Bournemouth University

MSc CAVE - Master's Thesis - NCCA

Demystifying Denoisers: A Simple Pipeline for Denoising Renders with Neural Networks

Chris Leu

August 19, 2018

Contents

0.1 Introduction . . . 2
0.2 Previous Work . . . 2
0.3 Technical Background . . . 3
0.4 Implementation . . . 4
0.4.1 Setup . . . 4
0.4.2 Maya Script . . . 4
0.4.3 Additional Files . . . 5
0.4.4 Training . . . 5
0.5 Results . . . 6
0.6 Future Work and Conclusion . . . 7
0.7 Appendix . . . 9

0.1 Introduction

Image noise is as undesirable as it is pervasive in visual effects and videography as a whole. Causes of image noise can be lighting and camera equipment in the live-action realm, or sampling rates and material properties in computer generated scenes. This project will focus on the latter. The noise caused by under sampling in monte carlo renders is a double-edged sword, on one hand the solution is extremely simple: up your samples; however, this leads to extremely long renders. The workflow at Bournemouth University is reliant on CPU renders that can be slowed down by multiple courses vying for time on one render farm, therefore render times will always be a serious prohibiting factor when students are working on projects. Denoising as a post-process could allow students to maintain high render quality and give them the time they need to do multiple iterations of a project on a tight schedule.

When examining the field of removing image noise created in renders, the denoiser has the possible benefit of utilizing different passes and arbitrary output values (AOVs) to understand the image better. However, this does add some difficulty in use because it places restrictions on how the artist can render their scene. CG denoising solutions can also be heavy in terms of libraries and supporting scripts they require to run, meaning the time to setup such a system on each machine could end up being quite expensive if the artist is left to figure it out on their own. Most denoisers are written in separate python or matlab files meaning at the very least there will be some tedious file management involved. An ideal denoiser would not only improve the image, but also integrate itself seamlessly into the workflow already in place. The goal of this project will be to take a very powerful denoiser built from "Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings" [1], and then do the necessary modification and augmentations to make something that fits into Bournemouth University's unique pipeline.

0.2 Previous Work

Image denoising is not a new task, even before monte carlo renders people were looking to remove imperfections, known as grain, from analog film using similar techniques to ones used in the digital age. Solutions to image noise can be a simple as blurring, ie averaging a pixel's color based on its neighbors. Many advanced denoisers inherit from this simple principle by creating small filters that give localized blurring and preserve the edges of objects in a way a gaussian blur function would not. This process is commonly known as bilateral filtering. The "bi" referring to the idea of correcting a noisy pixel by looking at both pixels that are nearby, and pixels that similar to it (regardless of where it may be positioned in image space). These methods falter when seeking to understand the difference between noise and detail. Thinking of how to determine between a sandy beach and a noisy tan surface is a common example of this problem. Conveniently for VFX applications, using multiple frames of a shot can help illuminate which is persistent detail and which is random noise in cases like this. Even when given a scene with clear cut objects, filtering methods such as this can produce color banding or aura artifacts (see Fig. 1).

