Neural Network Ambient Occlusion

Figure 1: *Comparison showing NNAO (our method) enabled and disabled, as implemented in a game engine.*

Abstract

We present Neural Network Ambient Occlusion (NNAO), a fast, accurate screen space ambient occlusion algorithm that uses a neural network to learn an optimal approximation of the ambient occlusion effect. Our network is carefully designed such that it can be computed in a single pass - allowing it to be used as a drop-in replacement for existing screen space ambient occlusion techniques.

Keywords: neural networks, machine learning, screen space ambient occlusion, SSAO, HBAO, game development, real time rendering

1 Introduction

Ambient Occlusion is a key component in the lighting of a scene yet is expensive to calculate. By far the most popular approximation used in real-time applications is Screen Space Ambient Occlusion (SSAO), a method which uses the depth buffer and other screen space information to calculate occlusions. Screen space techniques have seen wide adoption because they are independant of scene complexity, simple to implement, and fast to compute.

Yet, calculating effects in screen space often creates artifacts as these techniques lack the full information about the scene. The exact behaviour of these artifacts can be difficult to predict. We use machine learning to learn a SSAO algorithm that minimises these errors with respect to some cost function.

We build a database of camera depths, normals, and *ground truth* ambient occlusion as calculated using an offline renderer, and use a neural network to learn a mapping from the depth and normals surrounding the pixel to the ambient occlussion of that pixel. Once trained we convert the neual network into an optimised shader which is more accurate than existing techniques, has better performance, no user parameters other than the occlusion radius, and can be computed in a single pass allowing it to be used as a drop-in replacement.

Our contribution is:

- A technique for applying machine learning to screen space effects such as SSAO.
- A fast, accurate SSAO shader that can be used as a drop in replacement to existing techniques.

2 Related Work

Screen Space Ambient Occlusion Screen Space Ambient Occlussion (SSAO) was first introduced by [Mittring 2007] for use in *Cryengine2*. The approach samples around the depth buffer in a view space sphere and counts the number of points which are inside the depth surface to estimate the occlusion. This method has seen wide adoption, but often produces artifacts such as dark halos around object silhouettes, or white highlights on object edges. [Filion and McNaughton 2008] (SSAO+) sampled in a hemisphere around the pixel oriented in the direction of the surface normal. This removed the artifact related to white highlights around object edges and reduced the required sampling count, but still sometimes produced dark halos. [Bavoil et al. 2008] introduced Horizon Based Ambient Occlusion (HBAO), a technique which predicts the occlusion amount by estimating how closed or open the horizon is around the sample point. Rays are regularly marched along the depth buffer and the difference in depth used to calulate the horizon estimate. This was extended by [Mittring 2012] improving the performance using paired samples. This produces a more realistic effect but does not account for the fact that the camera depth map is an approximation of the true scene geometric. [McGuire et al. 2011] introduced Alchemy Screen-Space Ambient Obscurance (ASSAO), an effect which substitutes a fixed falloff function into the general lighting equation to create a more physically accurate integration over the occlussion term. ASSAO produces a physically based result, but still does not deal directly with the errors introduced by the screen space approximation.

Machine Learning for Screen Space Effects So far, machine learning has seen very limited application to rendering and screen space effects. In offline rendering [Kalantari et al. 2015] used machine learning to filter the noise produced by monte carlo rendering at low sample rates. [Ren et al. 2015] used neural networks to perform image space relighting of scenes, allowing users to virtually adjust the lighting of scenes even with complex materials. Finally [Johnson et al. 2011] used machine learning alongside a large repository of photographs to improve the realism of renderings - adjusting patches of the output to be more similar to cooresponding patches of photographs in the database.

3 Preprocessing

To produce the complex scenes required for training our network we make use of the geometry, props, and scenes of the Open Source first person shooter Black Mesa [Crowbar-Collective]. We take several scenes from the game and add additional geometry and clutter to ensure a wide variety of objects and occlusions are present.

