern. But as the model was also intended to handle different stages of convergence and a wide variety of shapes, there was a need for an exensive dataset. Taking into account the time-budget for training, a dataset of 50295 samples was made. The 479 3d models were rendered in 5 different stages of convergence (1%, 25%, 50%, 75% and 100% of pixels sampled over a uniform random distribution) and different resolutions. The resolutions used were 128*128, 256*256 and 512*512. Samples of higher resolution were then cut up into 128*128 patches for training, yielding 21 (1+4+16) samples per model per stage of convergence. Thus the dataset contained 50295 samples (469*21*5) of 5 layers data and a matching set of 50295 grayscale images for reference. Examples of what the training data looks like can be found in Figure 5.2.

Of the 50295 train/reference images, 1050 were kept separately for testing and of the remaining 49245 images, 90% was used for training and 10% was used for validation. The test images are rendered from 3d models that were not used to generate images in the training or validation sets.

5.2 Loss function

A loss function is a measure of how good a prediction model does in terms of being able to predict the expected outcome. When training our network, we rely

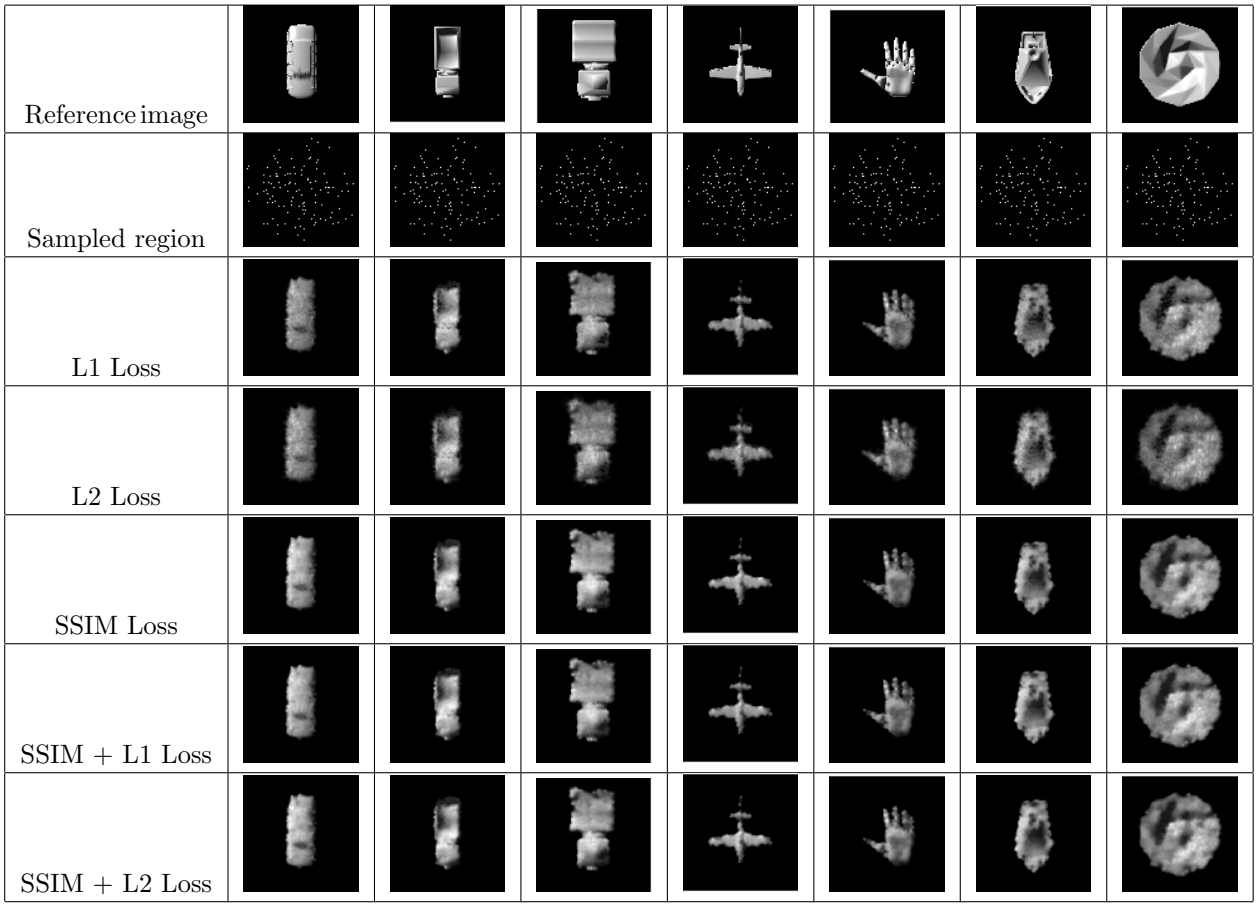


Figure 5.1: A comparison of loss functions. The results are over a small dataset of just 50 samples. The eventual model was trained on a much larger dataset, as described in section 5.1, but these small training examples were sufficient to distinguish the different characteristics that the different loss functions brought out.