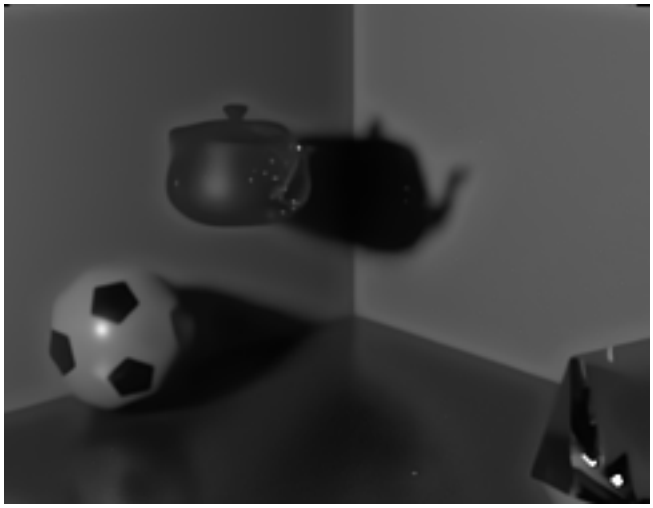

Figure 1: Aura effect in denoised image

The BM3D algorithm is one of the best image denoising technologies from before neural networks swept across the industry. Similar to the previously discussed filtering methods, it relies on finding similar parts (called blocks) within images [2]. These blocks are grouped via matching, which takes one block as a reference and compares it to other blocks, computing a dissimilarity metric. K-means clustering is discussed as being a slightly suboptimal choice for selecting similar blocks, but because of its ubiquity the reader may find it helpful to understand what the algorithm is doing. All blocks that score within a particular threshold are matched with the reference, thus turning the reference into the relative centroid of its set of blocks. These 2D blocks are then stacked on eachother creating a 3D array. Taken simplistically, the average of these blocks becomes a denoised block (see Fig. 2). In the full algorithm, this process is done twice, the first time applies a hard-threshold of the transform coefficients, and the second using Weiner filtering where the output of the first round is used as the true energy spectrum (see Appendix for flowchart).

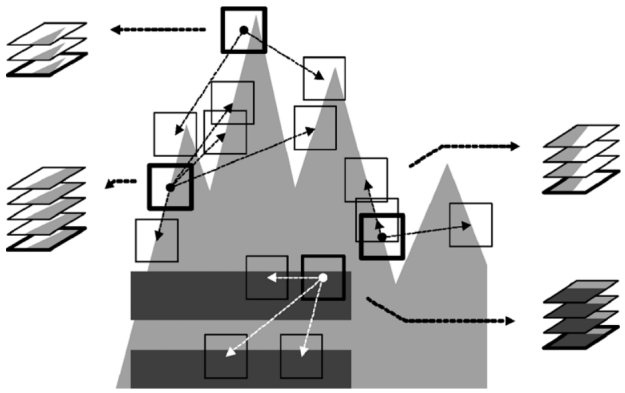

Figure 2: Grouping Blocks in an example (theoretical) image [2]

The next great leap in denoising (and image processing in general) was the neural network boom of the early 2010s. The creation of the ImageNet Database helped researchers have a consistent and extensive source of training data [3]. When training on photographs as opposed to renders, most neural networks focused on removing gaussian noise. The training process typically involves adding random noise to the image and then using the original image as the ground-truth for training. A training routine that uses images that are already clean cuts down on the significant cost of rendering noisy and clean image pairs for monte carlo denoising. However, there are now networks looking to learn how to clean images by looking only at noisy images, an idea especially important to denoising medical imagery, where generating a clean picture as a ground truth is often impossible [8].

"Beyond Gaussian Denoising" describes a convolutional neural network called DnCNN, that not only showcases a strong advancement in denoising, but also a certain amount of extra utility previous systems could not produce [7]. The convolution neural networks involve convolving a filter, which is essentially a patch of pixels, across an image to learn more about it. Initially these networks were used for image classification, and the layers of the network would allow them to develop filters that represented objects ranging from the concept of an edge, to something more domain specific like a human face. DnCNN is a relatively deep network with seventeen hidden layers, the slower training time for such a deep network is combated by utilizing batch normalization and residual learning. The paper also highlights how GPU acceleration plays an important role in the usability of this system, both for improving the time to denoise an image, but more importantly to dramatically cut down how long it takes to train (1-3 days). Because of the nature of the network to not be connected to a specific task, the network is also excellent at super-resolution and jpeg correction, provided the user has the right training set.

0.3 Technical Background

The denoiser to integrate into the BU pipeline is KPCN from the paper mentioned in the introduction. The network begins by splitting up the image into two parts, the specular and the diffuse. This is important not only because it is a significant departure from the denoisers that have been previously discussed, but also because it admits that the noise profile for specular and diffuse components are fundamentally different and deserve to be accounted for separately. The diffuse component is not simply the diffuse pass, but rather the diffuse pass divided (Hadamard division) by the albedo. This creates a sort of normalization to the image, as parts that reflect little light will be raised by dividing by a small albedo, and parts that reflect a large amount of light will be crushed by the large albedo. The authors do admit this step is not entirely necessary, as diffuse noise is not that challenging to correct, however it does allow for larger filtering kernels in practice, more on that later. The second half of the picture is the specular component, which is simply the specular component with a log transformation such that, specular = log(1+c specular). This helps deal with artifacts in areas with high dynamic range.

