Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Eilif Tandberg Swensen

Deep Learning Controlled Temporal Upsampling

An Efficient Recurrent Convolutional Neural Network Controlled Architecture for Temporal Upsampling

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
June 2021

NTNU
Norwegian University of
Science and Technology

Eilif Tandberg Swensen

Deep Learning Controlled Temporal Upsampling

An Efficient Recurrent Convolutional Neural Network Controlled Architecture for Temporal Upsampling

Master’s thesis in Computer Science
Supervisor: Theoharis Theoharis
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Real-time rendering is becoming increasingly expensive due to higher resolution displays, higher refresh-rates, and photo-realistic graphics. The rendering cost can be reduced by rendering at lower resolution than the display, followed by upsampling to the display resolution. This thesis introduces a novel architecture for deep learning temporal upsampling, referred to as Deep Learning Controlled Temporal Upsampling (DLCTUS). In contrast to previous work, which focuses on constructing the upsampled frame entirely through the use of neural networks, DLCTUS uses a hybrid approach where a recurrent neural network controls the history rectification and sample accumulation stage of temporal upsampling. The thesis then shows that this simplification allows the architecture to produce images with a high image quality, while using a small and fast neural network. The image quality is shown to be slightly worse than state-of-the-art for 2x2-upsampling, but significantly better than state-of-the-art for 4x4-upsampling. In addition, a spatio-temporal loss function is formulated. The loss function increases the temporal stability of the architecture, but also slightly reduce the overall image quality. Finally, an optimized implementation of DLCTUS is created, and the run-time is shown to be an order of magnitude faster than state-of-the-art.

Sammendrag

Rendering i sanntid blir stadig dyrere på grunn av skjermer med høyere oppløsning, høyere bildefrekvens og fotorealistisk grafikk. Kostnaden av rendering kan reduseres ved å rendere ved en lavere oppløsning enn skjermen, etterfulgt av oppsampling til skjermoppløsningen. Denne oppgaven introduserer en ny arkitektur for dyp læring tidsoppsampling, referert til som Deep Learning Controlled Temporal Upsampling (DLCTUS). I motsetning til tidligere arbeid, som fokuserer på å konstruere det oppsamplede bildet helt gjennom bruk av nevrale nettverk, bruker DLCTUS en hybrid tilnærming der et rekurrent nevralt nettverk styrer historie korrigering og akkumulerings stadiene for tidsoppsampling. Oppgaven viser så at denne forenklingen gjør at arkitekturen kan produsere bilder med høy bildekvalitet, mens den benytter et lite og raskt nevralt nettverk. Bildekvaliteten blir vist til å være litt dårligere enn state-of-the-art for 2x2-oppsampling, men betydelig bedre enn state-of-the-art for 4x4-oppsampling. I tillegg formuleres en spatio-temporal tapsfunksjon. Tapsfunksjonen øker stabiliteten over tid i arkitekturen, men reduserer også bildekvaliteten litt. Til slutt blir en optimalisert implementasjon av DLCTUS laget, og kjøretiden blir vist til å være en størrelsesorden raskere enn state-of-the-art.

Contents

List of Figures

List of Tables

Acronyms

AABB Axis-Aligned Bounding Box. 16, 19, 52

Adam Adaptive Moment Estimation. 7

BPTT Back-Propagation-Through-Time. 7, 8, 28–30, 59

CNN Convolutional Neural Network. v, 3, 5, 6, 9, 19–25, 51, 52

DLCTUS Deep Learning Controlled Temporal Upsampling. i, v–vii, 1, 21, 22, 25, 27, 30, 32–58

DLSS Deep Learning Super Sampling. 1, 20, 58

EDSR Enhanced Deep Super-Resolution network. 9, 23, 24

ESPCN Efficient Sub-Pixel Convolutional Neural Network. 6, 9

FEQE Fast and Efficient Quality Enhancement. 9, 23

GPU Graphics Processing Unit. 1, 2, 20, 28, 30–33, 43, 57, 58, 60, 61

HDF5 Hierarchical Data Format 5. 27, 33, 59

HR High Resolution. v, 8, 9, 19, 22, 24, 26, 27, 54

JAU Jitter-Aligned Upsampling. vi, 22, 31, 34, 38

L1 Mean Absolute Error. vii, 9, 10, 28, 29, 44, 46, 55, 56

LR Low Resolution. v, 8, 9, 19, 22, 27

MSE Mean Square Error. 9, 10, 29

MSSIM Mean Structural Similarity Index Measure. 11, 30

PNG Portable Network Graphics. 27

PSNR Peak Signal-to-Noise Ratio. v–vii, 10, 23, 30, 34–38, 44, 45, 47, 48, 51, 54, 56

ReLU Rectified Linear Unit. 6, 24, 43

RGB Red, Green and Blue. 16, 21, 22, 25, 32

RGB-D Red, Green, Blue and Depth. 22, 31

RNN Recurrent Neural Network. 7, 19, 59

SISR Single Image Super Resolution. 1, 5, 6, 8, 9

spp Samples per Pixel. 13, 27

SRCNN Super-Resolution Convolutional Neural Network. 9

SSIM Structural Similarity Index Measure. vi, vii, 10, 11, 30, 34–38, 44, 45, 47, 48, 50–52, 55, 56, 59

TAA Temporal Anti-Aliasing. 13, 16, 17, 19, 20, 25, 26

TBPTT Truncated Back-Propagation-Through-Time. 8, 61

TUS Temporal Upsampling. v, vi, 1–3, 5, 13, 17, 22, 25, 34, 38, 40–42, 52–54, 59

WRAE Warped Recurrent Auto-Encoder. 19

1 Introduction

The computational power needed to preform real-time rendering has increased with the high demand for photo-realistic graphics. New PC-monitors, mobile-devices and AR/VR-headsets require higher resolutions and higher refresh rates which increases the amount of pixels that have to be rendered every second. At the same time expensive rendering techniques such as ray tracing are becoming more popular, which increases the computational load of each pixel. One way to improve performance is to render at a lower resolution, and then upsample the image to the output resolution. This approach improves performance provided that the time spent on upsampling is smaller than the time saved by reducing the amount of pixels rendered. However, it can also drastically reduce the image quality depending on the upsampling technique used. An important factor for upsampling is the ratio between the high resolution image and the rendered image, called the upsampling factor. The upsampling factor can be used to trade performance for visual quality as a higher upsampling factor will require fewer pixels to be rendered, but more pixels have to be reconstructed by the upsampling technique. One such upsampling technique is Temporal Upsampling (TUS) [1]. TUS utilizes information from previous frames to reconstruct a high resolution upsampled image. This is done using motion vectors, which describe the motion of pixels in between frames. While TUS has been successfully used in engines such as Unreal Engine 4 [2], it still struggles with artifacts such as ghosting, flickering and blurring. The artifacts become even more prominent when TUS is used with large upsampling factors.

A similar task to TUS is Single Image Super Resolution (SISR), which upsamples images using only one low resolution image as input. Recent progress in SISR uses deep neural networks to achieve state-of-the-art results [3], raising the question of whether deep neural networks can be used to improve the performance of TUS. TUS has a stricter time requirement than SISR, because it has to finish in between two frames of a renderer, while SISR has no such requirement. This limits the possibilities of deep neural networks for TUS, as the network operations used are time consuming. The recent introduction of tensor cores to Graphics Processing Units (GPUs) [4] has been a big step in reducing this time, as they allow the networks to efficiently utilize lower precision memory formats. Nvidia successfully applied deep neural networks to TUS with their Deep Learning Super Sampling (DLSS) architecture [5], and further improved visual quality and performance in the second iteration, DLSS 2.0 [6]. However, details about the algorithm and training are proprietary, leaving little reliable information to be used for research. Another approach by Xiao et. al. uses an architecture for deep learning TUS which achieves great visual quality, but is too slow to be applied in a real-time renderer.

The goal of this thesis is to investigate how deep learning can be applied to TUS in a way that is fast enough to be used in conjunction with a real-time renderer. Inspired by state-of-the-art methods for TUS, deep learning TUS, and SISR, the thesis introduces a novel architecture for deep learning temporal upsampling, referred to as Deep Learning Controlled Temporal Upsampling (DLCTUS). A simplified illustration showing the differences between TUS and DLCTUS is shown in Figure 1. DLCTUS merges the rectification and accumulation stages of TUS into one stage.

This stage uses the output of a neural network to decide how to rectify and accumulate samples. The architecture uses a recurrent neural network to accumulate an unrestricted number of past frames without increasing the network size. Additionally, it reduces the complexity of the task performed by the neural network compared to other methods, allowing the neural network to be smaller and faster. To illustrate how the architecture can be integrated into a renderer, a DirectX 12 implementation is created using DirectML to execute the neural network. The thesis also shows how to generate training data and train the neural network efficiently, and explores how a spatio-temporal loss function influences the results. The technical contributions of this thesis can be summarized as follows:

- It introduces a novel architecture for deep learning TUS.
- It shows how an accumulation buffer can be used to improve the visual quality of the architecture.
- For 4x4-upsampling the architecture outperforms state-of-the-art methods for deep learning TUS in quantitative metrics for image quality, while being significantly faster. For 2x2-upsampling, the architecture has a slightly worse visual quality than state-of-the-art.
- It introduces a parameterized linear spatio-temporal loss function that can be use to trade off temporally unstable errors for temporally stable errors, without a large impact on the overall visual quality.

1.1 Research Questions

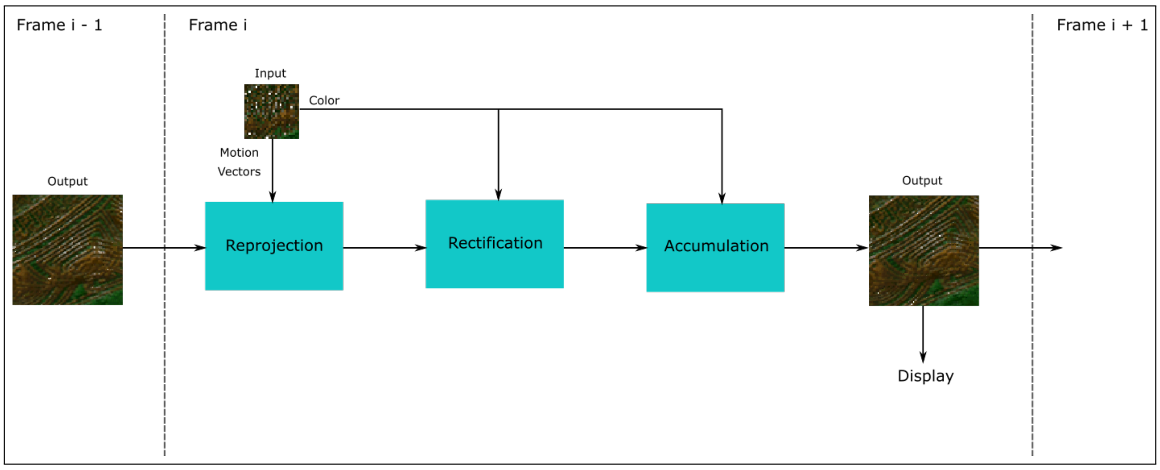
A few research questions were created to guide the thesis:

- **RQ1**: What are the main challenges of training and applying a recurrent neural network for TUS?
- **RQ2**: How can methods originally used to enhance TUS fit in a neural network approach, and how do they affect the visual quality?
- **RQ3**: How can a spatio-temporal loss function be formulated, and how does it impact the visual quality and temporal stability of the network?
- **RQ4**: What are the difficulties and limitations behind creating a neural network that runs in real-time on a modern GPU?

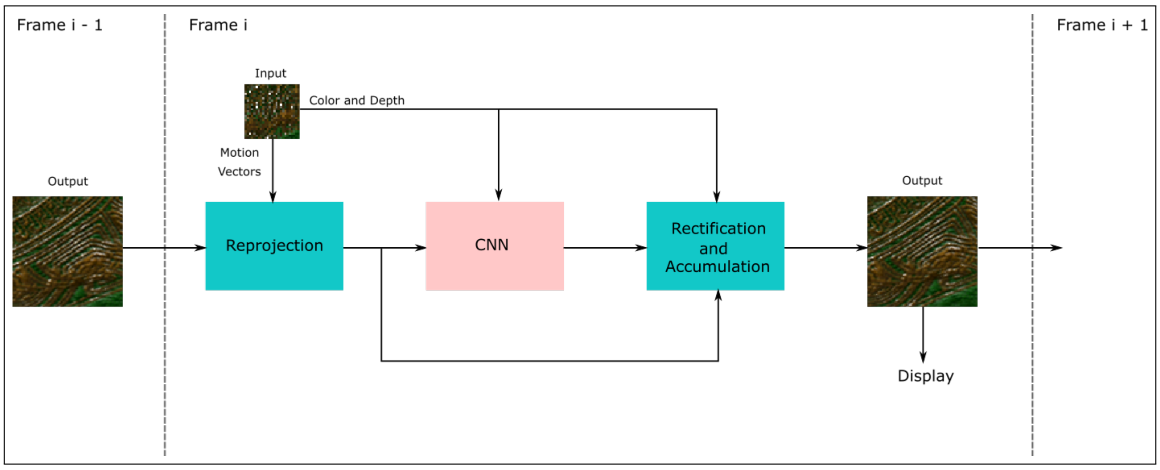
1.2 Structure

This section contains an overview of the structure of the thesis.

Section 2 - Background covers the background material necessary for understanding the work performed in this thesis.



(a) TUS



(b) DLCTUS

Figure 1: Simplified illustration of the differences between TUS and DLCTUS. TUS performs rectification and accumulation in two separate stages using algorithms based on heuristics. DLCTUS performs rectification and accumulation in the same stage. This stage is controlled by a Convolutional Neural Network. History reprojection, history rectification and sample accumulation are explained in detail in 2.4.

Section 3 - Previous Work summarises previous work done on the topic of applying deep learning to temporal upsampling.

Section 4 - Deep Learning Controlled Temporal Upsampling contains all information related to the proposed architecture.

Section 5 - Results presents the results from testing the proposed architecture.

Section 6 - Discussion discusses the results from the testing.

Section 7 - Conclusion concludes the thesis and presents possibilities for further research.

2 Background

This section provides a short introduction to the background material which is the foundation of this thesis. It outlines the inner workings of Convolutional Neural Networks (CNNs), SISR, anti-aliasing, and TUS.

2.1 Convolutional Neural Networks

A CNN is a neural network that employs one or more convolutional layers. The convolutional layers are used in combination with other layers such as activation functions and upsampling layers, where the layers are usually executed sequentially on the input of the network. The network represents a function $f_{CNN}(x)$ which maps an input tensor to an output tensor. CNNs are commonly used for tasks which involve an image as input, where the convolutional layers are used to extract features from the image. Convolutional neural networks were first used by Kunihiko Fukushima in the Neocognition [7], and was later popularized by LeNet-5 which successfully used convolutional layers to improve the ability of a neural network to classify images [8].

2.1.1 Convolutional Layer

A convolutional layer consist of an input tensor, output tensor, a filter kernel and a bias tensor. The input tensor has dimensions (C_i, H_i, W_i) which are respectively the channels, height and width of the input tensor. The output tensor has dimensions (C_o, H_o, W_o) which are respectively the channels, height and width of the output tensor. The filter kernel is also a tensor with dimensions (C_o, C_i, H_k, W_k) where H_k and W_k are the height and width of the kernel, and the bias tensor has one dimension with a size equal to C_o. The output tensor is created by performing a convolution or cross-correlation operation on the input tensor using the filter kernel and then adding the bias tensor. There are also other hyperparameters that control the output of a convolutional layer such as stride and padding. The stride is used to control how far the filter is shifted each step during the convolution. With a stride equal to 1 it is shifted one unit at the time, with a stride of 2 it is shifted 2 units at the time etc. Increasing the stride will reduce the width and height of the output tensor, which is why a stride larger than one is often used for downsampling. Another hyperparameter is padding. Padding is used to increase the width and height of the input tensor before the convolution is applied. This is commonly used to ensure that the spatial resolution of the input tensor matches the spatial input of the output tensor, which prevents information from being lost due to reduction in tensor sizes. Two commonly used padding types are zero-padding and replication-padding. Zero-padding increases the size of the input by inserting zeros at the edges, while replication padding inserts the color at the closest border.