We produce five scenes in this way, and select 100-150 viewpoints from which to render each scene using different perspectives and camera angles. From each viewpoint we use *Mental Ray* to render scene depth, camera space normals, and global ambient occlusion at a resolution of 1280×720. From each render, we randomly pick 1024 pixels, and perform the following process:

Given a pixel's depth we use the inverse camera projection matrix to calculate the position of the pixel as viewed from the camera (the *view space position*). We then take $w \times w$ samples in a view space regular grid centered around this position and scaled by the user given AO radius r. In this project we set $w = 31$. We reproject each sample into the screen space using the camera projection matrix and sample the GBuffer to find the cooresponding pixel normal and depth. For each sample we take the difference between its normal and that of the center pixel. Additionally we take the difference between its view space depth and that of the center pixel. These values we put into a four dimension vector. We then calculate the view space distance of the sample to the center pixel, divide it by the AO radius r, subtract one, and clamp it in the range zero to one. This value we use to scale the four dimensional input vector and ensures that samples outside of the occlusion radius are always zero and cannot have influence over the output. We concatenate these values from each sample into one large vector. This represents a single input data point $\mathbf{x} \in \mathbb{R}^{w^2 4}$. We then take the center pixel ambient occlusion value as a single cooresponding output data point $\mathbf{y} \in \mathbb{R}^1$.

Once complete we have a final dataset of around 500000 data points. We normalise the data by subtracting the mean and dividing by the standard deviation.

4 Training

Our network is a simple four layer neural network. The operation of a single layer $\Phi(\mathbf{x})_n$ is described by the following equation

$$\Phi(\mathbf{x})_n = PReLU(\mathbf{W}_n\,\mathbf{x} + \mathbf{b}_n, \alpha_n, \beta_n) \tag{1}$$

where $PReLU(\mathbf{x}, \alpha, \beta) = \beta\, max(\mathbf{x}, 0) + \alpha\, min(\mathbf{x}, 0)$ is a variation on the *Parametric Rectified Linear Unit* first proposed by [He et al. 2015] but with an additional scaling term β for the positive activation. The parameters of our network are therefore given by $\theta = \{\mathbf{W_0} \in \mathbb{R}^{w^2 4 \times 4}, \mathbf{W_1} \in \mathbb{R}^{4\times 4}, \mathbf{W_2} \in \mathbb{R}^{4\times 4}, \mathbf{W_3} \in \mathbb{R}^{4\times 1}, \mathbf{b_0} \in \mathbb{R}^4, \mathbf{b_1} \in \mathbb{R}^4, \mathbf{b_2} \in \mathbb{R}^4, \mathbf{b_3} \in \mathbb{R}^1, \alpha_\mathbf{0} \in \mathbb{R}^4, \alpha_\mathbf{1} \in \mathbb{R}^4, \alpha_\mathbf{2} \in \mathbb{R}^4, \alpha_\mathbf{3} \in \mathbb{R}^1, \beta_\mathbf{0} \in \mathbb{R}^4, \beta_\mathbf{1} \in \mathbb{R}^4, \beta_\mathbf{2} \in \mathbb{R}^4, \beta_\mathbf{3} \in \mathbb{R}^1\}$.

The cost function of our network is given by the following, which consists of the mean squared error and a small regularisation term controlled by the constant γ which we set to 0.01.

$$Cost(\mathbf{x}, \mathbf{y}, \theta) = \|\mathbf{y} - \Phi_3(\Phi_2(\Phi_1(\Phi_0(\mathbf{x}))))\|^2 + \gamma\,|\theta| \tag{2}$$

Using this function, the parameters of the network are learned via stochastic gradient descent. In minibatches of 16 random elements of our dataset are passed through the network and the parameters updated using derivatives calculated from Theano [Bergstra et al. 2010] and the adaptive gradient descent algorithm *Adam* [Kingma and Ba 2014]. To avoid overfitting, we use a *Dropout* [Srivastava et al. 2014] of 0.5 on the first layer. Training is performed for 100 epochs and takes around 10 hours on a NVIDIA GeForce GTX 660 GPU.