Like the DnCNN this is a convolution neural network, however since in the previous step we have two parts to our image, two CNNs are trained, one for each part. Each of the CNNs have no fully connected layers, and utilize the ReLU activation function, both of which are standard practices for similar networks. Each layer, l, of the network can be understood as $z^l = f^l(W^l * z^{l-1} + b^l)$, where W^l and b^l are the weights and biases, z^{l-1} is the output of the previous layer, and $f^l()$ is the ReLu activation function($f^l(x) = max(0, x)$).

The output of these two CNNs can either be a denoised pixel, as is traditionally the case, or, as is the namesake of the paper, a kernel for denoising said pixel. The authors found that if the network trained to output a 21 x 21 kernel to be applied to a pixel, it converged much faster than if the goal was to output the denoised pixel. However, if infinite time is given to train both types of CNN, they do give nearly identical denoised images. This idea is pulling from the older practice of attempting to create localized blurring, except now it is done at a much finer level and the blur is taylor-made for the noisy pixel by the network.

At the end, the kernel is applied to the pixel to denoise it, and the preprocessing functions for the diffuse and specular parts are inverted and combined to give a denoised image. See the appendix for a visual guide to the entire process from the authors.

One of the primary authors of "Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings" also released an implementation of KPCN using TensorFlow, an open source framework for machine learning. It is written in python and requires a few common extra libraries such as Numpy and OpenEXR. The implementation relies on the Tungsten

renderer, which was also created by the author, however, as the paper, notes, retraining allows the network to be utilized by Renderman as well (and theoretically other renderers). Thus, to utilize the implementation effectively the network needed to be retrained so the Tungsten compatible weights could be replaced by ones trained for Renderman.

The current pipeline of Bournemouth University is also extremely important to this project. The graduate program is primarily composed of three courses which use Red Hat Linux or Windows. The three most important pieces of software are Maya, Houdini, and Nuke. Students may use Vray, Arnold, Mantra, or Renderman for their projects. The paper discussing KPCN was implemented on Ubuntu and CentOS with the Tungsten and Renderman renders, thus the primary use-case for the denoiser would be a student on Red Hat using Maya with Renderman.

Students at Bournemouth also do not have admin privileges on their machines thus installing new programs is out of reach for most. A helpful tool to get around this issue is VirtualEnv which is installed on all Linux machines, this allows a user to install the necessary python libraries including TensorFlow.

0.4 Implementation

0.4.1 Setup

There are two primary components for this denoiser to work, a shell script to set up the virtual environment, and a python script to run from inside Maya to render something. The original download for the KPCN implementation was quite large (˜1.5Gb), and that did not include the supporting files such as tensorflow. A simple shell script was implemented to allow the artist to recieve something extremely lightweight (˜2Kb), that could be sent in an email, and it would then download all necessary components at the artist's convenience.

The script starts by creating a virtual environment in the user's home directory. In a production environment this might be a problem, but students will only be using their machine for a year's course of study, so they can afford to clutter up the home directory a bit. The use of a virtual environment also ensures isolation for this aspect of the pipeline. If the artist finds they do not want to use the denoiser after a few tests, they can guarantee that the install hasn't created any conflicts with other paths or directories. The script then installs tensorflow and the necessary python libraries. Pip is used for this step, as it is the standard at BU for installing python libraries. Github was chosen as the repository for program files because of its familiarity those in the tech industry, and because it was free to use. This download and unzip are the most time consuming aspect of the setup, the folder to be downloaded is about 1 GB, but a quick network could make it fairly trivial. In order to enable batch rendering from the command line in maya, a few lines must be added to the user's bashrc. This is the most invasive part of the installation, as it happens outside the virtual environment, however, in all tests done during the course of the project these added lines did not affect Maya or any other program negatively in any way. The key line in the bashrc additions is adding to the MAYA_RENDER_DESC_PATH, which lets the Renderman .xml file become a valid renderer to Maya's command line rendering system.