2.1.2 Pixel Shuffle Layer

A pixel shuffle layer is an upsampling layer which increases the spatial resolution of the input tensor by redistributing the input tensor's channels in the spatial dimensions. It was first used in the Efficient Sub-Pixel Convolutional Neural Network (ESPCN) [9] architecture for SISR which increased the efficiency by performing most of the convolutions in low resolution before performing upsampling using a pixel shuffle layer. Pixel shuffling with an upsample factor of r takes an input tensor of dimensions (C, H, W) and rearranges the tensor into an output tensor with dimensions $(C/r^2, Hr, Wr)$.

2.1.3 Activation Functions

Activation functions can be applied to the output of a network layer with the purpose of introducing non-linearity to the network. This non-linearity makes it possible for the network to fit non-linear functions. A popular activation function is the Rectified Linear Unit (ReLU):

$$ReLU(x) = \max(0, x) \tag{1}$$

ReLU is popular due to its computational efficiency and its resilience against vanishing gradients [10].

2.1.4 Residual Blocks

The ResNet architecture was introduced by He et al. [11] to improve the training of deep neural networks. The architecture uses shortcut connections to better propagate the gradients through the network while training. The shortcut connections are implemented by adding the output of one layer to the output of another layer later in the network. Deep networks are constructed by defining a "residual block", which consists of a sequence of layers, where the start and end point are connected with a shortcut connection. Then the residual blocks are stacked after each other until the desired depth is reached. The composition of the residual block varies, but it commonly includes two convolutional layers.

2.1.5 Supervised Learning

A CNN can learn by changing the values of the weights in the filter and bias kernels. The goal is to adjust the weights of the network until f_{CNN} approximates a function f_{target}. Supervised learning uses labeled training data paired as x_i, an input to the function, and $f_{target}(x_i)$, the target output. A loss function is utilized when using supervised learning for CNN. The loss function, $\mathcal{L}$, is used to measure the similarity between the output of the network and the target output. The similarity is given as a number where a smaller value equals greater similarity. The goal of supervised learning is to minimize the loss function over the training data. By making sure that the CNN is fully differentiable it is possible to calculate the derivative to the loss function with respect to a certain weight $\frac{\partial}{\partial w} L(f_{CNN}(x_i), f_{target}(x_i))$ called the gradient. The gradient is then used as part of an optimization algorithm that attempts to reduce the average loss over the training data by changing the network weights. It is common to use the average of the loss of multiple input values to calculate the gradient, this is called batching and the number of input values used is called the batch size.

2.1.6 The Adam Optimizer

Adaptive Moment Estimation (Adam) [12] is an algorithm for updating a parameter θ_t at timestep t given a computed gradient g_t. It keeps exponential moving averages of the mean of the gradients, m_t, and the uncentered variance of the gradients, v_t, using the following equations:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3}$$

where β_1 and β_2 are empirically chosen constants. The moving averages are biased towards 0, which is why Adam computes the bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{5}$$

The parameter θ is then updated using;

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{6}$$

where ϵ is a constant used to stabilize the equation and η is the learning rate. Kingma and Ba [12] found good default values for the constants to be $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

2.1.7 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of neural network which has memory. The memory is called the network's "hidden state" and can be used during the calculation of the output. This makes RNNs ideal for problems that involve time series, where the output of one iteration might depend on the output of the previous iteration. RNNs can be trained using Back-Propagation-Through-Time (BPTT), where the network is trained on a sequence of input and output pairs [13]. BPTT starts with setting the hidden state to an initial state. Then the loss for each iteration is calculated using the state from the previous iteration, and input and output from the current iteration. The average loss over all iterations are then calculated, and this average loss is back-propagated, where the gradients are propagated not just through the network, but backwards through all iterations of the network. Another training method called Truncated Back-Propagation-Through-Time (TBPTT) operates on the same principles as BPTT, but in addition uses two parameters $k1$ and $k2$. TBPTT performs multiple back-propagations for each sequence, where $k1$ decides how often back-propagations are performed, and $k2$ decides how many iterations back the back-propagation is propagated [13].

2.2 Single Image Super Resolution

SISR is the task of creating a High Resolution (HR) image from a Low Resolution (LR) image. This task is challenging because the LR image does not contain all the information necessary to perfectly recreate the corresponding HR image. The difficulty of the SISR task depends upon the upsampling factor. The upsampling factor is the ratio between the dimensions of the HR-image and LR-image. An upsampling factor of 2 would increase the width and height by 2 and one pixel in the LR-image would correspond to four pixels in the HR-image. The theory in this subsection is from Wang et al. [14], unless explicitly stated otherwise.

2.2.1 Interpolation-based Methods

Traditional methods for SISR relies on spatial coherence. The idea is that the missing pixels in the HR-image are similar to adjacent pixels in the LR-image. Interpolation is used to derive the color-values of HR-pixels in between LR-pixels. Three different interpolation schemes are commonly used; nearest-neighbor interpolation, bilinear interpolation, and bicubic interpolation. Nearest-neighbor interpolation uses the color of the closest LR-pixel as the color for a HR-pixel. This is computationally efficient, but leads to blocky and pixelated results. Bilinear interpolations uses the closest 2x2 LR-pixels to the HR-pixel and preforms first linear interpolation in one direction, followed by linear interpolation on the result in the other direction. This leads to a smoother result than nearest-neighbor upsampling, however it looses some sharpness as the linear interpolation works as a low-pass filter. Bicubic interpolation works the same way as bilinear, but it uses the 4x4 closest pixels and cubic interpolation instead of linear interpolation. This extra information makes bicubic interpolation better at upsampling of high frequency components, which results in a sharper and smoother image than bilinear upsampling, but at the cost of extra computations.

A common way to implement bicubic interpolation is by using the bicubic convolution algorithm introduced by Keys in [15]. This algorithm performs bicubic interpolation by applying a convolution with a filter $W_a(x)$. The filter is parameterized by a parameter a. Common values for a is -0.5 and -0.75, and Keys showed that the algorithm achieves third order convergence when $a = -0.5$.

2.2.2 Deep Learning Based Methods

Deep learning based methods have a different approach to SISR than the traditional methods. The traditional methods for SISR relied on the information of the LR image to reconstruct the HR-image, and does not introduce any new information in the process. Deep learning based methods on the other hand, use information learned from training data to "hallucinate" missing information into the HR-image.

The first successful attempt at using deep learning for SISR was Super-Resolution Convolutional Neural Network (SRCNN) [3]. SRCNN starts off with applying bicubic upsampling to the image, and then enhances the upsampled image using a series of convolutional layers. This works well since the network only has to enhance an already upsampled image, rather than learning a mapping from LR to HR. The downside to this approach is that all of the convolutions execute on HR data, which is computationally demanding.

Newer approaches attempt to reduce the computational cost, while increasing image quality. Three examples are Efficient Sub-Pixel Convolutional Neural Network (ESPCN) [9], Enhanced Deep Super-Resolution network (EDSR) [16], and Fast and Efficient Quality Enhancement (FEQE) [17]. ESPCN reduces the computational complexity and the final image quality by applying the convolutions to the LR-image before using a pixel-shuffle layer to increase the resolution. EDSR increases the performance by demonstrating that batch normalization is unnecessary in SISR networks. FEQE decreases the computations needed for SISR by performing downsampling at the start of the network, allowing most of the computations to be performed at a lower resolution. Both EDSR and FEQE uses residual connections to enable deeper networks.

Datasets for SISR are created by taking a set of HR-images denoted as $\hat{I}$ and downsampling them to create a set of LR-images denoted as I, where bicubic interpolation is commonly used for the downsampling process. This makes datasets for SISR easy to obtain, as only a set of images are needed.

2.2.3 Loss Functions

A common way to calculate the loss of a CNN is to calculate the pixel-wise Mean Absolute Error (L1) or the pixel-wise Mean Square Error (MSE):

$$\mathcal{L}_{L1}(\hat{I}, I) = \frac{1}{hwc} \sum_{i,j,k} |\hat{I}_{i,j,k} - I_{i,j,k}| \tag{7}$$

$$\mathcal{L}_{mse}(\hat{I}, I) = \frac{1}{hwc} \sum_{i,j,k} (\hat{I}_{i,j,k} - I_{i,j,k})^2 \tag{8}$$

where h, w and c are respectively the height, width and number of channels of the target image, $\hat{I}$ is the target image and I is the reconstructed image. MSE-loss has a larger penalty for larger errors, but it also has a large tolerance for small errors. L1-loss has shown to have better convergence properties and performance and is therefore often preferred over MSE-loss. Both L1-loss and MSE-loss does not take the perceptual quality of the image into account, i.e. the perceived similarity by a human observer. Johnson et al. [18] suggested to use the output of specific layers of a trained VGG-16 network as a perceptual loss function:

$$\mathcal{L}_{perceptual}(\hat{I}, I) = \frac{1}{hwc} \sum_{m \in V} \sum_{i,j,k} (\phi^m(\hat{I}_{i,j,k}) - \phi^m(I_{i,j,k}))^2 \tag{9}$$

where ϕ^m is the output of the m-th layer, and $V = \{2, 5, 9, 13\}$. This function is used because the output of intermediate layers in the VGG-16 network corresponds to specific features in the target image.

2.2.4 Evaluation Metrics

Two different metrics are often used to evaluate the image quality of an upsampling method: Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index Measure (SSIM). PSNR is calculated using the peak signal value L of the image, and the measured noise of a constructed image $\hat{I}$ relative to a target image I, given by $\mathcal{L}_{mse}(\hat{I}, I)$. PSNR is then defined as:

$$\text{PSNR}(\hat{I}, I) = 10 \log_{10}(\frac{L}{\mathcal{L}_{mse}(\hat{I}, I)}) \tag{10}$$

L is commonly equal to 255 due to the usage of 8-bit color-channel values.

PSNR is a logarithmic scale with units of decibel, and is criticised for not taking image structure into account. This is why SSIM has become more popular. SSIM is calculated using the mean, μ_I, standard deviation, σ_I, and correlation, $\sigma_{I\hat{I}}$, of the constructed image and the target image, defined as:

$$\mu_I = \frac{1}{N} \sum_{i=0}^{N} I(i) \tag{11}$$

$$\sigma_I^2 = \frac{1}{N-1} \sum_{i=0}^{N} (I(i) - \mu_I)^2 \tag{12}$$

$$\sigma_{I\hat{I}} = \frac{1}{N-1} \sum_{i=0}^{N} (I(i) - \mu_I)(\hat{I}(i) - \mu_{\hat{I}}) \tag{13}$$

They are then used to calculate three factors:

$$\mathcal{C}_l(I, \hat{I}) = \frac{2\mu_I \mu_{\hat{I}} + C_1}{\mu_I^2 + \mu_{\hat{I}}^2 + C_1} \tag{14}$$

$$\mathcal{C}_c(I, \hat{I}) = \frac{2\sigma_I \sigma_{\hat{I}} + C_2}{\sigma_I^2 + \sigma_{\hat{I}}^2 + C_2} \tag{15}$$

$$\mathcal{C}_s(I, \hat{I}) = \frac{\sigma_{I\hat{I}} + C_3}{\sigma_I \sigma_{\hat{I}} + C_3} \tag{16}$$

where C_1, C_2 and C_3 are empirical constants used for numerical stability. SSIM is then defined as:

$$\mathrm{SSIM}(I,\hat{I}) = |\mathcal{C}_l(I,\hat{I})|^{\alpha}|\mathcal{C}_c(I,\hat{I})|^{\beta}|\mathcal{C}_s(I,\hat{I})|^{\gamma} \tag{17}$$

where α, β and γ are constants used to give different importance to the different factors.

Including the correlation between the images make SSIM better at quantifying the structure of the images, which is a desired property. A variant of SSIM referred to as Mean Structural Similarity Index Measure (MSSIM) is often used as it is better at picking up the local structure instead of the global structure. MSSIM calculates a local SSIM for each pixel of the image, and uses the average for quality assessment. The local SSIM is calculated over a neighbourhood of the pixel, called the window size, and uses gaussian weights w_i with $\sum_i w_i = 1$ to calculate the mean, standard deviation, and correlation:

$$\mu_I = \sum_{i=0}^{N} w_i I(i) \tag{18}$$

$$\sigma_I^2 = \sum_{i=0}^{N} w_i (I(i) - \mu_I)^2 \tag{19}$$

$$\sigma_{I\hat{I}} = \sum_{i=0}^{N} w_i (I(i) - \mu_I)(\hat{I}(i) - \mu_{\hat{I}}) \tag{20}$$

where the sum is over the local window. The local SSIM is then calculated using Equation 17, and MSSIM is calculated using:

$$\mathrm{MSSIM}(I,\hat{I}) = \frac{1}{M}\sum_{j=0}^{M} \mathrm{SSIM}_j(I,\hat{I}) \tag{21}$$

where M is the window count and the sum is over all windows.

2.3 Anti-aliasing

Aliasing occurs when the rasterizer samples the screen-space shading function S using a grid pattern. Nyquist's sampling theorem states that any frequency of S larger than $\frac{1}{2}f_s$ will be reconstructed incorrectly, or aliased, as a lower frequency. For the grid used in rasterization is this frequency given by $f_s = \frac{1}{w_p}$ in the horizontal direction and $f_s = \frac{1}{h_p}$, where w_p is the width of a pixel, and h_p is the height of a pixel. Anti-aliasing can be done either by pre-filtering S by limiting its bandwidth, or with post-filtering by decreasing the distance between samples or applying anti-aliasing filters. This section is based on the background section of a fall project [19] by the author.

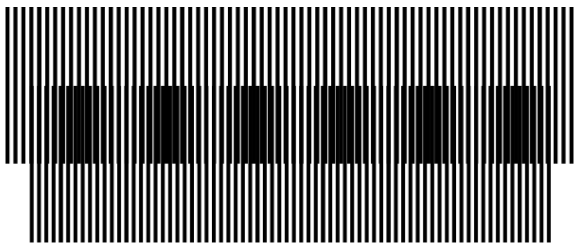


Figure 2: An example of a moiré pattern caused by two overlapping sequences of lines. The bottom sequence has a slightly shorter distance between the lines, making the lines of the bottom sequence iterate between falling on and off the the lines in the top sequence.

2.3.1 Moiré Pattern

A common artifact caused by aliasing is moiré patterns. Figure 2 illustrates a moiré pattern caused by two overlapping sequences of lines, with slightly different frequencies. Moiré patterns can be confusing to an observer, since the observed moiré pattern can be vastly different from the expected underlying signals. An additional problem with moiré patterns are their behavior under motion, since a small change to the underlying signal can cause a large change to the moiré pattern, which can be disturbing to an observer.

2.3.2 Mip-mapping

Mip-mapping was introduced by Lance Williams in [20] and is a pre-filtering anti-aliasing technique for textures. To avoid aliasing should every texel correspond to 2 pixel samples or more. With mip-mapping this is done by creating a pyramid of D levels, called mip-levels. The width and height of the texture at level i is half of the width and height of the texture at level $i-1$ and the texture at level 0 is the original texture. Each texture is created using a 2x2 box filter on the texture at the level above. The following equation us used to caluclate the mip-level d during sampling:

$$d = max\left[\sqrt{\left(\frac{\partial u'}{\partial x}\right)^2 + \left(\frac{\partial v'}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u'}{\partial y}\right)^2 + \left(\frac{\partial v'}{\partial y}\right)^2}\right] + b \tag{22}$$

where (u', v') is the texture position in texels, (x, y) is the pixel position in screen pixels and b is a manually selected bias used to give the programmer control over the mip-level selection and are in most cases left as 0. Trilinear interpolation is used when d is fractional to create a smooth transition between mip-levels.