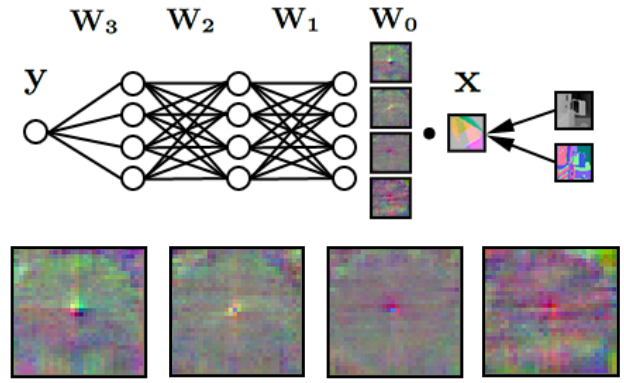

Figure 2: *Top: overview of our neural network. On the first layer four independant dot products are performed between the input and* $\mathbf{W}_0$ *represented as four 2D* filters. *The rest of the layers are standard neural network layers. Bottom: larger visualisation of* $\mathbf{W}_0$ *represented four filters.*

5 Filters

After training, the steps performed in the data preprocessing and neural network forward pass need to be reproduced in a shader for use at runtime. The shader is mostly a straight forward translation of the preprocessing and neural network steps with a few exceptions. This shader is provided in the supplimentary material for complete reference.

As the total memory required to store the network weight $\mathbf{W_0}$ exceeds the maximum memory reserved for local shader variables it cannot be stored in the shader code. Instead we observe that multiplication by $\mathbf{W_0}$ can be described as four independant dot products between columns of the matrix and the input $\mathbf{x}$. As the input $\mathbf{x}$ is produced by sampled in a grid, these dot products can be performed in 2D, and the weights matrix $\mathbf{W_0}$ stored as four 2D textures called *filters*. These 2D textures are then sampled and multiplied by the cooresponding parts of the input vector (see Fig. 2).

Performing the dot product in 2D also allows us to approximate the multiplication of $\mathbf{W_0}$. We can take fewer samples of the filter images and afterwards rescale the result using the ratio between the number of samples taken and the full number of elements in $\mathbf{W_0}$. We use stratified sampling - regularly picking every nth pixel of the filters and multiplying by the cooresponding input sample, finally multiplying the result by n. We also introduce a small amount of 2D jitter using random noise to spread the approximation over the image. This allows us to accurately approximate the multiplication of $\mathbf{W_0}$ at the cost of some noise in the output. To cope with this, as with other SSAO algorithms, the output is post processed using a bilateral blur.

6 Results

In Fig. 3 we visually compare the results of our method to SSAO+ (with 16 samples) and HBAO (with 64 samples), and to the ground truth. HBAO in general produces good results, but in many places it creates areas which are too dark. See: under the sandbags, behind the furniture, inside the car, between the railings, on the stairs, behind the pillar to the left of the truck. Additionally HBAO requires almost twice the runtime of our method to produce acceptable results (See Fig. 4).