0.4.2 Maya Script

The user can then open any scene in Maya and set it up with the proper materials and lights that are compatible with Renderman. Settings in the standard Render Settings tab are honored by the script, so once that is set up the user runs Run_NN from the script editor. During testing the standard way to run the program was the classic maya three line comb: "import Run_NN, reload(Run_NN), Run_NN.showUI()." This script will start a batch render from the command line. The user can keep track of the progress of this render via the terminal they opened maya from. As an alternative, the user could choose to skip the batch render if they already have noisy images ready to be denoised. This use case is especially important for larger scenes where the artist may want to take advantage of BU's renderfarm. Initially some efforts were made to wrap this function up in automation, however, there were problems because using the render farm requires the user to enter their password at multiple points. It is also safe to assume that the artists are already very experienced with the render farm and doing it for them via a script would only have marginal benefits. Once the render is done, the script starts a simple Nuke comp from the command line. This is necessary because there was not a way to have maya render all passes in one EXR while having the variances passes. Near the end of the project it was discovered that during testing variance passes could be substituted with an all black frame, and the difference in output was impercetable. However, these passes were still necessary in training, and the nuke comp also served to rename the passes to something the network expected, so this step of the process was kept. The multi-channel EXRs are then moved to the appropriate directories for KPCN to run. The progress for the denoising process can also be seen in the terminal from which maya was opened. Once the denoising is done the files are moved to an output folder for ease of access by the user.

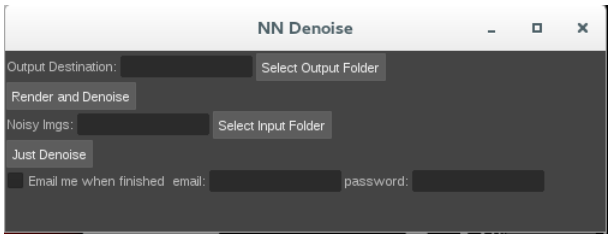

Figure 3: Simple GUI for the denoiser

The key functions of this process are as follows:

ShowUI: A simple GUI is created for the user to interact with. The primary components of this will drive the rest of the functionality of the script. The user may select a specific directory to output their denoised images. The interface uses the standard Maya file browser, fileDialog2. There is also a field for selecting a directory that has noisy images ready to be denoised. It is important to note that the dialog does not allow them to directly select a folder on the renderfarm, this limitation is related to previously mentioned issue with passwords, so if they have gone that route they must move those files first. Finally, the artist can choose to enter their information for an email account to give them a notification when the render and denoising has been completed (see Fig. 3).

FindPathNames: Throughout the program finding certain names and paths are important, thus a function is run at the beginning to collect and parse this important information. The information collected includes: scene name with and without the file extension, the path to the scene, the path to the rendered images, the user's home directory/student number, and the frame range. This information is added to a dictionary for the rest of the script to utilize.

BrowseForOutput/Input: These functions simply allow access to fileDialog2.

SendEmail: The user must specify their username and password in the previous GUI, these are used to then send a simple email to themselves that acts as a notification for when the program is done. The email and email.mime packages are used to compose the body of the email and the subject line.

MoveFiles: File management and manipulation is key to much of this program succeeding. Because of the work put into this aspect of the program the artist does not need to worry about where they are outputting their images, as it will find them and move them to the appropriate places. The function removes any previous noisy images from NN_Denoise directory, but does not remove anything outside that directory. Once the new images are in place, regular expressions are generated to find the specific passes. These are then sorted into the appropriate folders.

NukeComp: In order for the Nuke comp to run correctly from the command line, two python files are moved into the users .nuke folder. These allow Nuke to automatically look for all exrs in a certain directory, in this case each read node looks for a different pass in a directory of EXRs. The .nk file is then ran, and the two python files are moved out of the users .nuke to avoid clutter.

Denoise: This function simply runs the actual python script that is the denoiser, and then moves the denoised files to the appropriate directory.

NN_Render_Denoise: This function represents the primary use case of the project. The batch render is called via the subprocess module. It was found in testing that this only worked with the "shell=True," which can be a security risk since it allows for injection of potentially malicious shell code. Given that BU student projects are fairly safe in terms of being attacked by dangerous code, and the fact that the batch render command only pulls in outside variables for the scenename; the decision was made that this was a safe piece of code. The function then moves files, runs the nuke comp, denoises, and finally sends an email if necessary.

NN_Just_Denoise: This function is nearly identical to previous function expect instead of running a batch render, it pulls files from the user specified path.