2.4 Temporal Upsampling

TUS performs upsampling on an image sequence by reusing samples stored in previous images to upsample the current image. It is both an upsampling technique and an anti-aliasing technique, as the previous samples can be used to reduce aliasing in addition to upsampling. TUS is a natural expansion of Temporal Anti-Aliasing (TAA), where TAA only performs anti-aliasing and not upsampling. This section will first go into TAA, and then show how TAA can be expanded to TUS. The material in this section is based on the background section of a fall project [19] by the author.

2.4.1 Temporal anti-aliasing

TAA was first introduced by Yang et al. in [1]. In TAA, samples from previous frames are reprojected and reused in the current frame to increase Samples per Pixel (spp) and thereby reducing aliasing. It is common to use a history buffer to store the accumulated samples. This reduces complexity because only the history buffer has to be reprojected, and not several previous frames. Every frame of the current history buffer value $f_i(p)$ is updated according to the following equation:

$$f_i(p) = g(s_i(p), f_{i-1}(\pi_i(p))) \tag{23}$$

where g is an accumulation function, $\pi_i(p)$ returns the position of a pixel, p, in the previous frame, and $s_i(p)$ is the new sample.

TAA works in 4 stages: jittered rendering, history reprojection, history rejection or history rectification, and accumulation.

2.4.2 Jittering

To ensure high quality output the accumulated samples should be distributed evenly within a pixel [21], this makes it necessary to add a sub-pixel offset to the sample position of every frame. In addition, every sub-sequence of the jittering offsets should be evenly distributed. This is because a pixel can become disoccluded at any point in time, which makes any offset in the sequence a possible starting point. Low discrepancy sequences have this property and a popular choice is the Halton sequence. The Halton sequence uses two Van der Corput sequences with coprime bases for the x and y component of the jitter offset. The n-th element of a Van der Corput sequence of base b is defined using the base b representation of n:

$$n = \sum_{k=0}^{L-1} d_k(n, b) b^k \tag{24}$$

where L is the number of digits and $d_k(n, b)$ is the k-th digit of n in base b. The n-th Van der Corput element of base b, $h_b(n)$, is then defined as:

$$h_b(n) = \sum_{k=0}^{L-1} d_k(n) b^{-k-1} \tag{25}$$

The Halton(2,3) offset g_i of index i is then defined as in the following equation.

$$g_i = (h_2(i), h_3(i)) \tag{26}$$

This offset is applied to samples during rendering by adding the offset to the projection matrix, P_i:

$$P_{i,\text{jitter}} = P_i + \begin{bmatrix} 0 & 0 & \frac{2g_{i,x}-1}{w} & 0 \\ 0 & 0 & \frac{2g_{i,y}-1}{h} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{27}$$

where w and h is the window width and height respectively.

2.4.3 History reprojection

History reprojection involves finding a function $\pi(p)$ that maps the pixel p to its location in the previous frame. It is important that the reprojection does not involve any jittering, as this will introduce unnecessary blurring due to the bilinear filtering used for sampling. Let P_i denote the projection matrix for frame i without jitter, and V_i denote the view matrix for frame i. For static objects, π can be denoted as in the following equation:

$$\pi_{\text{static}}(p) = P_{i-1} V_{i-1} V_i^{-1} P_i^{-1} p \tag{28}$$

Dynamic objects requires additional information about the objects movement. This is done using motion vectors. Motion vectors contain the offset from a pixel in the current frame, to the pixels position in the previous frame. They are stored in buffer with the same dimension as the rendered color buffer. During rendering, the motion vectors are calculated in a vertex shader. The position of the vertex in the previous frame and current frame is calculated according to the following equation:

$$\begin{aligned} v_i^{ECS} &= P_i V_i W_i v^{MCS} \\ v_{i-1}^{ECS} &= P_{i-1} V_{i-1} W_{i-1} v^{MCS} \\ v_i^{CCS} &= \frac{v_i^{ECS}}{(v_i^{ECS})_w} \\ v_{i-1}^{CCS} &= \frac{v_{i-1}^{ECS}}{(v_{i-1}^{ECS})_w} \end{aligned} \tag{29}$$

where W_i is the vertex' world matrix at frame i, v^{MCS} is the vertex position in the model coordinate system, and v_i^{CCS} is the vertex' position in clip coordinate system at frame i.

The rasterizer then interpolates between the vertex motion vectors to get the pixel motion vectors which are then stored in the motion vector buffer, shown in the following equation:

$$mv(p) = R((v_{i-1}^{CCS} - v_i^{CCS})_{xy}, p) \tag{30}$$

where R is the rasterizer interpolation function and $mv(p)$ is the motion vector for the pixel p.

The reprojection $\pi_{\text{dynamic}}(p)$ is then calculated using the following equation:

$$\pi_{\text{dynamic}}(p) = p + mv(p) \tag{31}$$

Since $\pi(p)$ can be fractional interpolation can be used to get the reprojected pixel color. This interpolation introduces errors into the reprojected history. Reprojection error can accumulate over multiple frames due to constant motion and can be perceived as a bluring of the output. The extent of the error was calculated by Yang et al. [1] for a bilinear filter under constant velocity. Better interpolation techniques, such as bicubic interpolation, can be used to reduce the error.

Another problem that arises during reprojection comes from the aliased nature of the motion vectors. Aliased motion vectors can produce the wrong reprojection, especially on object boundaries where the motion vector can either reflect the motion of the foreground object or the background object. When choosing between background and foreground motion vectors, the foreground is preferred as the foreground attracts more attention from the observer. Motion vector dilation can be used to sample foreground motion vectors over background motion vectors. It uses a dilation window during motion vector sampling, and compares the depth of the pixels inside the window. Then the motion vector of the pixel closest to the camera is chosen.

2.4.4 History rejection/rectification

History reprojection is not perfect and the reprojected history color can sometimes be incorrect. This happens when a pixel has been occluded in the previous frames, or for effects that cannot be captured by motion vectors, such as transparency, shadows, and specular highlights. To solve this Nehab et al. [22] proposes history rejection which compares the depth at $\pi(p)$ in the previous frame with the depth at p in the current frame. Other approaches combines depth data with other geometry data such as normals and object ids. By using geometry data, incorrect history will be successfully identified in the case of pixel disocclusion, but will fail for other effects such as shadows and transparency. It also fails on geometry edges, where a pixel can contain both valid foreground and valid background samples, leading to aliased edges.

Due to problems related to history rejection other methods have been proposed that aim for rectification rather than rejection. These methods fall under history rectification, and attempt to utilize the spatial coherence of the input samples to correct incorrect history color. Using the assumption that the local 3x3 neighborhood of input samples is a good representation of valid color values for the history color, the history color is clipped to the convex hull of the 3x3 neighborhood of the corresponding sample also known as the samples color gamut. The clipping is done by calculating the intersection between the color gamut and a line from the history color to the average color of the 9 samples. An approximation is usually used since this convex hull intersection is computationally expensive. The two most common approximations are Axis-Aligned Bounding Box (AABB) clamping and AABB clipping, referred to as history clamping and history clipping respectively [21]. Both methods approximate the convex hull with an AABB, but history clamping further approximates with the introduction of clamping instead of clipping. Karis [23] proposes to use the YCoCg color space for the AABB, as the AABB might fit more tightly to the convex hull, the transformation between the two color spaces are shown in the following equations:

$$\begin{bmatrix} p_Y \\ p_{Co} \\ p_{Cg} \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & -2 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} p_r \\ p_g \\ p_b \end{bmatrix} \tag{32}$$

$$\begin{bmatrix} p_r \\ p_g \\ p_b \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} p_Y \\ p_{Co} \\ p_{Cg} \end{bmatrix} \tag{33}$$

where (p_Y, p_{Co}, p_{Cg}) are the color components of a pixel in YCoCg-space, and (p_r, p_g, p_b) are the color components in RGB-space.

2.4.5 Sample accumulation

The last step of TAA is to combine the jittered samples, $s_i(p)$, with the previous history buffer color, $f_{i-1}(\pi_i(p))$, using the following equation:

$$f_i(p) = \alpha s_i(p) + (1 - \alpha) f_{i-1}(\pi_i(p)) \tag{34}$$

where α is a blending factor.

TAA implementations use two different choices for α, either $\alpha = 1/N(p)$ where $N(p)$ is the number of accumulated samples, or $\alpha = \mathit{const}$. When $\alpha = 1/N(p)$, Equation 34 turns into an average over all samples. The benefits of this choice is that it has optimal convergence rate and optimal use of samples, since the effective number of samples equals the total number of samples. The downside is that it requires an accumulation buffer to store $N(p)$. It also weights every sample equally, which is not always good for TAA, since the older samples can include errors due to resampling and rectification. On the other hand, constant α does not require an additional storage channel, and assigns higher weighs to newer samples. It does however have a lower effective number of samples which can be shown using the following equation:

$$\frac{1}{N'_i} = \alpha^2 + (1-\alpha)^2 \frac{1}{N'_{i-1}} \tag{35}$$

which is Equation 30 from the appendix of Yang et al.. Here N'_i is the effective number of samples after i iterations of Equation 34 and $N'_1 = 1$. This equation converges when i tends towards infinity, as is shown in the two following equations:

$$\begin{aligned} \frac{1}{N'_{max}} &= \lim_{i\to\infty} \frac{1}{N'_i} \\ &= \lim_{i\to\infty} \alpha^2 + (1-\alpha)^2 \frac{1}{N'_{i-1}} \\ &= \alpha^2 + (1-\alpha)^2 \frac{1}{\lim_{i\to\infty} N'_{i-1}} \\ &= \alpha^2 + (1-\alpha)^2 \frac{1}{N'_{max}} \end{aligned} \tag{36}$$

$$N'_{max} = \frac{2-\alpha}{\alpha} \tag{37}$$

A smaller α leads to an increase in effective samples per pixel, but it also increases the accumulated reprojection error discussed in 2.4.3. Yang et al. [1] calculated a lower limit for α under motion when using bilinear filtering. A more common approach is to use a simple empirical approximation that limits α based on pixel velocity [23].

2.4.6 Upsampling

Temporal Upsampling differs from Temporal Anti-Aliasing in that there is no longer a one to one mapping between input samples and output pixels. This makes it necessary to scale the input samples up to the output resolution, which is done using the following equation:

$$\begin{aligned} \bar{s}_n(p) &= \frac{1}{w(p)} \sum_{i\in N(p)} \delta(o_i) s_i \\ w(p) &= \sum_{i\in N(p)} \delta(o_i) \end{aligned} \tag{38}$$

Here $\bar{s}_n(p)$ is the scaled input sample for output pixel p, $N(p)$ is a fixed neighborhood around p, s_i is the i-th sample in $N(p)$, o_i is the distance between p and s_i, $\delta(o_i)$ is a reconstruction filter kernel, and $w(p)$ is a normalization factor.

Directly using the rescaled samples in Equation 34 will result in blurred output. This is because some output pixels will use an average of multiple input samples that are located far away from the output pixel. An additional parameter β is introduced to recover the sharpness of the image, which is shown in the following equation:

$$f_i(p) = \alpha\beta(p)\bar{s}_i(p) + (1 - \alpha\beta(p))f_{i-1}(\pi_i(p)) \tag{39}$$

Yang et al. [1] use a bilinear tent as reconstruction filter, shown in the following equation:

$$\delta(d) = \text{clamp}(1 - \frac{|d_x|}{W_p}, 0, 1) \times \text{clamp}(1 - \frac{|d_y|}{H_p}, 0, 1) \tag{40}$$

To calculate β, Yang et al. uses a one pixel wide box. If distance between the sample and the pixel is less than the size of the pixel then $\beta = 0$, otherwise $\beta = 0$.

Herzog et al. and Unreal Engine 4 [24] [23] use a gaussian reconstruction kernel:

$$\delta(d) = e^{-\frac{d \cdot d}{2s^2}} \tag{41}$$

and use the largest non normalized filter weight as $\beta(p)$:

$$\beta(p) = \max_{i \in N(p)} \delta(o_i) \tag{42}$$

3 Previous Work

This section presents previous work done on the topic of applying deep learning for temporal upsampling.

3.1 Salvi

Salvi [25] uses deep learning to improve TAA quality by applying two different approaches. The first approach uses convolutional layers (2.1.1) to learn the optimal color extents used in AABB clamping (2.4.4). The second approach improves the quality even more by using a Warped Recurrent Auto-Encoder (WRAE) to combine the new samples with history data to create the output. The WRAE is a RNN (2.1.7) where the hidden state is warped. This warping is the same as history reprojection from TAA (2.4.3). The auto-encoder is a CNN (2.1) that progressively compresses the input using strided convolutional layers until a bottleneck is reached. Then the data is progressively decompressed using transposed convolutions until it reaches the target resolution. The WRAE produces images that are much closer to ground truth than TAA. However, while it performs better at history rectification in situations were TAA struggles, the WRAE struggles with ghosting in simple situations that TAA handles well. Salvi also proposed the use of a spatio-temporal loss function:

$$\mathcal{L}_{st} = ||T - P||_2 + ||\frac{\partial}{\partial t}T - \frac{\partial}{\partial t}P||_2 \tag{43}$$

where T is the target image and P is the reconstructed image. Including a temporal term in the loss function increases the temporal stability of the trained network.

3.2 Xiao et al.

Xiao et al. [26] propose an architecture that combines the color, depth and motion vectors of five subsequent frames into a final upsampled frame. This is done in three stages: a future extraction stage, a feature re-weighting stage, and a reconstruction stage.

Feature extraction is done by running the LR input color and depth from the last five frames separately through feature extraction networks. The architecture uses two feature extraction networks, each consisting of 3 convolutional layers (2.1.1). The first one is used for the current frame, and the second one is shared among the other four frames. The feature extraction networks create eight features for each frame, which are then concatenated with the frame's color and depth, resulting in twelve features for each frame. The twelve features are then upsampled to the target resolution using zero-upsampling, which increases the resolution by mapping the LR-pixel to the HR-pixel that corresponds with the pixels sample position, and leaves all other pixels as zero.

The four previous frames are then projected to the current frame using backwards warping. This is done by first upsampling the motion vectors using bilinear interpolation, and then using the upsampled motion vectors to resample each frame onto the current frame, using bilinear filtering when the motion vectors end up between pixels. This is similar to history reprojection from TAA (2.4.3).

The color and depth of current frame and four warped frames are then passed through a feature reweighting network. This network is a CNN (2.1), consisting of three layers, that produces a weight for each pixel of the four previous frames. The weights are scaled to lie between 0 and 10, and are multiplied with the features of their respective frame. The features of the current frame and the reweighted features of the previous frame are then concatenated and passed through a reconstruction network.

The reconstruction network uses a 3-level U-Net architecture and has a total of 10 convolutional layers. The reconstruction network outputs the final upsampled image.

The network is trained on 80 sequences consisting of 60 frames each, 10 more sequences are used for validation, and another 10 are used for testing. When the network was optimized for 16-bit precision and ran using Titan V GPU, it used 24.42 ms for 4x4-upsampling to 1920x1080-pixels.

3.3 DLSS

DLSS [5] and DLSS 2.0 [6] are architectures developed by NVIDIA, which performs deep learning temporal upsampling. Little public information is available about algorithm details, training setup, and performance. However, it is publicly known that DLSS 2.0 uses jittered input frames and motion vectors as input, where the motion vectors are used to provide temporal feedback to the algorithm. At its core is a convolutional auto-encoder. The algorithm is trained by upsampling 1920x1080 pixel images to 3840x2160 pixel images, which are then compared to a ground truth which is a 15360x8640 pixel anti-aliased image. No quantitative numbers describing the quality of the upsampled images are available, but the quality is good enough for the algorithm to be used in commercial products. The algorithm uses 0.579 ms on a NVIDIA RTX 2080 Super, 0.647 ms on a NVIDIA RTX 2070 Super and 0.736 ms on a NVIDIA RTX 2060 Super when upsampling to 1920x1080 pixels [27].