In Fig. 1 we implement our method in a game engine. This

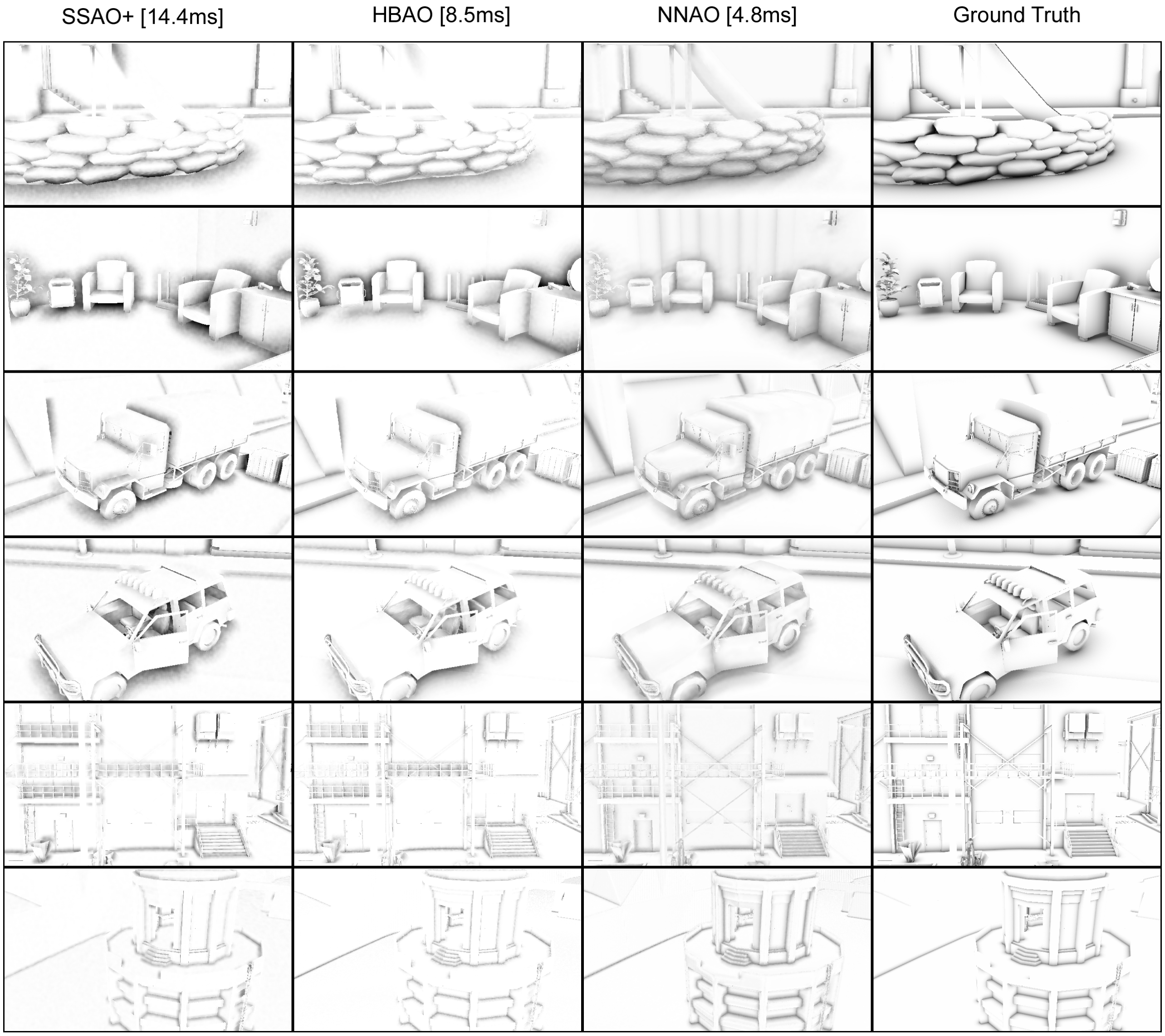

Figure 3: *Comparison to other techniques. From left to right: SSAO+, HBAO, NNAO (Our Method), Ground Truth.*

shows that the trained network is generalisable beyond the situations present in the training data and that it works well in interactive applications. Please see supplementary video for a long demonstration of this.

In Table. 1 we perform a numerical comparison between our method and previous techniques. Our method has a lower mean squared error on the test set with comparable or better performance to previous methods. All measurements are taken at half resolution renderings (640×360) on a NVIDIA GeForce GTX 660 GPU. Due to the unpredictability in measuring GPU performance absolute runtimes may vary in practice, but the quality of NNAO remains high even with a reduced sample count.

7 Discussion

In Fig. 5 we visualise what is being learned by the neural network. We show the activations of the first three hidden units using the Cyan-Yellow-Magenta channels of the image. Each unit learns a separate component of the occlusion with cyan learning unoccluded areas, magenta learning the occlusion of horizontal surfaces and yellow learning the occlusion of vertical surfaces.

Our method is capcable of performing many more samples than other methods in a shorter amount of time. Primarily this is because it samples in a regular grid, which gives it excellent cache performance, but also there is no data dependancy between samples which gives it a greater level of parallelism. Finally each sample is re-used by each filter which results in less noise.

7.1 Limitations & Future Work

Our method is training on data that does not includes high detail normal maps in the GBuffer. Although our method can be used on GBuffers with detailed normals (see Fig. 1) it is likely our method would perform even better in this case if trained on this kind of data.

Reducing the sampling count of our method below 64 does not tend to significantly reduce the runtime. This may be due to the opera-

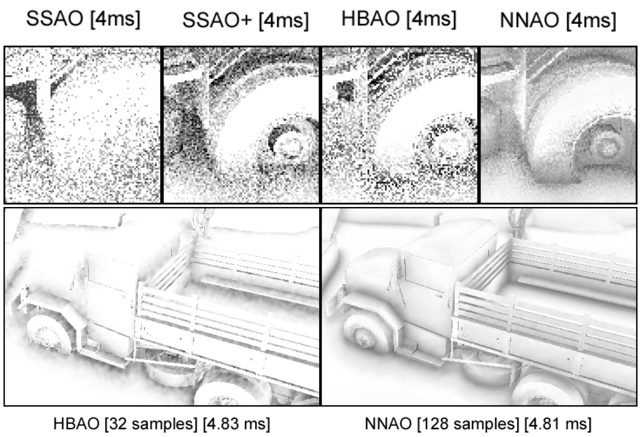

Figure 4: *Given similar runtimes, our algorithm can perform more samples producing less noise and a better quality output - as opposed to HBAO which appears blotchy at low sampling rates.*