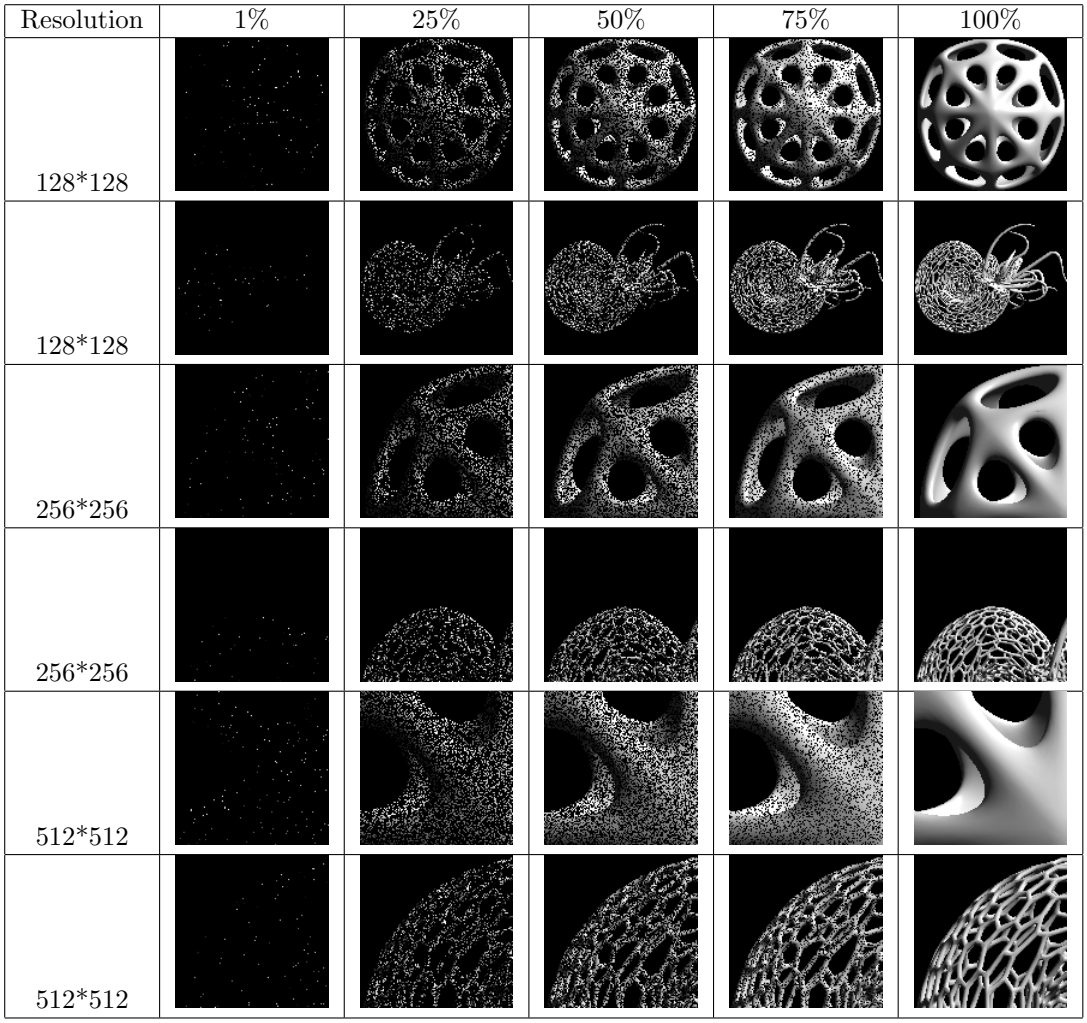


Figure 5.2: Samples from the dataset that was used for training. Pixels were sampled over a uniform random distribution. Take note that this entire dataset was used to train *one* model in order to obtain a model that could handle all different stages of convergence.

on the loss function to find the best parameters (weights) for the data. As the parameters, and therefore the results of the model, depend on the loss function, the loss function is an important choice.