4 Deep Learning Controlled Temporal Upsampling

This section introduces a novel architecture for applying deep learning to temporal upsampling, referred to as Deep Learning Controlled Temporal Upsampling (DLCTUS). In addition to this, it explains the approach used for dataset generation and network training. Finally, it presents an efficient way to implement the architecture in DirectX 12. All code related to this section can be found in the thesis' GitHub repository [28].

4.1 Architecture Description

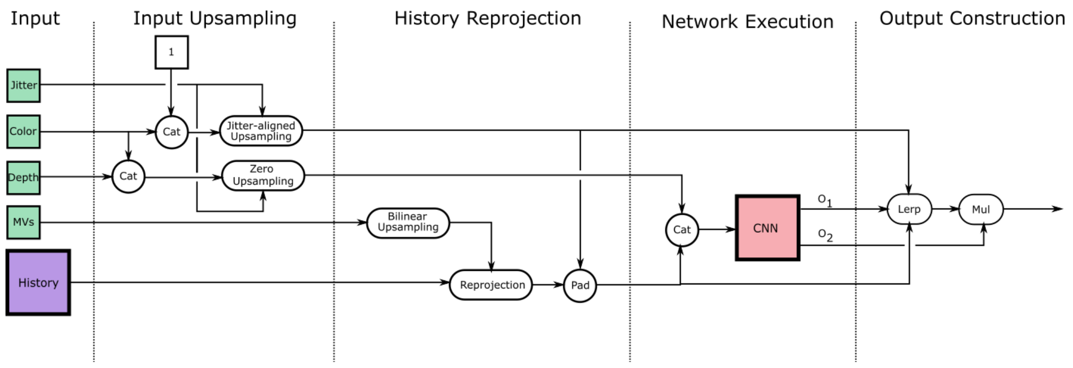


Figure 3: Illustration of the four stages of DLCTUS. The current frame input (green) consist of the frame's color in RGB-format, the frame's depth buffer, the frame's motion vectors (MVs), and the frame's jitter offset. The input from the previous frame (purple) consists of the frame's upsampled color in RGB-format, and the frame's accumulation buffer, both stored in the history buffer. The input upsampling stage (4.1.1) first concatenates (Cat) the frame color and depth and upsamples the result using zero-upsampling. Then the frame color is concatenated with a tensor consisting entirely of ones, resulting in a color tensor with four channels, where the three first are RGB and the fourth is one. The result of this concatenation is then upsampled using jitter-aligned upsampling. The history reprojection (4.1.2) stage fist upsamples the frame's motion vector using bilinear upsampling, and then use the upsampled motion vectors to reproject the history buffer. The reprojected history buffer is then padded (Pad) using the jitter-aligned upsampled frame color. The network execution stage (4.1.3) concatenates the zero-upsampled frame color and depth with the reprojected and padded history buffer, and use the results as input to a CNN. The structure of the CNN is illustrated in Figure 5. The output construction stage (4.1.4) starts linear interpolation (Lerp) between the jitter-aligned input and reprojected history using the first output of the CNN. Then it multiplies (Mul) the accumulation value of the result with the second output of the CNN to get the final result.

An overview of DLCTUS is illustrated in Figure 3. The figure shows that DLCTUS starts with upsampling the input, reprojecting the history, and passing these as input to a CNN. The outputs of the CNN are then used to combine the upsampled input with the reprojected history to construct an output. DLCTUS is similar to

TUS (2.4), but differs in that DLCTUS performs history rectification and sample accumulation at the same stage, and this stage is controlled by a CNN. In addition, an accumulation buffer is utilized to improve sample accumulation. The contents of the accumulation buffer is also controlled by the CNN, giving the network additional hidden state. The architecture can be split up into four parts: input upsampling, history reprojection, network execution, and output construction. These are explained in detail in 4.1.1, 4.1.2, 4.1.3, and 4.1.4, respectively. The handling of the accumulation buffer is explained in 4.1.5.

4.1.1 Input Upsampling

Two different upsampling methods are used on the input frame color: zero-upsampling and Jitter-Aligned Upsampling (JAU).

Zero-upsampling takes the input pixels and maps them to the target resolution pixel that corresponds with the input pixels' sample location, and leaves all other pixels as black. Zero-upsampling is used as input for the convolutional neural network as it provides information about the position of the input samples. The input RGB color is concatenated with the input depth value before zero-upsampling, creating an Red, Green, Blue and Depth (RGB-D) value for each input pixel. This is done to give the network information about the frame depth, which might be useful for the network.

JAU subtracts the jitter offsets from the sampling positions, which properly aligns the input frame with the output frame when either bilinear or bicubic upsampling is performed. This offset is important since it removes the error introduced from the shift of the input images. Removing this error makes JAU a much better representation of the upsampled current frame than using normal bilinear upsampling. Figure 4 compares JAU with normal upsampling. It shows that JAU is more stable than normal upsampling, and that bilinear is superior to bicubic when jittered input frames are used. Before JAU is performed the alpha channels of the input color are set to 1. This is done because 1 represents the correct accumulation buffer value for the input, which will be shown in 4.1.5.

4.1.2 History Reprojection

History reprojection (2.4.3) was used to reproject the history to the current frame. History reprojection requires motion vectors in the same resolution as the history buffer. However, the input motion vectors were in LR while the history buffer was in HR, which made upsampling of the motion vectors necessary. The motion vectors were upsampled using bilinear upsampling. This upsampling method was chosen since the motion vectors are mostly piece-wise smooth, which makes it possible for bilinear upsampling to accurately recreate missing motion vectors. A downside to the use of bilinear upsampling was that it introduced errors in discontinuous regions.

History reprojection requires an interpolation technique when the previous pixel position falls between pixels in the history buffer. In this case, bicubic interpolation

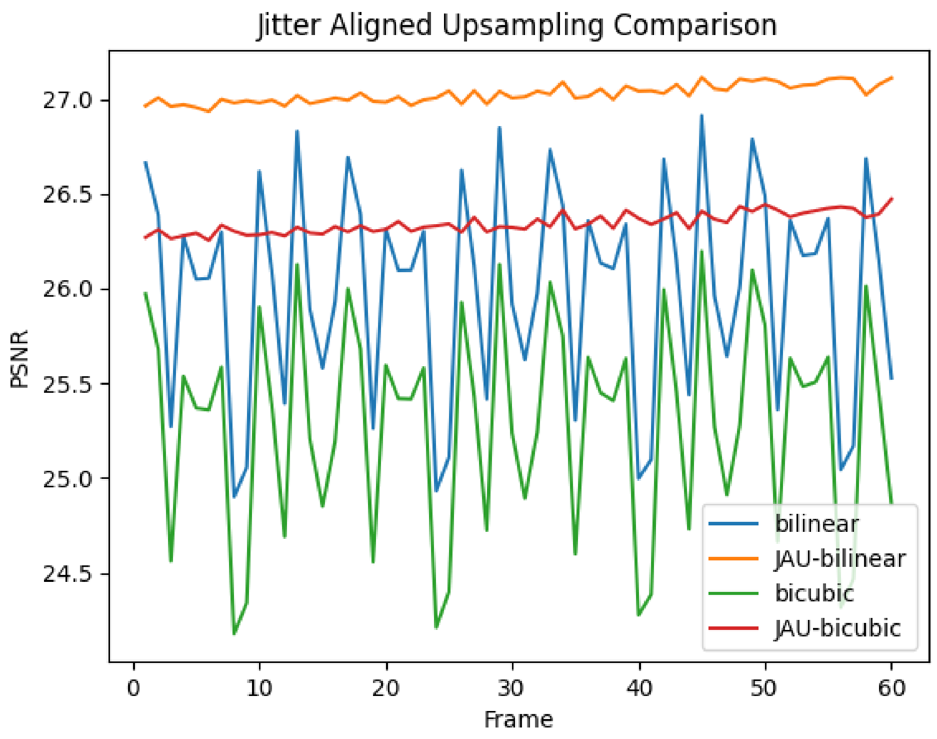


Figure 4: Comparison of jitter aligned upsampling with normal upsampling, using bilinear and bicubic interpolation. An upsampling factor of 4 is used, and the PSNR is calculated over the 3rd video in the test set.

was chosen because of its efficiency and high quality. The motion vectors can point to pixels outside the previous history buffer which makes a padding scheme necessary. Xiao et al. [26] uses zero padding, which matches well with their architecture since the reprojection is performed on zero upsampled inputs, which already have a lot of zeros. Using zero padding when reprojecting the history buffer will however create sharp discontinuities, which is difficult for the CNN to handle. This can be avoided by using the jitter-aligned upsampled input frame as padding, which will match the edge of history buffer better and introduce a less noticeable discontinuity. This approach is similar to most TAA implementations where α is set to 1 on pixels outside the history buffer, which will essentially lead to using the input frame as padding. Since the alpha channel of the jitter-aligned upsampled input frame is 1, the accumulation buffer value will be set to 1 during padding.

4.1.3 Network Execution

The structure of the convolutional neural network can be seen in Figure 5. The structure is similar to FEQE [17] in that it performs most of its computation in low resolution and uses a small amount of channels to increase the depth of the network. It also applies the residual block structure of EDSR [16] which improves image quality by removing batch normalization.

As shown in the figure, the network consists of three stages: a downsampling stage,

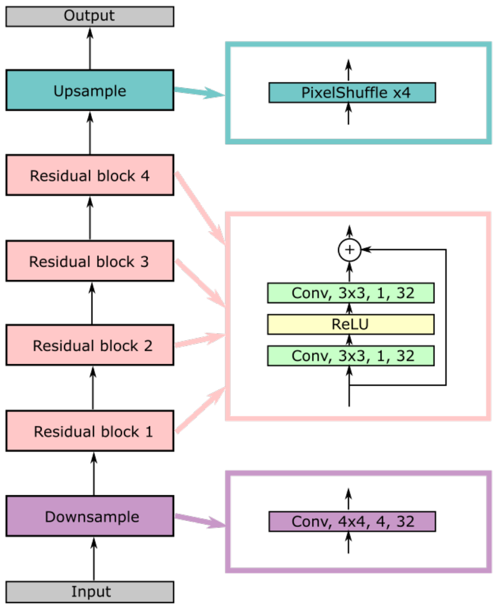


Figure 5: Detailed illustration of the CNN from Figure 3. The numbers in the convolutional layers represents the filter size, stride and output channels respectively. The first part of the network downsamples the input to a resolution that has a width and height that is four times smaller than the input, using a strided convolutional layer. Then the downsampled input is passed through a residual net consisting of four residual blocks. The last stage upsamples the output of the residual net back to the same resolution as the input using a pixel shuffle layer (2.1.2).

a residual network stage, and an upsampling stage. The downsampling stage starts with an input in HR with 8 channels. By using a convolution with 4x4 filter size and stride 4 the input is downsampled to a resolution with a width and height that is four times smaller than the input, with 32 channels. This downsampling method was chosen because it was found to be an efficient way to reduce the resolution at the same time as increasing the channels. However, the downsampling became a bottleneck as it reduces the information contained in $4 \times 4 \times 8 = 128$ input values into $1 \times 1 \times 32 = 32$ downsampled values. This bottleneck forces the network to compress the input and is necessary to reduce the inference time of the network. The second part of the network is a residual network with 4 residual blocks (2.1.4). Each block performs a 3x3 convolution followed by a ReLU activation function followed by a second 3x3 convolution, with a shortcut connection connecting the start and end of the residual block. The convolutions in the residual blocks use a replication padding of 1 and a stride of 1 to make sure the resolution is not changed by the convolution. This residual block architecture is identical to EDSR and was chosen because it has been shown to perform well for superresolution tasks. The last step of the network is upsampling, which is done using a PixelShuffle layer (2.1.2). This layer takes the 32 low resolution channels and rearranges them into high resolution buffer with only 2 channels.

4.1.4 Output Construction

The output of the network consists of a high resolution buffer with two channels, referred to as $o_1(p)$ and $o_2(p)$, where p is the pixel position. The first output, o_1, was clamped to stay between 0 and 1 and was used to linearly interpolate between the reprojected history and the jitter-aligned upsampled input colors using the following equation:

$$c_i'^{\text{Out}}(p) = c_i^{\text{JAU}}(p)o_1(p) + c_i^{\text{Hist}}(p)(1 - o_1(p)) \tag{44}$$

where i refers to the four components of the history buffer and jitter-aligned upsampled input, $i \in \{r, g, b, a\}$. c^{JAU} is the jitter-aligned input, and c^{Hist} is the reprojected and padded history buffer. This enabled the network to do both history rejection and sample accumulation through a single value. History rejection was performed by setting o_1 to 1 which would completely ignore the history buffer and return the jitter-aligned upsampled input color. Sample accumulation could be done by setting o_1 to a fraction, which would blend the input with the history. o_1 is similar to $\alpha \cdot \beta$ from 2.4.6, which is used to interpolate between the history value and the new sample value for TUS. The second output from the CNN, o_2, was clamped to be between 0 and 1 and multiplied with the accumulation buffer value of the output, shown in the following equation:

$$\begin{aligned} c_r^{\text{Out}}(p) &= c_r'^{\text{Out}}(p) \\ c_g^{\text{Out}}(p) &= c_g'^{\text{Out}}(p) \\ c_b^{\text{Out}}(p) &= c_b'^{\text{Out}}(p) \\ c_a^{\text{Out}}(p) &= c_a'^{\text{Out}}(p)o_2(p) \end{aligned} \tag{45}$$

The final RGB and accumulation values, c^{Out}, were then clamped to the 0 to 1 interval before the RGB values were returned as the final output color.

4.1.5 Accumulation Buffer

DLCTUS uses the fourth channel of the history buffer as an accumulation buffer. The accumulation buffer was first used for TAA and its purpose is to store information about how many samples have been accumulated in the respective pixel. This information enables faster accumulation of input samples, as a moving average of input samples can be used instead of an exponential moving average. The accumulation buffer used by DLCTUS was intended to be used as a quality indicator for each pixel, where 1 indicates the lowest quality and 0 indicates the highest. The lowest possible sample quality was the jitter-aligned upsampled input image, so the accumulation buffer value should be set to 1 where the color values were equal to the jitter-aligned upsampled input values. This was done by setting the fourth channel of the jitter-aligned upsampled input image to 1, which reduced Equation 44 and Equation 45 to:

$$c_a^{\text{Out}}(p) = (o_1(p) + c_i^{\text{Hist}}(p)(1 - o_1(p)))o_2(p) \tag{46}$$

This equation is similar to the accumulation formula used in TAA (Equation 35) when comparing $c_a^{\text{Out}}(p)$ to $\frac{1}{N'}$, o_1 to α^2, and $(1 - o_1)$ to $(1 - \alpha)^2$, except for the multiplication with o_2. This multiplication were added to give the network the ability to compensate for the reduced quality of an input sample due to the bilinear upsampling of the input samples.

4.2 Dataset Generation

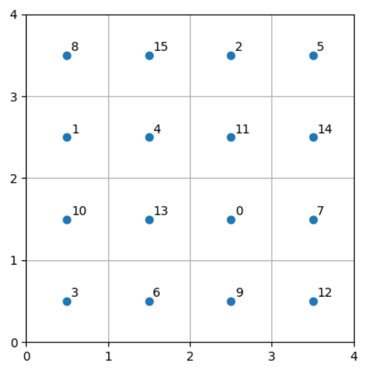
Large datasets are necessary for training deep neural networks. Since no standardized public dataset for deep learning temporal upsampling exist, an application to generate datasets was created. Details about this application is covered in 4.2.1, and details about the generated datasets are discussed in 4.2.2.