0.4.3 Additional Files

BuildEXR.nk: While Maya does support outputting multiple passes into one multi-channel EXR automatically, at the time of this project's completion no solution to add in the necessary variance passes was found. The denoiser also looks for the variance channels to be in a one dimesional channel rather than RGB, which is not automatically supported by Renderman for Maya. The solution to this problem was a simple Nuke comp that combined the passes into one exr. This process should be relatively seamless in the whole scope of the project, as it only takes a few seconds per frame.

NN_Helper.sh: The necessity of script is unfortunately a function of unknown technical problems. When adding a certain line to the user's pythonpath in their bashrc, it caused a complete crash for maya. This shell script exists to do that export statement and then call the denoiser using mayapy, as that remedied the problem. Multiple forum posts and bug reports to Autodesk were not able to reveal the cause of the issue.

Uninstall.sh: This denoiser may not be a permanent addition to the user's workflow, so a simple uninstall script allows them to quickly remove the main NN_Denoise directory and uninstall tensorflow. The script also moves the Output directory to their desktop in case they forgot to grab their last set of images.

0.4.4 Training

The implementation of KPCN that is being run was created for Tungsten rather than Renderman, thus it needed to be retrained. Retraining denoising neural networks is traditionally difficult and time-consuming because of the need to create noisy and clean pairs of images. Totally clean images are obviously difficult to produce (or the denoiser wouldn't be necessary), so the training set was kept very simple. Scenes for training were provided by Bournemouth University's Aleks Czarnojan. These two scenes create a set of 400 frames, with some variety of movement and light. While the frame count is similar to the paper's training set (they used 600), the complexity of the set is what holds it back. The 600 from the paper were pulled from the extremely diverse pool of images in the film "Finding Dory." The authors suggest using frames that include many kinds of textures, in-camera effects, and lighting for best results, however, that simply isn't feasible given the timeframe of the project.

All images, training and testing were rendered at 1280x720 so they could be HD, while still being as quick

as possible. The training scenes were then rendered at 16 samples for the noisy images, and 1024 samples for the clean images. Because of the simplicity of the scenes the noisy side took seconds per frame and the the clean ones only took a couple of minutes per frame. The noise was not meant to be extreme, but rather to represent a more realistic and subtle degradation of image quality (see Fig. 4)

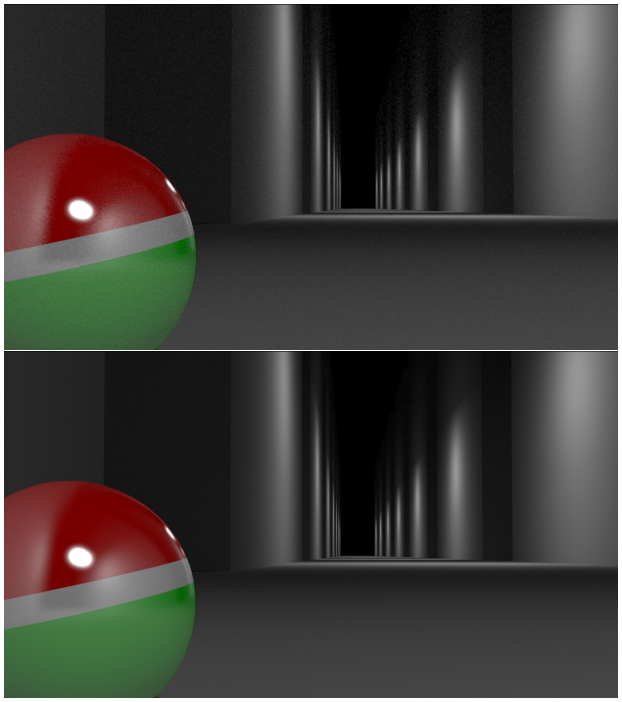

Figure 4: Clean and noisy image in a test scene

Another difficulty with denoising neural networks is the time it takes to train. The use of TensorFlow's GPU support is thus essential to train the network, as the speedup is several orders of magnitude. However, a limitation of BU's pipeline is that artists do not have admin access to their own machines, something which is necessary for the lengthy installation of TensorFlow with GPU support. The first option explored was VirtualEnv, but it did not allow for the changes that needed to be made in important directories outside of the viritual environment. Docker was briefly considered, but issues with filling up the fairly small local drive became prohibitive. Using a virtual machine on a laptop not owned by the University was investigated, but the extra steps to get a GPU working with this setup made it quite slow to get off the ground. The laptop that was available also had a signifigantly slower GPU than the one used in the paper. In the end, the project's solution was to train the network once on a specific computer used primarily for test builds, which, with the help of Jon Macey, had the correct components for TensorFlow GPU.