To see which loss function would bring the most desirable outcome for our purpose, different loss functions have been tested. An elaborate discussion of loss functions for CNN's can be found in (Zhao, Gallo, Frosio, & Kautz, 2017), but the results we found are in line with their results. The loss functions that were tested are:

- Mean Absolute Error (MAE or L1)
- Mean Squared Error (MSE or L2)
- Structural Similarity Index (SSIM)
- A combination of SSIM and L1
- A combination of SSIM and L2

L1 and L2 losses are both used a lot in training a wide variety of neural nets and work quite well for images too. SSIM predicts the *perceived* quality of pictures by taking a weighted combination of luminance, contrast and structure over small patches of the image. SSIM values range between -1 and and 1, a higher value indicates higher similarity. To express this value in a loss function that can be minimized, the loss of the network is expressed as `abs(SSIMvalue - 1)`, summed up over the samples in the batch.

Figure 5.1 shows a comparison of the reference image, the sampled region and the results when training with different loss functions. As one can observe, loss functions that involve SSIM perform noticeably better on contrast and sharpness than L1 and L2 on their own. The differences between SSIM+L1 and SSIM+L2 are small but SSIM+L2 seems to work a little better on contrast (most noticeably in the leftmost column). Thus SSIM+L2 was chosen to train the models on the large dataset.

5.3 Training the network

In this section the details of the training of the network are discussed.

5.3.1 Learning rate

To facilitate the learning of the network without tedious hyper-parameter tuning, an adaptive learning rate algorithm called AdaDelta (Zeiler, 2012) was used to select the learning rate during training. AdaDelta modifies the learning rate at each time step for every parameter, based on a moving window of past gradient updates.

5.3.2 Mini-batch gradient descent

To reduce variance in the parameter updates, the network was trained with mini-batch gradient descent. The dataset is subdivided in 'mini-batches' of 32 samples and the parameters of the network are updated for each mini-batch of samples. This can help the network convergence in a stable manner. To facilitate mini-batch gradient descent with the large dataset, which did not fit into memory, a custom generator for the mini-batches was written in Keras.

5.3.3 Convolutional groups

We observed that introducing convolutional groups made the network significantly harder to train. Where prototyped earlier models that contained more parameters were more converging rapidly, the fewer parameters of the network required longer training to achieve a similar performance.

5.3.4 Implementation

The model was build and trained with Keras, an open source neural network library for Python. We used a TensorFlow backend that enabled training on the gpu. To produce the input for the first step, all input data was read from the pre-generated dataset. The data was stored in numpy arrays and contained input vectors with 5 channels per pixel.

6 Results

In this chapter the results of the final model are analyzed.

6.1 Performance

The application was benchmarked on a device with an Intel i7-4810MQ processor (2.80GH) and an NVIDIA GeForce GTX 870M gpu. The execution times of the model depend on the image size the model operates on. In Figure 6.1, we see how the prediction times of the model are a linear function of the image size. For medium resolution images (640x480) and smaller, we note prediction times of 80 ms and shorter, projecting possible framerates of 12 fps and faster.

6.2 Evaluation

A test set with 1050 samples was kept separately, so that the model can be tested on data it has never seen before. The results of the model on each stage of convergence, expressed in SSIM (see section 5.2) can be seen in Figure 6.2. The model performs very well from 25% of convergence on, as is also discussed in section 6.3. The model is clearly performing worst on images of which only 1% is traced, which is to be expected as these images contain the least information on the image to reconstruct.

6.3 Visual Results

For a visual analysis, we have included images that were rendered of the Stanford dragon[1] in different stages of convergence in Figure 6.3 and 6.4, with the grayscale channel of the input data to give an impression of the visual improvement by the network. As seen in 6.2, the model performs very well from 25% on and has a harder time reconstructing sparsely sampled images. Therefor we will separately discuss the results on sparsely sampled inputs, 1-10% and the densely sampled inputs, 25% and more.

6.3.1 Sparse inputs

The model is able to construct roughly the shape of the final image at 1% already. In Figure 6.3 we see quick improvements, with already much less blobbyness at 2% and the first appearance of details at 5%, such as the outline of

[1] http://graphics.stanford.edu/data/3Dscanrep/

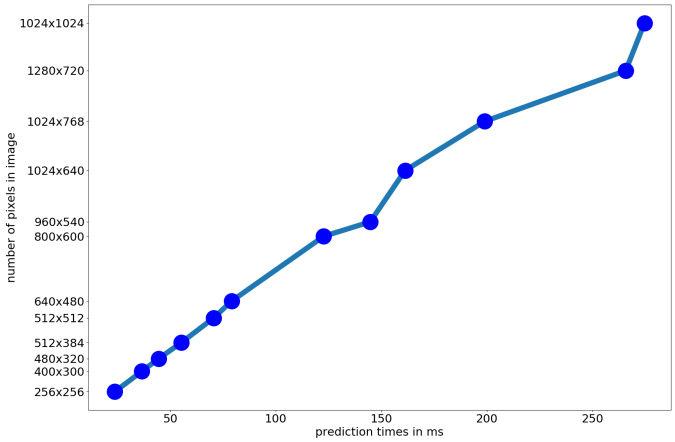


Figure 6.1: Prediction times of the model increase with the image size.