4.2.1 Testing Application

The dataset generation application was implemented using C++ and DirectX 12. DirectX was chosen because it was compatible with DirectML, which enabled efficient execution of the neural network. The application used deferred rendering to render the necessary images, motion vectors, and depths for the dataset. The rendered images were stored using 8-bits per color channel, the motion vectors were stored using 16-bits per channel, and the depth buffer was stored using 32-bits per channel. The memory format for color and motion vectors was chosen as they were the smallest possible formats that did not result in any visible reduction in image quality. The memory format of the depth buffer was chosen to be the same as the renderer's memory format for convenience.

4.2.2 Dataset

Each dataset used for training, validation, and testing was generated using 100 videos of 60 frames. The videos used realistic prerecorded camera movements that were captured using the testing application. Each dataset was split up into a training set of 80 videos, a validation set of 10 videos, and a testing set of 10 videos, similar to Xiao et al. [26]. Five items were saved from each frame, a low resolution input image, a low resolution input depth buffer, the low resolution input motion vectors, the input frame's jitter offset, and a high resolution supersampled target image. The resolution of the target was 1920x1080, and the resolution of the input differed depending on the upsampling factor. Two different upsampling factors were considered: 2 and 4, resulting in two different versions of the input data, with respectively a resolution of 960x540 and 480x270. Figure 6 shows the jittering patterns used during generation. For 4x4-upsampling, a jittering pattern where every sample is positioned in the center of a HR-pixel was used. The pattern was chosen so that no two consecutive samples are neighbors. This pattern was chosen because the center of the HR-pixel is a better estimate for the color of the whole HR-pixel, than any off-center sample. For 2x2-upsampling, a jittering pattern consisting of the 2nd, 3rd, 4th and 5th element of the Halton(2,3) sequence within each HR-pixel


(a) 4x4

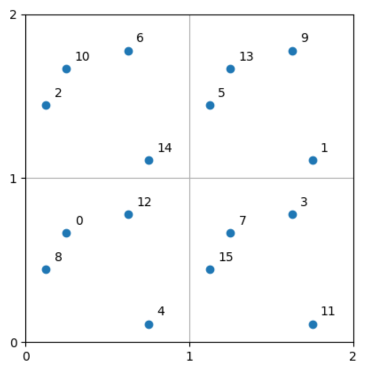

(b) 2x2

Figure 6: Sample patterns used for 4x4-upsampling and 2x2-upsampling. Each grid cell corresponds to a HR-pixel, while the whole grid corresponds to a LR-pixel.

was used. The HR-pixels are iterated over in a hourglass pattern. This pattern was chosen because it covers each HR-pixel well.

The mipmap bias (2.3.2) was set for all input frames to match the mipmap level of the target image. Two different datasets were created with different mipmap biases: one with a bias that matches 1 spp and one with a bias that matches 4spp. The formula used to calculate the input mipmap bias was as follows:

$$b = -\frac{1}{2}\log_2(f_u^2) - \frac{1}{2}\log_2(spp) \tag{47}$$

where b is the mipmap bias, f_u is the upsample factor, and spp is the samples per pixel that the target should match. The dataset with mipmap bias matching 1spp are referred to as dataset 1, while the dataset with a mipmap bias matching 4spp are referred to as dataset 2. The reason for using two different mipmap biases was to investigate how well the architecture can resolve the additional detail. 2x2-upsampling with 16 jitter points can result in a final image that is supersampled with 4 samples per pixel, which makes it possible to resolve the aliasing that appears as a result of reducing the mipmap bias. As mipmapping is not a perfect process, this approach will produce better images. However, this also introduces more aliasing to the input frames, which might be more difficult for DLCTUS to resolve.

The data was saved using the Portable Network Graphics (PNG) format. This format was chosen because of its lossless compression. The depth buffer's and motion vectors' formats are not supported by PNG, and a workaround was created by splitting each 32-bit float into 4 8-bit gray-scale pixels, and the two 16-bit floats into 4 8-bit gray-scale pixels. Prior to training, the data was converted from individual PNG files into a single Hierarchical Data Format 5 (HDF5) file. This was done because HDF5 files allow efficient loading of multiple cropped images, which was used during training. This decreased the training time, as the crops could be loaded directly rather than loading the full image, and then extracting the crop. However, this also increased the size of the dataset, as HDF5 does not use compression.

4.3 Training Details

This section provides details of the methodology used to train the network. It starts with a short overview of the whole training process, then it presents details about input cropping, loss calculation, the training process and validation, and testing. All network training and image quality evaluation were performed using PyTorch [29].

4.3.1 Training Overview

The network was trained on overlapping sub-sequences of the videos in the training set. The frames in each sub-sequence were cropped using random cropping, and the loss was calculated and accumulated using a selected number of the last outputs of the sub-sequence. The error was then back-propagated using BPTT.

For training the following values were used: a crop size of 256, a mini-batch size of 4, a sequence length of 30, and 5 target images. The training rans for 200 epochs with a learning rate of 10^{-4}.

4.3.2 Random Cropping

Random cropping was applied to the input and target images during training with the purpose of reducing GPU memory consumption. The target frame was cropped to a sub-square with sides of 256 pixels out of its original 1920x1080 pixels, and the low resolution input color, depth, and motion vectors were cropped to squares with sides of $256/f_u$, where f_u is the upsampling factor.

Generating the crops before loading the data, followed by only loading the crops, was found to significantly accelerate training opposed to loading the data before cropping.

4.3.3 Loss Calculation

The loss was calculated using the L1-loss function (Equation 7), as it has been shown to produce better results than mean square error (Equation 8) on image super-resolution tasks. In addition to the L1-loss, a spatio-temporal loss function was also tested. This spatio-temporal loss function was derived from Salvi's [25] spatio-temporal loss function (Equation 43). First, Salivi's function was discretized in the following way:

$$\hat{\mathcal{L}}_{\hat{\theta}}^{st}(I, \hat{I}) = \frac{1}{hwc} \sum_i \sum_j \sum_k \hat{l}_{\hat{\theta}}^{st}(I_{i,j,k} - \hat{I}_{i,j,k}, f_{\text{prev}}(I_{i,j,k}) - f_{\text{prev}}(\hat{I}_{i,j,k})) \tag{48}$$

with:

$$\hat{l}_{\hat{\theta}}^{st}(x, y) = x^2 + y^2 + \hat{\theta}(x - y)^2 \tag{49}$$

where $\hat{\mathcal{L}}^{st}_{\hat{\theta}}$ is the spatio-temporal loss, $\hat{l}^{st}_{\hat{\theta}}$ is a per pixel, per channel, spatio-temporal loss function, f_{prev} is a function that returns the value of the input in the previous frame, $\hat{\theta}$ is an arbitrary constant, and I and $\hat{I}$ are the reconstructed and target images respectively. Expanding $\hat{l}^{st}_{\hat{\theta}}$ gives:

$$\begin{aligned} \hat{l}^{st}_{\hat{\theta}}(x, y) &= x^2 + y^2 + \hat{\theta}x^2 + \hat{\theta}y^2 - 2\hat{\theta}xy \\ &= (1 + \hat{\theta})x^2 + (1 + \hat{\theta})y^2 - 2\hat{\theta}xy \end{aligned} \tag{50}$$

Introducing a new constant $\theta = \frac{\hat{\theta}}{1+\hat{\theta}}$, and by using two new functions $\mathcal{L}^{st}_{\theta} = \frac{\hat{\mathcal{L}}^{st}_{\hat{\theta}}}{1+\hat{\theta}}$ and $l^{st}_{\theta} = \frac{\hat{l}^{st}_{\hat{\theta}}}{1+\hat{\theta}}$ the loss becomes:

$$\mathcal{L}^{st}_{\theta}(I, \hat{I}) = \frac{1}{hwc} \sum_i \sum_j \sum_k l^{st}_{\theta}(I_{i,j,k} - \hat{I}_{i,j,k}, f_{\text{prev}}(I_{i,j,k}) - f_{\text{prev}}(\hat{I}_{i,j,k})) \tag{51}$$

with:

$$l^{st}_{\theta}(x, y) = x^2 + y^2 - 2\theta xy \tag{52}$$

The benefit of rewriting the loss in this way is that it binds θ to be between zero and one. With $\theta = 0$ the loss function is equal to the sum of the L1-loss of two consecutive frames, which does not include any temporal loss. However, when $\theta = 1$ the loss function will have a minimum when $x = y$, which is not desirable, because then the reconstructed image will not necessarily converge towards the target image. Setting θ to a value between 0 and 1 allows for a tradeoff between spatial and temporal loss.

Inspired by the performance of the L1-loss over MSE-loss this equation is linearized. This is done by first writing l^{st}_{θ} on the following form:

$$l^{st}_{\theta}(x, y) = x^2 + y^2 + \theta((x - y)^2 - x^2 - y^2) \tag{53}$$

and then exchanging the square with the absolute value:

$$l^{\text{st-linear}}_{\theta}(x, y) = |x| + |y| + \theta(|x - y| - |x| - |y|) \tag{54}$$

with a total loss of:

$$\mathcal{L}^{st-linear}_{\theta}(I, \hat{I}) = \frac{1}{hwc} \sum_i \sum_j \sum_k l^{st-linear}_{\theta}(I_{i,j,k} - \hat{I}_{i,j,k}, f_{\text{prev}}(I_{i,j,k}) - f_{\text{prev}}(\hat{I}_{i,j,k})) \tag{55}$$

Accurately calculating this loss function requires the function f_{prev}, which returns the value of a pixel in the previous frame. Finding this value is not trivial as the pixel can be under motion or not exist in the previous frame. During training an approximation was calculated by projecting the previous frame and the previous target to the current frame using the available motion vectors. This introduced errors to the loss function, but was found to work well regardless.

4.3.4 Network Training

The network was trained using BPTT. Each video sequence in the training set was split up into smaller overlapping sequences. The network was then iteratively executed on each input in the sequences using the output of the previous iteration as history. Finally, the loss of a few of the last iterations were calculated, accumulated, and back-propagated. Calculating loss for only a few of the last iterations, and not all iterations, allowed for faster training since the time spent on loading target images was reduced.

BPTT requires the hidden state to be initialized to a predetermined value. For DLCTUS the hidden state is the history buffer, which was initialized to the jitter-aligned upsampled input values of the first input frame, as this was the same values that were used for padding during history reprojection. This was advantageous since it allowed the network to train for both scenarios at the same time.

The sequence length used during training has a great impact on the final quality of the network. One use-case for the network are applications running at 60 frames per second, which will lead to 3600 frames in just a minute while the application might be used for hours. A sequence length of 3600 was unfeasible, especially when training on a GPU where all of the gradient data had to be stored on the GPU. Instead of a very large sequence length, a small one was used and several actions were taken to make sure that the network generalized temporally to longer sequences. The first action was to choose a sequence length that was large enough to ensure that the network was close to a steady state solution. Since only 16 jitter positions were used, no new information would enter the network after 16 iterations. Some information might however have been lost during the 16 first iterations, making a sequence length slightly larger than 16 necessary to ensure that the network was close to a steady state. A sequence length of 30 was found to be a good value. The second action was to calculate the loss at several time-steps. The idea was that it should be more difficult for the network to overfit on multiple targets than it was to overfit to exactly the last one. During training, the loss was calculated and accumulated for the five last iterations. The last action was to let the network only effect the output by interpolating between the history color and jitter-aligned upsampled input color. This limited the networks influence over the output which also limited its options for overfitting.

4.3.5 Validation and Testing

Cross validation was performed during training, to ensure that the network generalizes to data outside of the training set. The validation set consisted of 10 videos of 60 frames. Validation was done after each epoch of training, and had two steps. In the first step, loss was calculated in the same way as during training, but the random position of the crops were the same every time validation is performed to remove the randomness. The second step iterated from the start to the end of every video in the validation set and calculated the PSNR and SSIM for every iteration. The second step was performed to check the network's temporal generalization as it used the full video sequence, and not just a sub-sequence. It also made the validation comparable to networks trained with other loss functions and to the test results. The test results were calculated the same way as the second step of the validation. SSIM was calculated using MSSIM width a window size of 11, $\alpha = \beta = \gamma = 1$, and the constants $C_1 = 0.01^2$, $C_2 = 0.02^2$, and $C_3 = \frac{C_2}{2}$.

4.4 DirectX 12 Implementation

To showcase that the network can run in realtime beside a rendering pipeline a version was implemented in DirectX 12 in combination with DirectML. DirectML is a low-level machine learning API which allows for efficient execution of neural network layers on the GPU. It also interfaced well with DirectX 12, which made it possible to integrate an efficient neural network into a DirectX 12 rendering pipeline. The network was optimized to run on tensor cores, which enabled efficient 16-bit floating point operations. The execution was split up into three different parts: input preparation, network execution, and output construction.

4.4.1 Input Preparation

Input preparation was implemented as a single compute shader. This shader was executed once for each pixel in the high resolution history buffer. It performed zero-upsampling and JAU on the input frame, before it reprojected and padded the history buffer. The zero-upsampled input RGB-D values and reprojected history were then written to an output buffer. The reprojection method used was the bicubic convolution algorithm with $a = -0.75$ as this was the method used by PyTorch. The compute shader used an implementation of bicubic interpolation that needed 9 bilinear texture fetches, which was more efficient than the 16 memory loads necessary for the bicubic convolution algorithm, because the GPU is optimized for bilinear fetchers.

4.4.2 Network Execution

Several optimizations where performed in order to ensure that the network ran efficiently. Since the targeted graphics card was a NVIDIA RTX 2060, the NVIDIA Convolutional Layers User Guide [30] was followed to optimize the performance: the NHWC tensor memory format was used, and input and output channels from the convolutional layers was a multiple of 8 to make sure that the convolutions were performed on the tensor cores. Fused activations were also used, which incorporated the activation functions into the convolutional layer removing the need for another read and write for the activations.

4.4.3 Output Construction

Output construction was implemented as a full screen pixel shader. For each pixel on the high resolution output, it first performed JAU on the input frame. Then it read the reprojected history from the networks input buffer and the network output from the networks output buffer. After this, it interpolated between the reprojected history and jitter-aligned upsampled input, and modified the accumulation buffer. Lastly, it wrote the result to an output buffer.

4.4.4 Additional Optimizations

Two additional optimization were tested as an attempt to improve performance:

Optimization 1 merged upsampling with output construction. This was done by removing the pixel-shuffle layer form the network, and then performing manual pixel shuffling during output construction. This removed a read and write of the data, possibly improving performance.

Optimization 2 integrated parts of the downsampling with the input preparation compute shader. This was done by splitting the 4x4 convolution with stride 4, with an inverse pixel-shuffle operation, followed by an 1x1 convolution with stride 1. The two implementations were equivalent as long as the weights used for the 1x1 convolution were the same as the ones for 4x4 convolution, but pixel-shuffled. This implementation required an additional pixel shuffle of the reprojected history when the reprojected history was reused during output construction. This optimization would improve performance if the 1x1 convolution was faster than the 4x4 convolution, and if there is not too much overhead from the additional pixel-shuffle operations.

4.5 Recreation of Xiao et al. Method

A recreation of the architecture from Xiao et al. [26] was implemented to use as a reference point for DLCTUS. The algorithm was implemented and trained using the method described in the paper, except that the optional RGB to YCbCr transformation was omitted. Other possible differences were the format of the depth, and the jittering pattern, both of which were not specified in the paper. The jittering pattern used is described in 4.2.2, and the depth was linear and normalized to be in the 0 to 1 interval, where 0 is the near plane, and 1 the far plane. The training was done using the method described in the paper, however, the crop size for 2x2 upsampling had to be reduced to 192x192 pixels due to GPU memory constraints. The networks were trained for 300 epochs each.