0.5 Results

The denoiser and its implementation proved qutie easy to use, it could be downloaded and used for the first time on a new system within five minutes, provided the user already a renderman ready scene they wanted to work with. These test iterations were conducted on another collegue's computer running RHEL. The user was able to create a new project, output their images somewhere specific, and in general work in maya as if they were not constrained by any specific post process requirements. The main expection to this was setting up the specific renderpasses and susequently using them in comps. The denoiser only outputs a denoised beauty, diffuse, and specular pass. However, utilizing maya's render layers could allow a student to get more out of a denoised scene. Because of the inability to use Tensorflow GPU on the artist's machine, it took ˜75 seconds per 720p frame to denoise, compared to around ˜15 seconds on the training machine. Student's shown the denoiser also found the novelty of the email functionality to be appealing. A small quirk of using someone else's implementation was that all denoised images come out with the red and blue channels swapped. An investigation into the code didn't reveal any obvious problems, so the issue was ignored, and the currect solution is simply to add a shuffle node in nuke when using the denoised images.

The ability of the system to denoise is still quite lacking and perhaps not even ready for simple use cases at BU. While the paper shows very clearly that the denoiser is capable of producing production quality images that match a ground truth clean image at nearly a per-pixel level, the model in use now needs more training. While the implementation downloaded from the paper's author's had trained for 360,000 iterations over a diverse training set, the current model only reached 50,000 iterations with its limited training set in the time provided. However, in complicated images like this, the network was showing no sign of improvement after the 20,000 or so iteration mark. Thus future time devoted to training would be best spent on either collecting or creating a more diverse set of images, rather than simply letting the network train longer. There may be some evidence of overfitting as well, the shadow details in the car's bumper in figure 5 appearing to be darkening. This could be from the dark training scene that represented half of the training set (see Fig. 4).

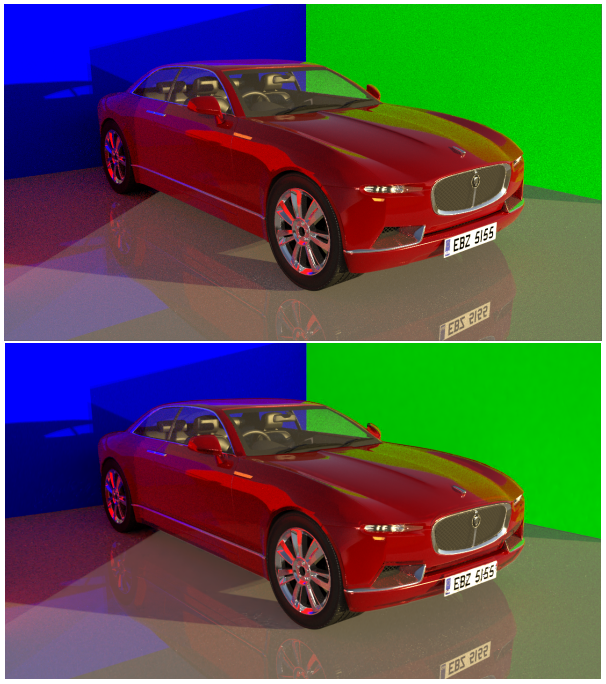

Figure 5: Complex scene denoised after 42,000 iterations. Noisy image rendered with 16 spp