Convergence	1%	25%	50%	75%	100%
SSIM	0.789893	0.947632	0.970101	0.980241	0.987065

Figure 6.2: Evaluation of the model per stage of convergence. The values in the table are the mean SSIM values of the test samples.

the teeth and legs. At 10% the artifacts between the neck and body have disappeared and we can already see some of the texture of the the dragon. This observation is confirmed by calculating the SSIM of the 10% image with the reference image, which already gives an index of 0.90, indicating indeed a high similarity already.

6.3.2 Dense inputs

On the test set, the model has a high SSIM score for images that are sampled for 25% or more. We can see in Figure 6.4 that from 50% on, the output image barely changes anymore, as confirmed by the respective high similar SSIM scores for these stages in 6.2. When we compare the result of BobRossNet on the fully converged image (100%) to its unprocessed reference image, we do note slightly more contrast in the output image, but also a little blurriness. Despite the blurriness, the contours of the depicted geometry remain sharp.

6.4 Temporal stability

When designing the network, recurrent nodes (Chaitanya et al., 2017) were considered for the application to ensure that there would be not too much difference between two consecutive images, which might cause flickering. However, as it is

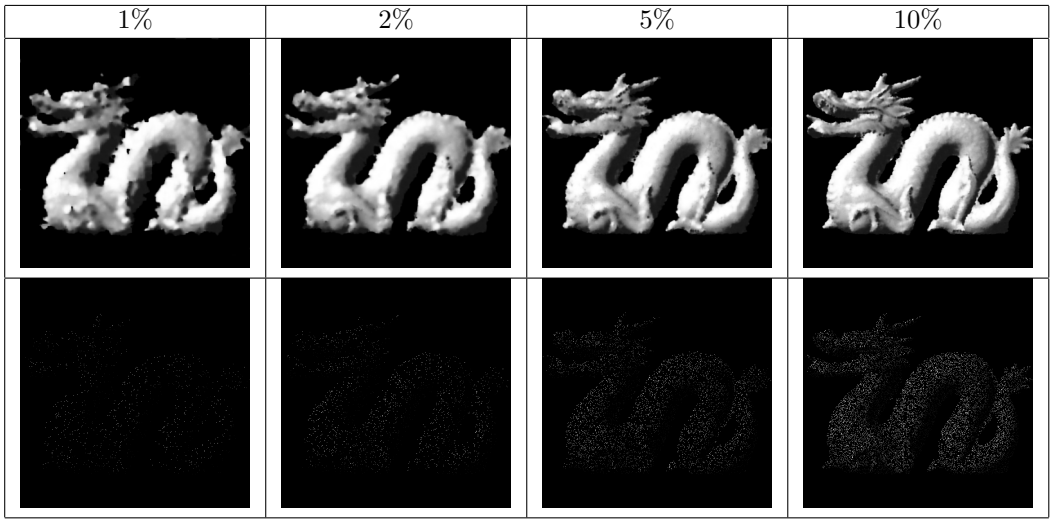


Figure 6.3: Top row: Results from the network on sparsely sampled images (1 to 10 percent). Bottom row: the grayscale channel of the input data.