5 Results

This section begins by covering the experimental setup used during dataset generation, network training, and network evaluation. After this, the results from the dataset generation are presented, followed by an evaluation of the DLCTUS architecture and results from the network training. Finally, the results from modified versions of DLCTUS are provided.

5.1 Experimental Setup

Dataset generation, network training, and network run-time testing were performed on a machine with an Intel Core i5-3570K 3.40Hz processor, a NVIDIA RTX 2060 GPU, and a Kingston A2000 M.2 SSD.

5.2 Dataset Generation

Table 1 shows the generation time and memory size of the dataset. The total size of the dataset is 29.06GB, which increases to 74.5GB when the dataset is stored as HDF5 data. Figure 7 illustrates the average distance to a pixel center, for each motion vector in the testing set of dataset 1 for 2x2-upsampling.

Resolution	Type	Time (min)	Size (GB)
1920x1080	Target	26.35	16.7
960x540	Input	15.97	9.46
480x270	Input	9.30	2.90

Table 1: Time consumption for generation and size of the dataset. Target images uses 64spp, while input images use 1spp, but include motion vectors, depth, and jitter positions.

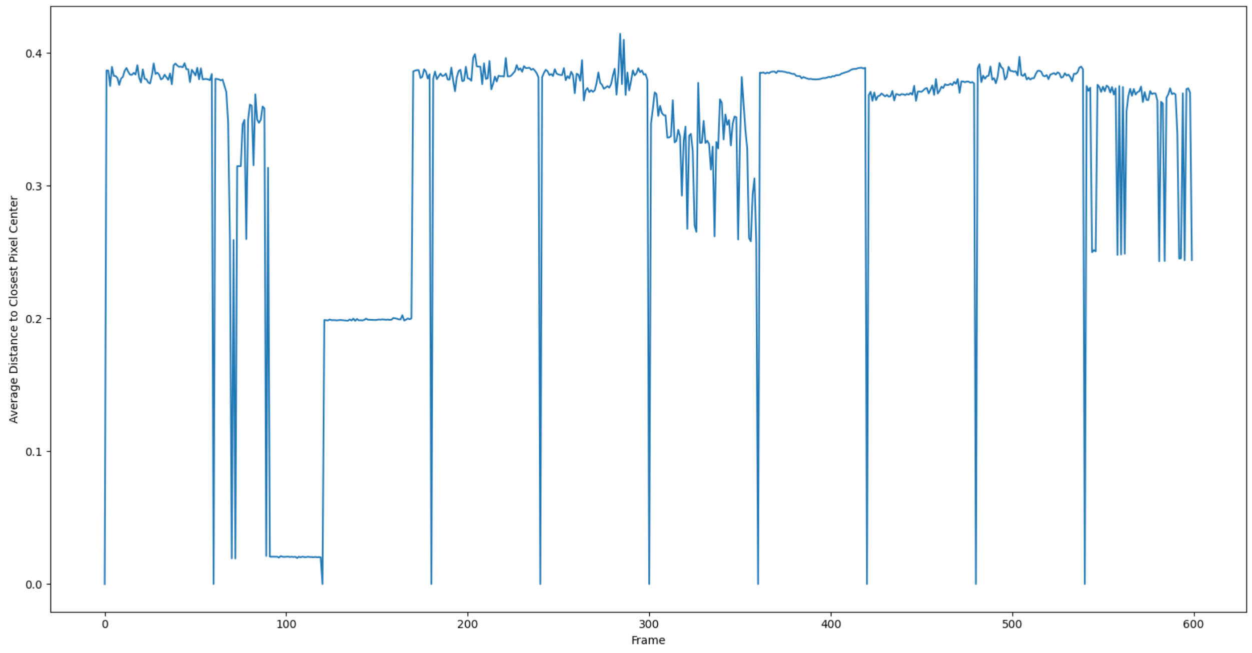


Figure 7: Average distance to the closest pixel center for each motion vector in every frame in testing dataset 1 for 2x2-upsampling.

5.3 Architecture Evaluation

This subsection cover the result of the networks after being trained using the method described in 4.3.4. The trained networks are referred to as DLCTUS(f_u, f_{dataset}), where f_u is the upsampling factor and f_{dataset} is the number of the dataset used to train the network. The results of three networks are considered in this subsection: DLCTUS(4, 1), DLCTUS(2, 1), and DLCTUS(2, 2).

5.3.1 Image Quality Metrics

Figure 8 and Figure 9 illustrates the PSNR and SSIM for each image in the testing dataset for 4x4 upsampling respectively. Figure 10 and Figure 11 illustrates the same for 2x2 upsampling. The results are compared to JAU, TUS, and Xiao et al. to give perspective. Table 2 shows the average SSIM and PSNR over all images in the testing set for the same methods. The TUS implementation uses history clamping, bicubic reprojection, no motion vector dilation and an α equal to 0.5 for 2x2 upsampling, and 1.0 for 4x4 upsampling. It is initialised and padded equivalently to DLCTUS.

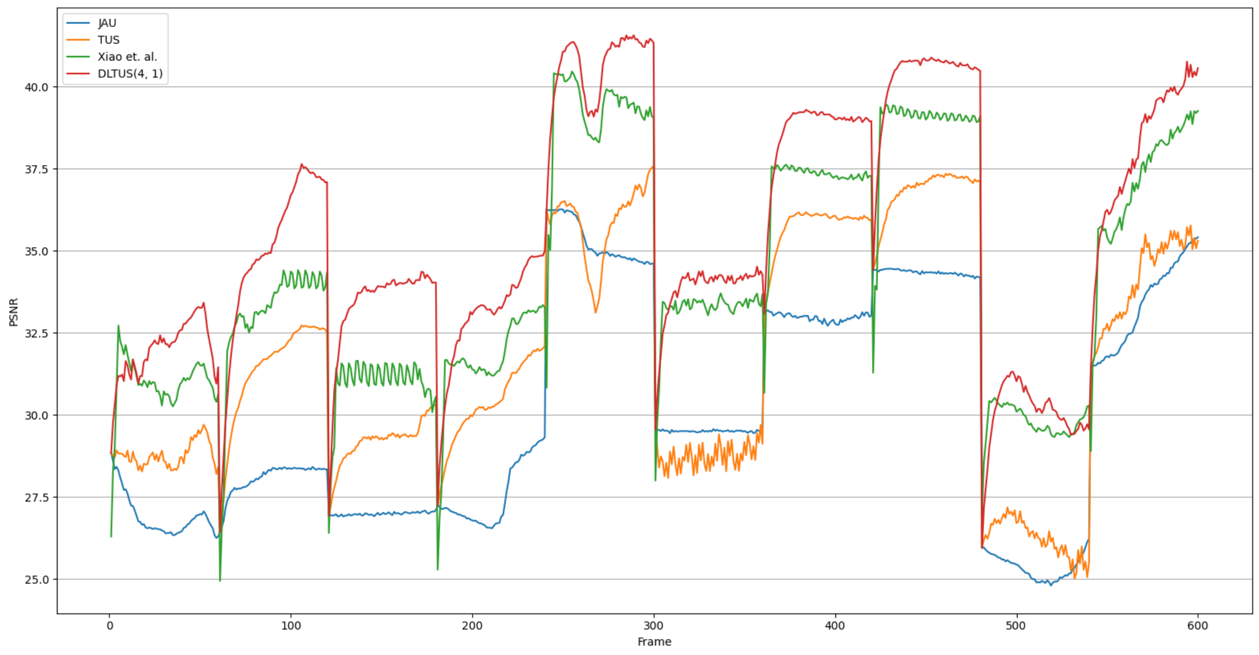


Figure 8: PSNR for the 10 videos in the testing set of dataset 1 for 4x4 jitter aligned upsampling, temporal upsampling, the Xiao et al. method, and DLCTUS(4,1).

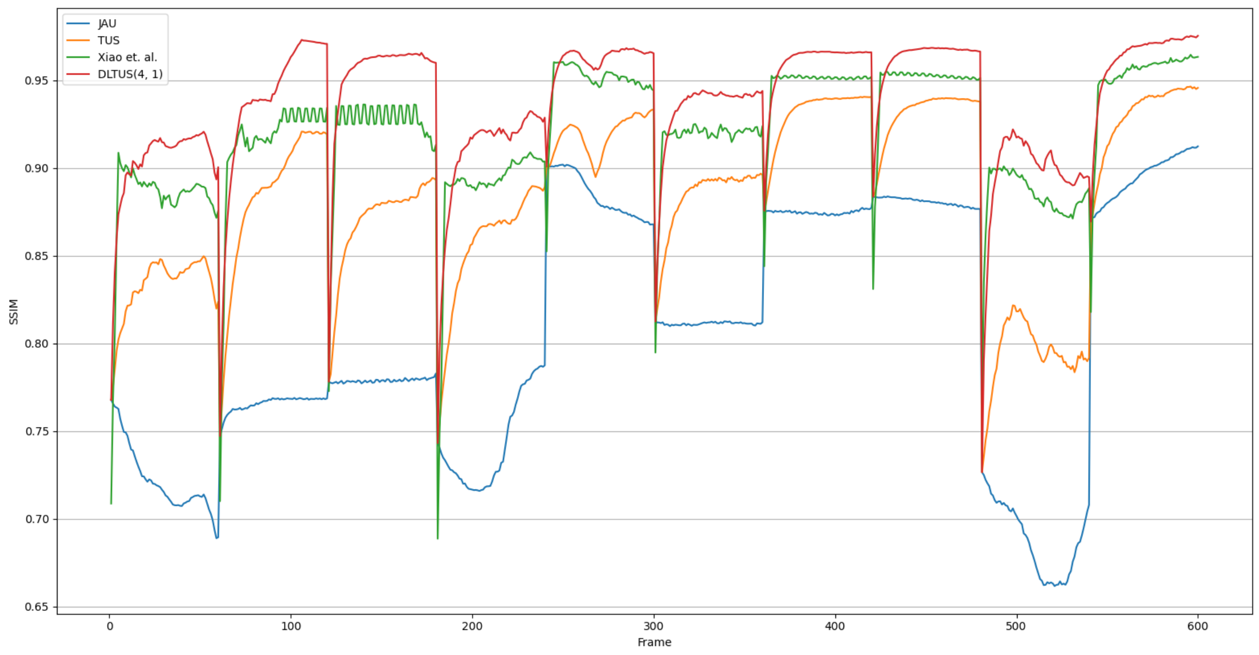


Figure 9: SSIM for the 10 videos in the testing set of dataset 1 for 4x4 jitter aligned upsampling, temporal upsampling, the Xiao et al. method, and DLCTUS(4,1).

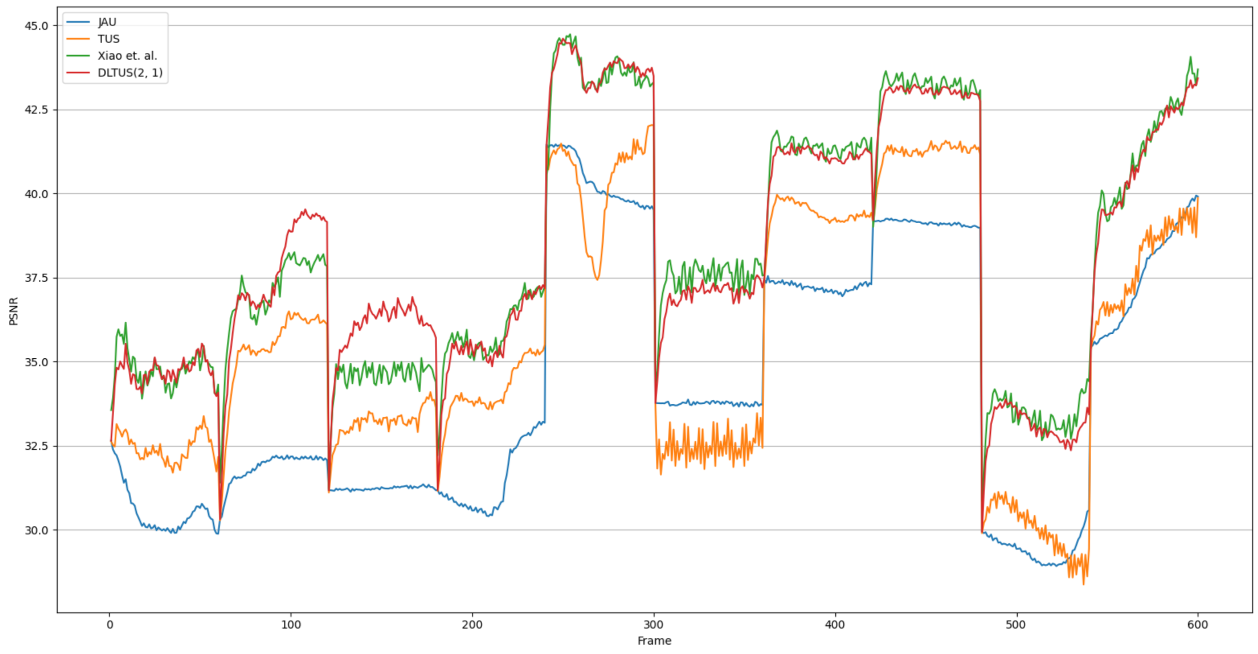


Figure 10: PSNR for the 10 videos in the testing set of dataset 1 for 2x2 jitter aligned upsampling, temporal upsampling, the Xiao et al. method, and DLCTUS(2,1).

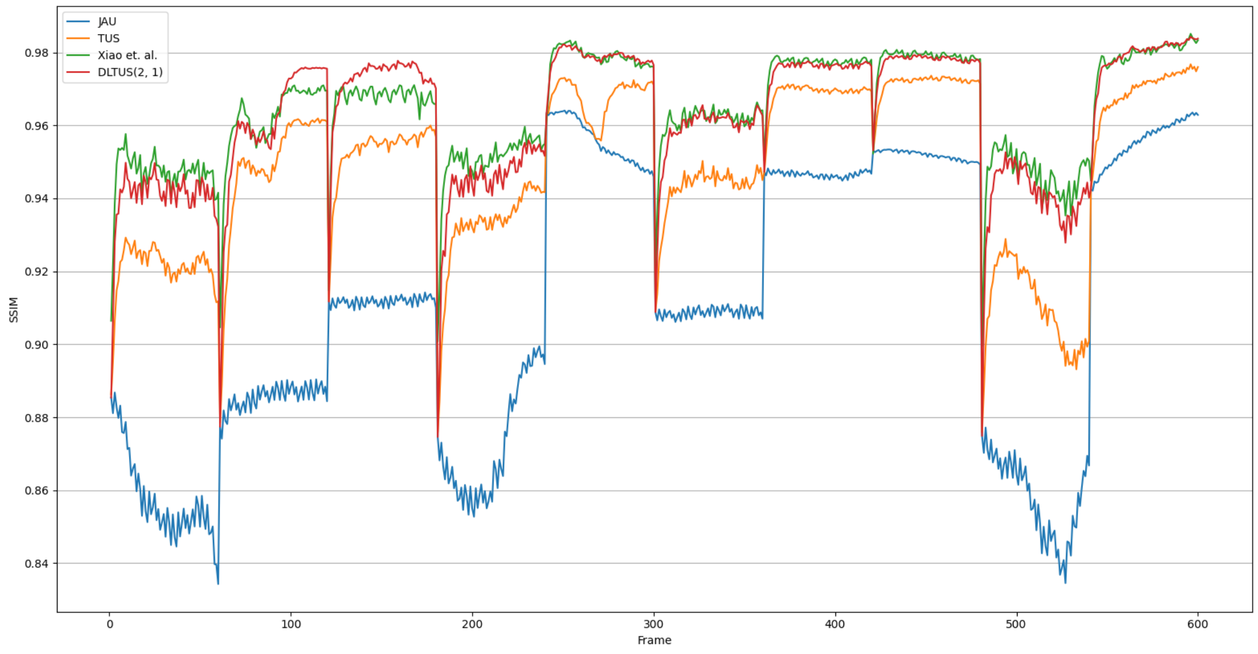


Figure 11: SSIM for the 10 videos in the testing set of dataset 1 for 2x2 jitter aligned upsampling, temporal upsampling, the Xiao et al. method, and DLCTUS(2,1).