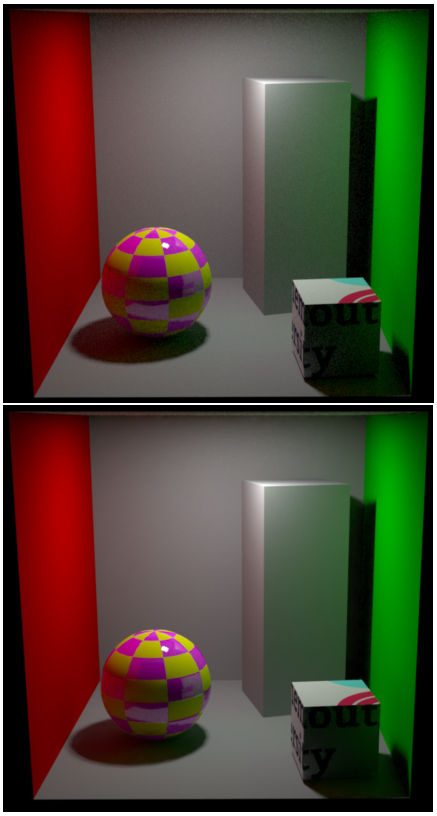

Figure 6: Simple test image

When tested with much simpler scenes and smaller levels of noise, the system did preform much better. It can be said that the more of a scene that resides in its diffuse component, the better this model will perform. The smaller amount of reflections and indirect light in the simpler scene also play a large role in how effective the denoising is (see Fig 6). Some scenes did require a bit more than previously stated list of passes to work properly. For example, the car in figure 5 required the transmissive pass to be combined with the diffuse pass in order for the denoised image to have anything behind the glass. This is because the standard diffuse pass in renderman was not picking up the objects enclosed in the car, presumably because they were surround by the highly specular glass.

0.6 Future Work and Conclusion

While this project did succeed in creating useful tools and an easy workflow, the work produced leaves something to be desire; all this to say the denoiser doesn't denoise very well. Fortunately, the implementation and it's integration is sound, so all that is required to improve the performance of the system is more training data and more time to tune the training. The group projects completed every year at BU present a good opportunity to get a large variety of scenes for training.

The installation of TensorFlow and its GPU support on the next year's lab build would also make training much easier as it could be done on any machine. Going along with this, other students discussed a desire to use the denoiser with other renderes, especially Vray. Having a drop down menu in the GUI that let the user select a different renderer would be trivial. The difficult part of this change would be the required training for each individual renderer, and indeed this can be cited as a weakness of denoisers that utilize AOVs in general.

While almost all of the project is automated the user must create the render pass setup. While training explicitly requires the variance passes, which are difficult to add in, testing can be done with standard passes only. The project isn't made to allow individual users to retrain the network, but it would be nice to both allow for that possibility and to remove one thing

the artist needs to remember. The best that can be done at the moment is a section in the README that gives a step by step guide.

One of the key weaknesses of KPCN is that it does not take advantage of the tendency of users to need to denoise multiple frames in a sequence. Almost as this paper was being written, Disney released the sequel to KPCN that utilizes data across a frame range to help denoising [9]. This strategy is also used on the training side of the system to actually remove the need for a ground truth image for every single input of the training. Integrating this into the current implementation of KPCN would certainly be time consuming, but would also give a large boost to the usability of the network. The end result is not completely useless though, as early projects that involve a simple bouncing ball animation in Maya could be aided by this. The UI makes this denoiser simple enough to be used by even the most novice of Maya users, especially artists who may not be comfortable using the renderfarm yet, and are looking for other ways to speed up iterations for their project.