Algorithm	Sample Count	Runtime (ms)	Error (mse)
SSAO	4	1.20	1.765
SSAO	8	1.43	1.558
SSAO	16	14.71	1.539
SSAO+	4	1.16	0.974
SSAO+	8	1.29	0.818
SSAO+	16	14.46	0.811
HBAO	16	3.53	0.965
HBAO	32	4.83	0.709
HBAO	64	8.50	0.666
NNAO	64	4.17	0.516
NNAO	128	4.81	0.497
NNAO	256	6.87	0.494

Table 1: *Numerical comparison between our method and others.*

tions of the other layers which still need to be performed. For some more constrained applications this may not be ideal. Further control over the performance in this case is something that interests us.

Our technique produces ambient occlusion but we see no reason why it could not be applied to other screen space effects such as Screen Space Radiosity, Screen Space Reflections and more.

7.2 Conclusion

We present a technique for performing Screen Space Ambient Occlusion using a neural network. After training we create an optimised shader that reproduces the network forward pass efficiently and controllably. Our method produces fast, accurate results and can be used as a drop-in replacement to existing Screen Space Ambient Occlusion techniques.

References

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 Talks*, ACM, New York, NY, USA, SIGGRAPH '08, 22:1–22:1.

BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. 2010. Theano: a CPU and GPU math expression compiler. In *Proc. of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

CROWBAR-COLLECTIVE. Black mesa. `http://www.blackmesasource.com/`.

FILION, D., AND MCNAUGHTON, R. 2008. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, ACM, New York, NY, USA, SIGGRAPH '08, 133–164.

HE, K., ZHANG, X., REN, S., AND SUN, J. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR abs/1502.01852*.

JOHNSON, M. K., DALE, K., AVIDAN, S., PFISTER, H., FREEMAN, W. T., AND MATUSIK, W. 2011. Cg2real: Improving the realism of computer generated images using a large collection of photographs. *IEEE Transactions on Visualization and Computer Graphics 17*, 9 (Sept), 1273–1285.

KALANTARI, N. K., BAKO, S., AND SEN, P. 2015. A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2015) 34*, 4.

KINGMA, D. P., AND BA, J. 2014. Adam: A method for stochastic optimization. *CoRR abs/1412.6980*.

MCGUIRE, M., OSMAN, B., BUKOWSKI, M., AND HENNESSY, P. 2011. The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 25–32.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses*, ACM, New York, NY, USA, SIGGRAPH '07, 97–121.

MITTRING. 2012. The technology behind the "unreal engine 4 elemental demo". In *ACM SIGGRAPH 2012 Talks*, ACM, New York, NY, USA, SIGGRAPH '12.

REN, P., DONG, Y., LIN, S., TONG, X., AND GUO, B. 2015. Image based relighting using neural networks. *ACM Trans. Graph. 34*, 4 (July), 111:1–111:12.

SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res. 15*, 1 (Jan.), 1929–1958.

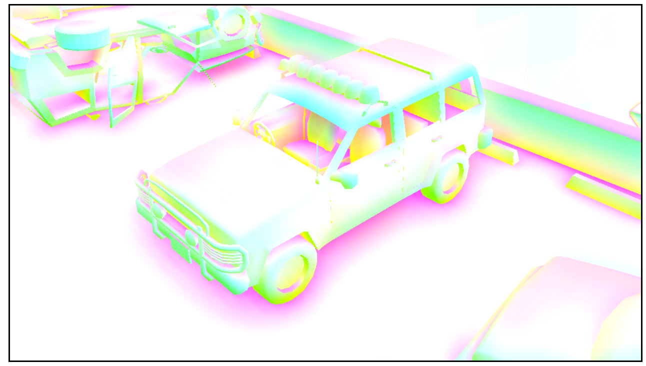

Figure 5: *The activations of the first three filters represented by the cyan, yellow, and magenta channels of the image.*