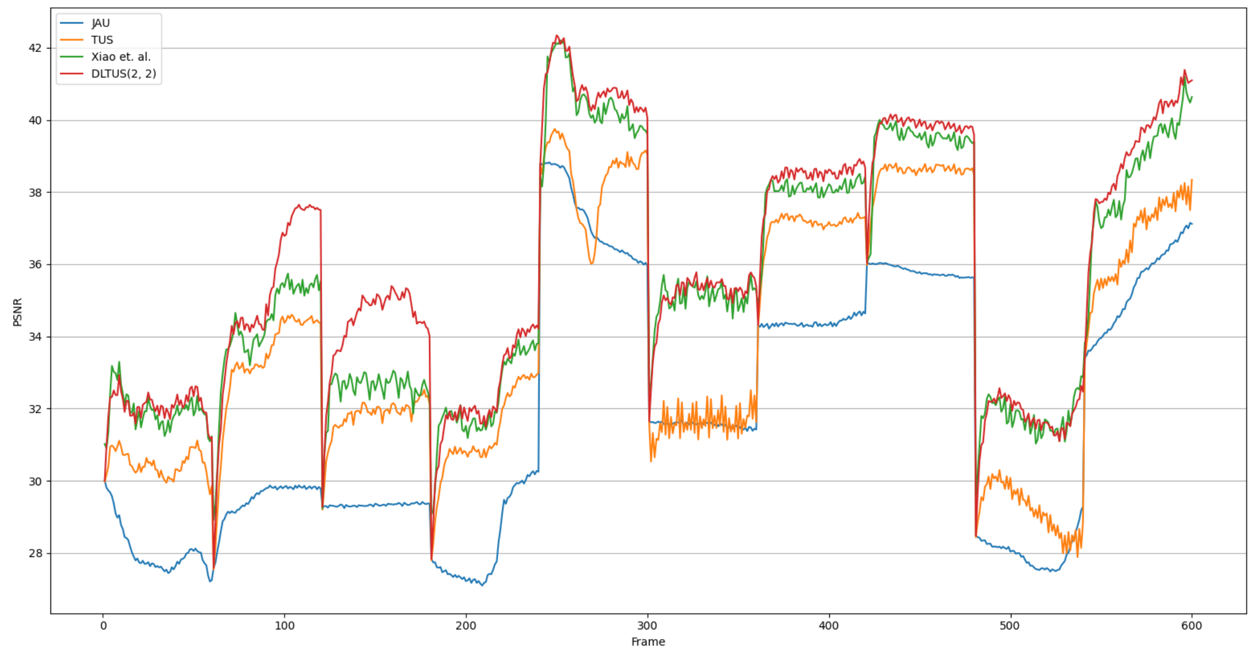


Figure 12: PSNR for the 10 videos in the testing set of dataset 2 for 2x2 jitter aligned upsampling, temporal upsampling, the Xiao et al. method, and DLCTUS(2,2).

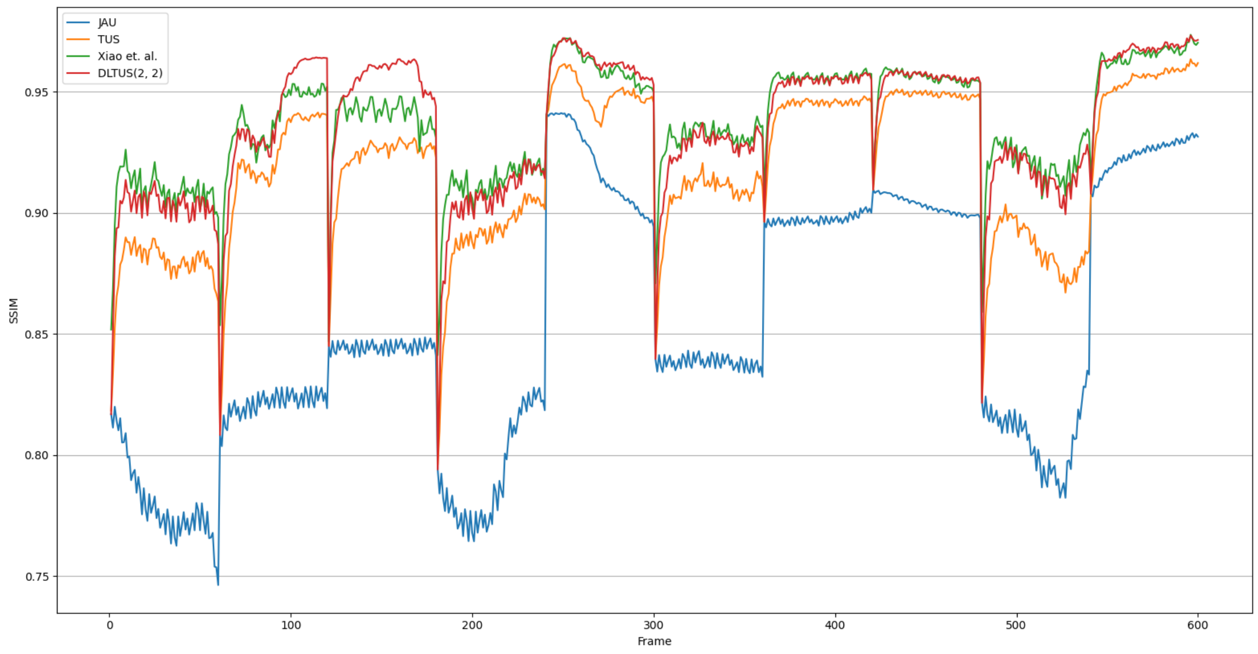


Figure 13: SSIM for the 10 videos in the testing set of dataset 2 for 2x2 jitter aligned upsampling, temporal upsampling, the Xiao et al. method, and DLCTUS(2,2).

f_u	Dataset	JAU	TUS	Xiao et al.	DLCTUS
4	1	30.04 dB / 0.8046	31.73 dB / 0.8830	34.18 dB / 0.9201	**35.49 dB** / **0.9376**
2	1	34.26 dB / 0.9104	35.65 dB / 0.9484	38.21 dB / **0.9645**	**38.22 dB** / 0.9625
2	2	31.78 dB / 0.8528	33.82 dB / 0.9188	35.40 dB / **0.9376**	**35.89 dB** / 0.9364

Table 2: Average PSNR / SSIM over the whole testing dataset using different upsampling factors. The methods used are JAU, TUS, Xiao et al., and DLCTUS. The DLCTUS version used, corresponds to DLCTUS(f_u, Dataset).

5.3.2 Visual Quality Evaluation

Two different frames from the testing set were used to illustrate the visual quality of the architecture. The first frame is shown in Figure 14 which is the 30th frame of the 1st video. In this frame the camera is under motion, which makes it good for illustrating the visual quality during motion. The second frame is shown in Figure 15 which is the 60th frame of the 2nd video. In this frame the camera has been motionless for a while, but the knight to the left is rotating, making it good at illustrating the visual quality of static frames. A total of six crops are used, where three are from the first frame, and three are from the second. The crops are marked in their respective figure with red squares. The first crop shows an area with a lot of detail. The second crop shows the edge of the knight's helmet during motion, with the area behind the helmet being recently disoccluded. The third crop shows a rod during motion with the area above being recently disoccluded, which is similar to the second crop, but differs in that the overall colors are darker. The fourth and fifth crop showcase two different places after the camera view has been stationary for a few frames, and the sixth crop shows the shadow of the rotating knight. The crops for 4x4 upsampling on dataset 1 is shown in Table 3, the crops for 2x2 upsampling on dataset 1 in Table 4, and the crops for 2x2 upsampling on dataset 2 in Table 5.

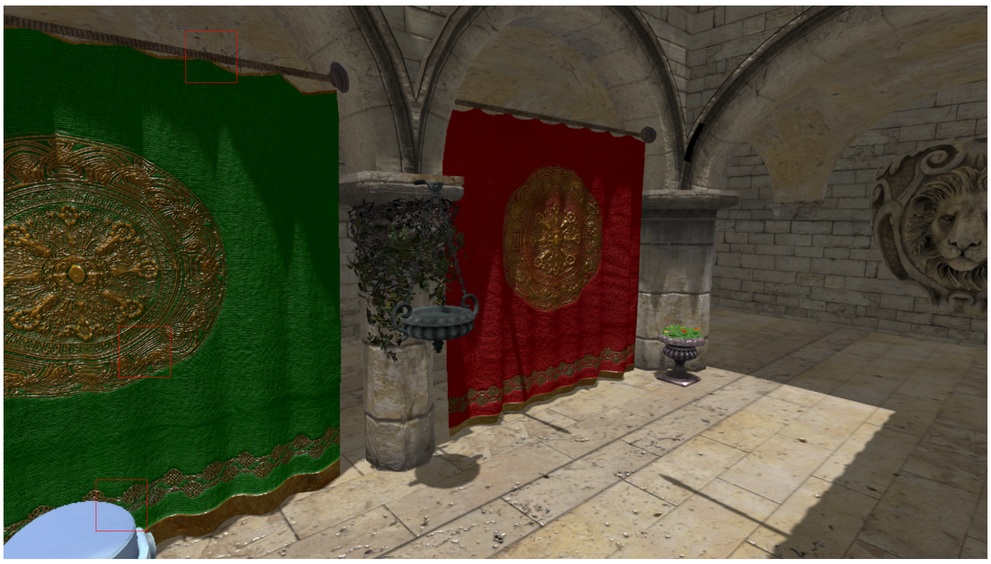


Figure 14: Frame 30 of video 1 of the testing dataset 2, upsampled using DLCTUS(2,2). The red squares show the position of the crops used for illustrations.

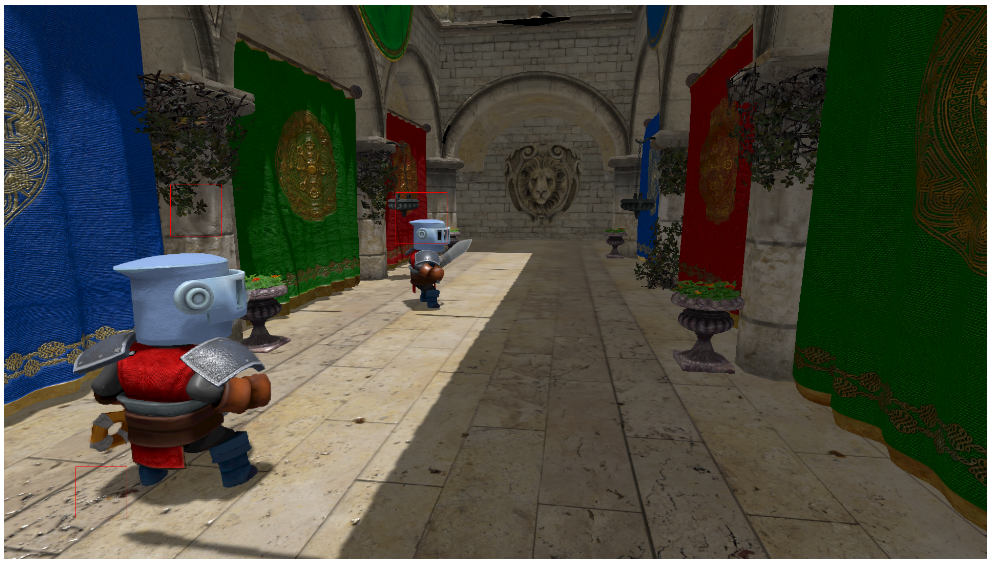


Figure 15: Frame 60 of video 2 of the testing dataset 1, upsampled using DLCTUS(4,1). The red squares show the position of the crops used for illustrations.

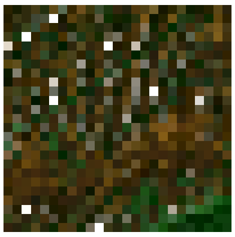
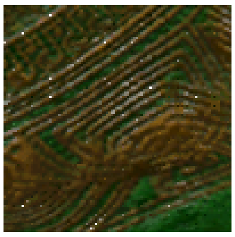
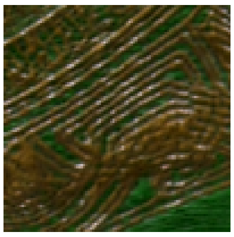
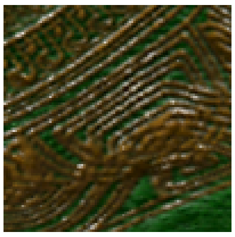
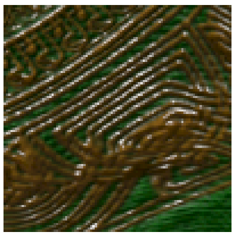
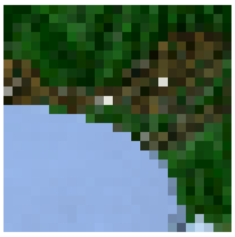
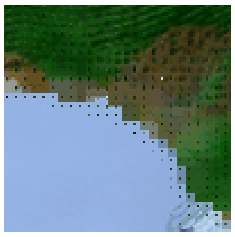
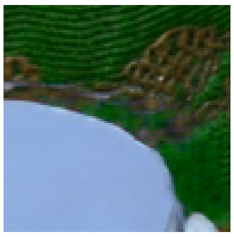
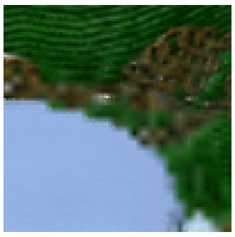
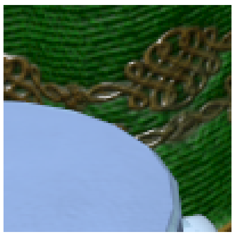
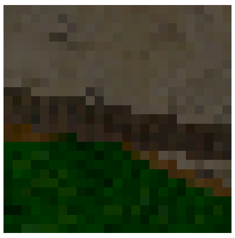
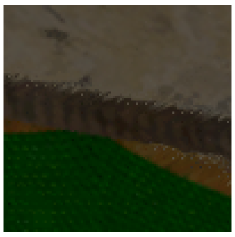
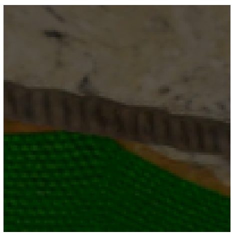
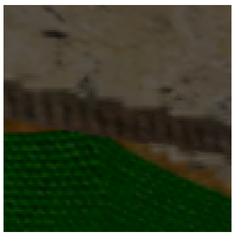
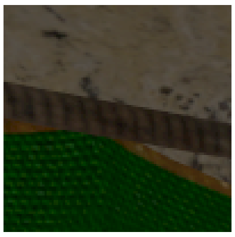
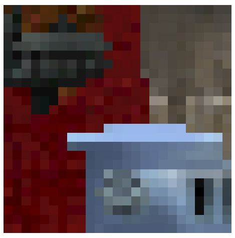
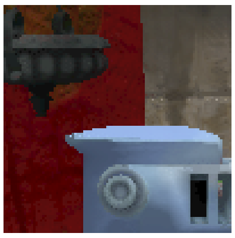
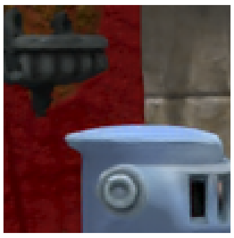
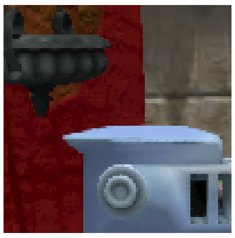
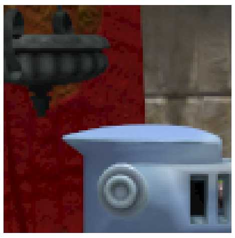
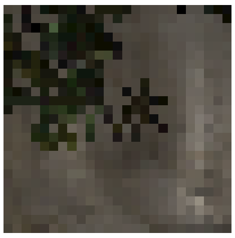
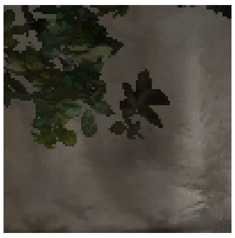
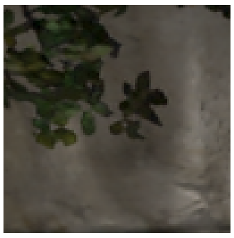
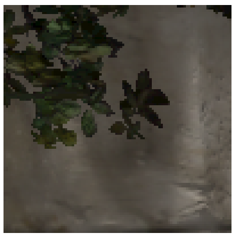
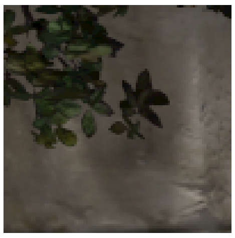
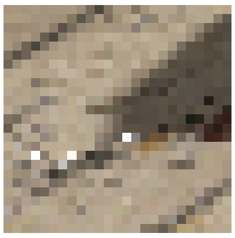
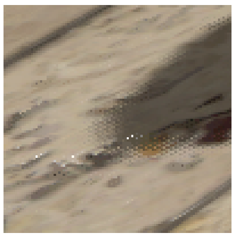
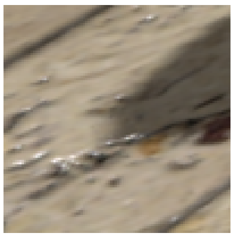
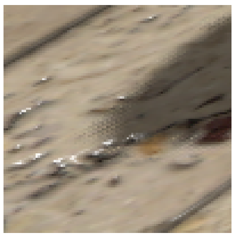
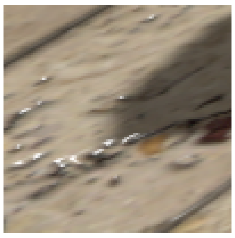
Crop	Input	TUS	Xiao et al.	DLCTUS(4,1)	64spp
1					
2					
3					
4					
5					
6					