0.7 Appendix

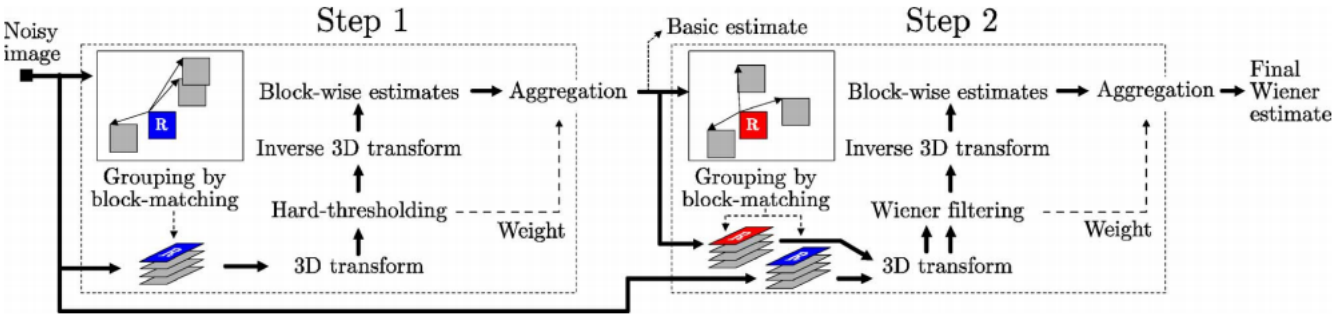

Figure 7: Flow Chart for BM3D

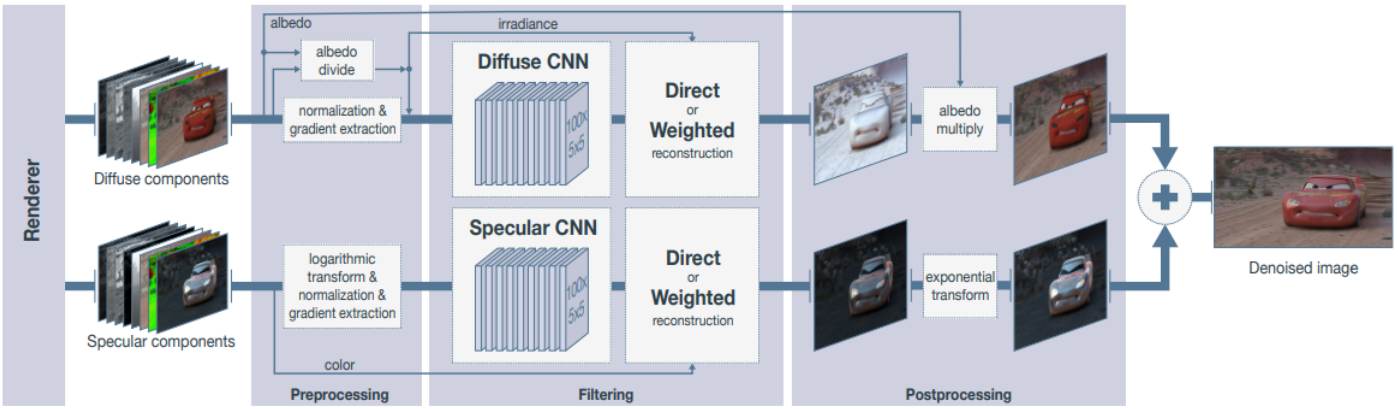

Figure 8: Flow Chart for KPCN

Bibliography

[1] Bako, S., Vogels, T., Mcwilliams, B., Meyer, M., NováK, J., Harvill, A., Sen, P., Derose, T. and Rousselle, F. (2017). *Kernel-predicting convolutional networks for denoising Monte Carlo renderings*. ACM Transactions on Graphics, 36(4), pp.1-14.

[2] Dabov, K., Foi, A., Katkovnik, V. and Egiazarian, K. (2007). *Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering*. IEEE Transactions on Image Processing, 16(8), pp.2080-2095.

[3] Hardy, Q. (2018). *Reasons to Believe the A.I. Boom Is Real*. [online] Nytimes.com. Available at: https://www.nytimes.com/2016/07/19/technology/reasons-to-believe-the-ai-boom-is-real.html [Accessed 13 Aug. 2018].

[4] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11), pp.2278-2324.

[5] Lehtinen, J., Munkberg, J., Hasselgren, J., Laine, S., Karras, T., Aitttala, M. and Aila, T. (2018). *Noise2Noise: Learning Image Restoration without Clean Data*. International Conference on Machine Learning. [online] Available at: http://research.nvidia.com/publication/2018-07_Noise2Noise%3A-Learning-Image [Accessed 13 Aug. 2018].

[6] Martin, X. (2015). *Cleaning Noise on Video*. [Blog] Xavier Martin VFX. Available at: http://www.xaviermartinvfx.com/x_denoise/ [Accessed 13 Aug. 2018].

[7] Infomatics Homepages Server. (2018). *Bilateral Filtering for Gray and Color Images*. [online] Available at: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html [Accessed 13 Aug. 2018].

[8] Vogels, T., Rousselle, F., Mcwilliams, B., Röthlin, G., Harvill, A., Adler, D., Meyer, M. and Novák, J. (2018). *Denoising with kernel prediction and asymmetric loss functions*. ACM Transactions on Graphics, 37(4), pp.1-15.

[9] Zhang, K., Zuo, W., Chen, Y., Meng, D. and Zhang, L. (2017). *Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising*. IEEE Transactions on Image Processing, 26(7), pp.3142-3155.