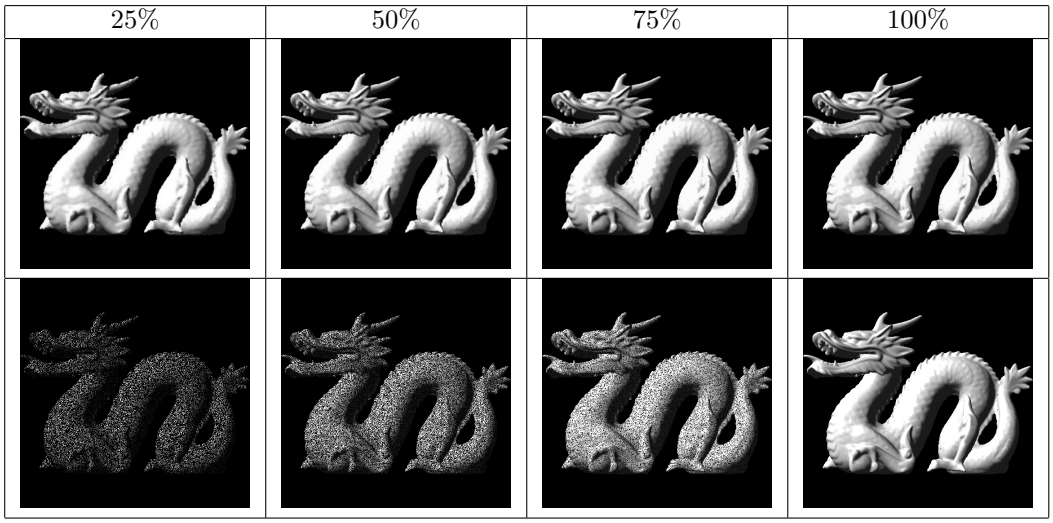


Figure 6.4: Top row: Results from the network on densely sampled images (25 to 100 percent). Bottom row: the grayscale channel of the input data.

now, the application shows surprising temporal stability. The image seems to converge in a very smooth manner.

7 Future Work

This chapter will explore three possible areas of future work: Sampling prediction, adding material support and training separate models for sparse and densely sampled images.

7.1 Sampling prediction

Keeping in mind the original intent of this project, speeding up a raytracer, one could not only use a network for inpainting, but also apply an additional network to predict which pixels are most relevant to trace next. This could speed up the (visual) convergence of the image even more.

7.2 Materials

The network was developed for a particular application, with very limited materials. However, we would be very interested in studying how the model can be adapted to also support materials that vary in color, specularity and maybe even transparency and reflectivity. We are, after all, speeding up a raytracer.

7.3 Multiple models

Currently we apply one and the same network on each input, whether 1% of the pixels is known or 99%. To further improve the accuracy of the predictions, the network could be trained twice, once with a dataset containing only sparsely sampled images (1-24%) and one with a dataset containing more densely sampled images (25% and more). The resulting two models can be applied for predicting respectively sparse and dense inputs. We suspect that this will improve accuracy on the sparsely sampled images, while maintaining high accuracy on the densely sampled images, for which we've already seen great results.

8 Acknowledgements

This is a very brief word to say thank you to Elmar Eisemann (TU Delft), Jochem van der Spek (Borges3D), Mathijs de Weerdt (TU Delft), Jacques Povee (Endless Emotional Support) and Marijn Roelvink (Spark Of Inspiration) for enabling me to have the most interesting, enjoyable and educational Individual Research Project.

Bibliography

Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., & Aila, T. (2017, July). Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph. 36*(4), 98:1–98:12. doi:10.1145/3072959.3073601

Fukushima, K. & Miyake, S. (1982, January). Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, *15*(6), 455–469. doi:10.1016/0031-3203(82)90024-3

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 25* (pp. 1097–1105). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Long, J., Shelhamer, E., & Darrell, T. (2014). Fully convolutional networks for semantic segmentation. *CoRR*, *abs/1411.4038*. arXiv: 1411.4038. Retrieved from http://arxiv.org/abs/1411.4038

Nair, V. & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on international conference on machine learning* (pp. 807–814). ICML'10. Haifa, Israel: Omnipress. Retrieved from http://dl.acm.org/citation.cfm?id=3104322.3104425

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: convolutional networks for biomedical image segmentation. *CoRR*, *abs/1505.04597*. arXiv: 1505.04597. Retrieved from http://arxiv.org/abs/1505.04597

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Parallel distributed processing: explorations in the microstructure of cognition, vol. 1. In D. E. Rumelhart, J. L. McClelland, & C. PDP Research Group (Eds.), (Chap. Learning Internal Representations by Error Propagation, pp. 318–362). Cambridge, MA, USA: MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=104279.104293

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, *abs/1212.5701*. arXiv: 1212.5701. Retrieved from http://arxiv.org/abs/1212.5701

Zhang, X., Zhou, X., Lin, M., & Sun, J. (2017). Shufflenet: an extremely efficient convolutional neural network for mobile devices. *CoRR*, *abs/1707.01083*. arXiv: 1707.01083. Retrieved from http://arxiv.org/abs/1707.01083

Zhao, H., Gallo, O., Frosio, I., & Kautz, J. (2017, March). Loss functions for image restoration with neural networks. *IEEE Transactions on Computational Imaging*, *3*(1), 47–57. doi:10.1109/TCI.2016.2644865