Table 3: Visual quality comparison for 4x4-upsampling for 6 different crops in data-set 1. The demonstrated methods are: raw input, temporal upsampling, Xiao et al., DLCTUS(4,1), and 64spp ground truth.

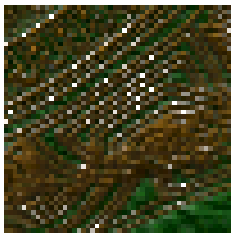
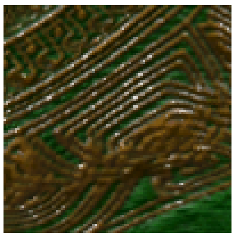
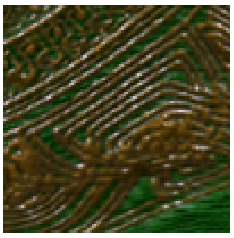
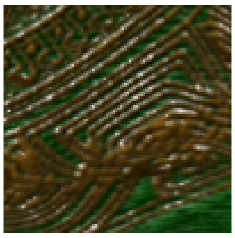
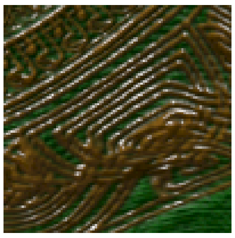
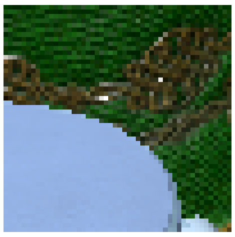
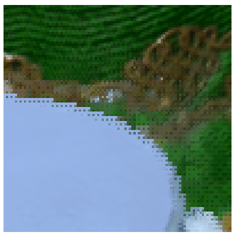
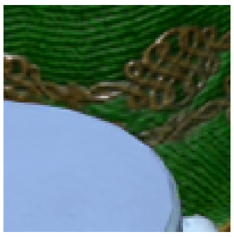
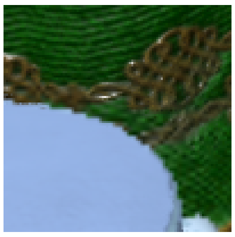
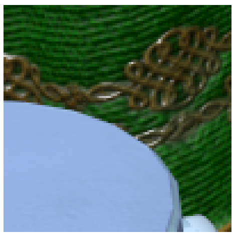
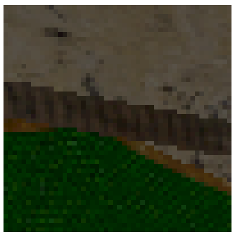
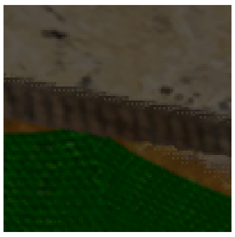
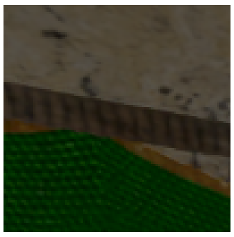
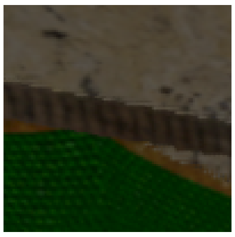
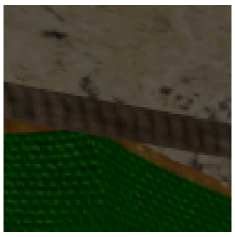
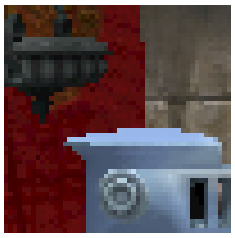
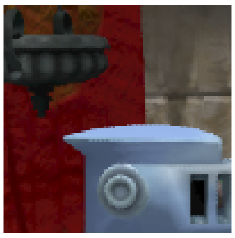
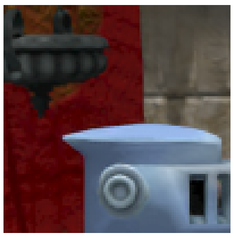
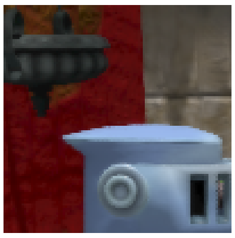
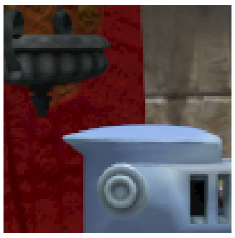
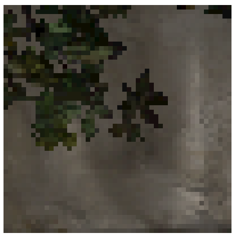
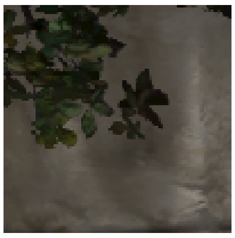
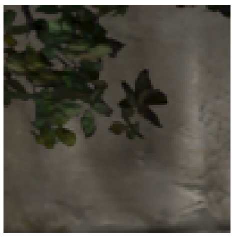
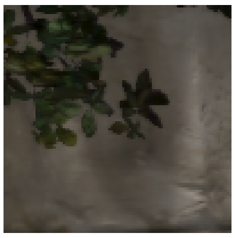
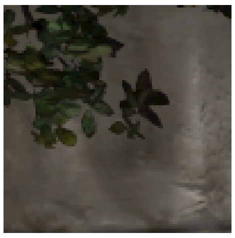
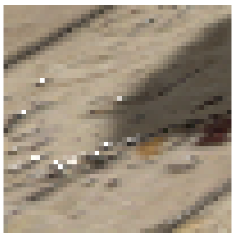
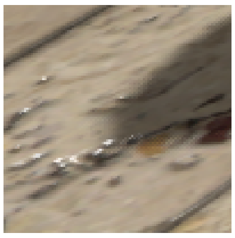
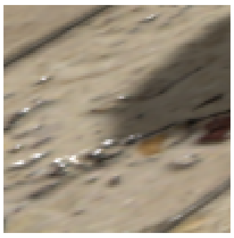
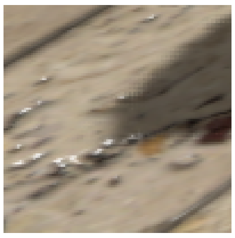
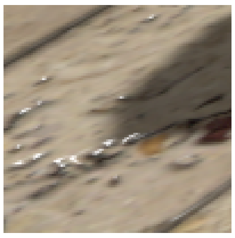
Crop	Input	TUS	Xiao et al.	DLCTUS(2,1)	64spp
1					
2					
3					
4					
5					
6					

Table 4: Visual quality comparison for 2x2-upsampling for 6 different crops in dataset 1. The demonstrated methods are: raw input, temporal upsampling, Xiao et al., DLCTUS(2,1), and 64spp ground truth.

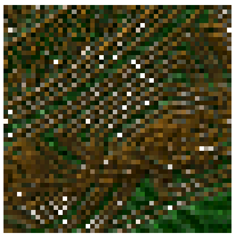
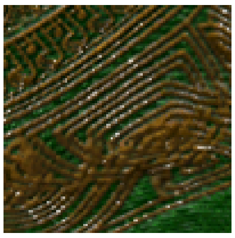
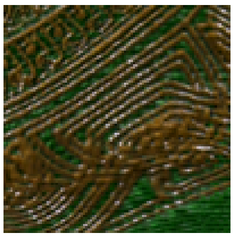
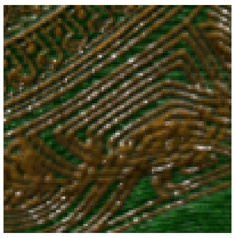
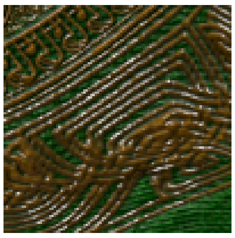
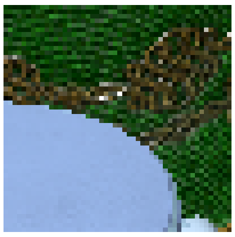
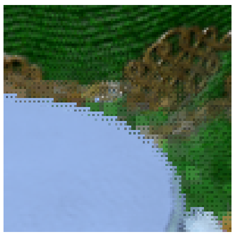
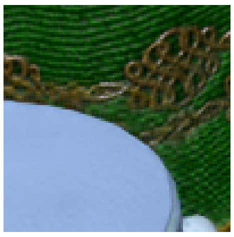
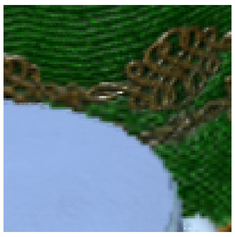
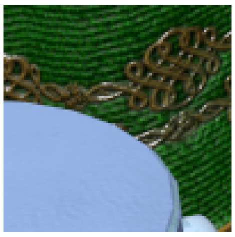
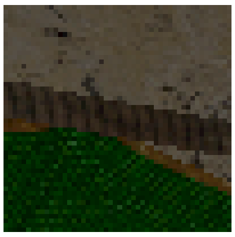
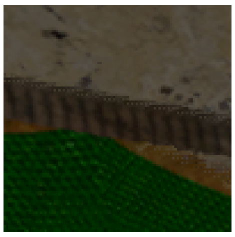
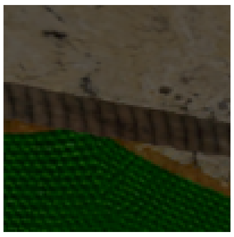
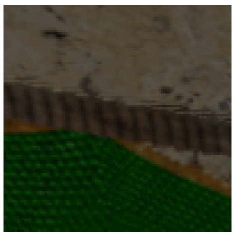
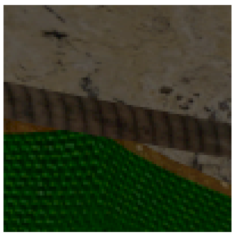
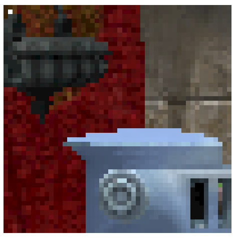
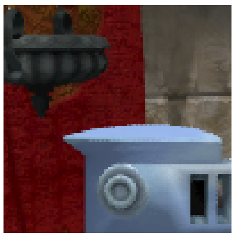
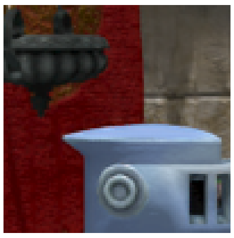
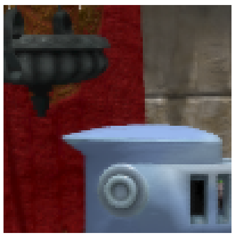
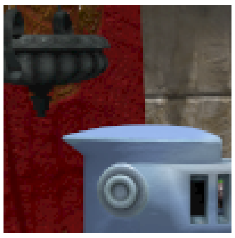
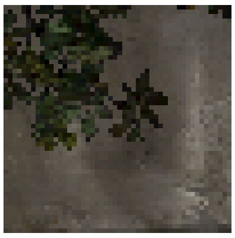
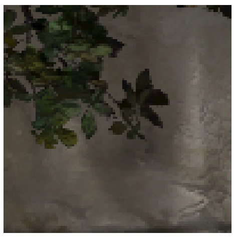
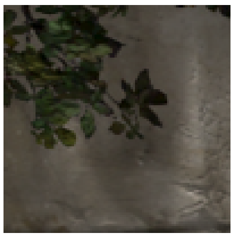
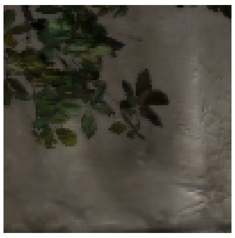
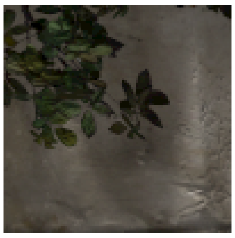
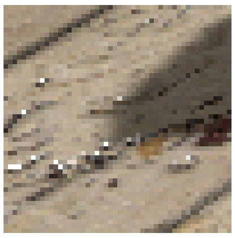
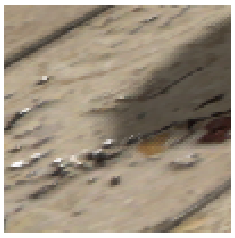
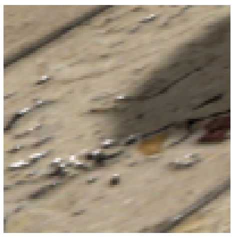
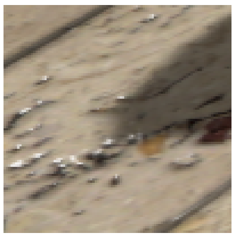
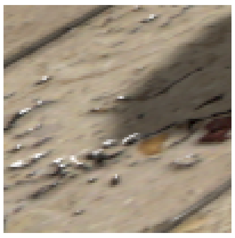
Crop	Input	TUS	Xiao et al.	DLCTUS(2,2)	64spp
1					
2					
3					
4					
5					
6					

Table 5: Visual quality comparison for 2x2-upsampling for 6 different crops in data-set 2. The demonstrated methods are: raw input, temporal upsampling, Xiao et al., DLCTUS(2,2), and 64spp ground truth.

5.3.3 Run-time

The time the DirectX 12 implementation use to execute the different layers of a residual block are shown in Table 6. Table 7 shows the time it takes for different parts of the convolutional neural network to run, and Table 8 shows the total time of the three parts of the implementation described in 4.4. The data was captured using PIX [31], and using a NVIDIA RTX 2060 GPU. The tables include the runtime with and without the optimizations described in 4.4.4, showing that both optimizations have a positive effect on the performance.

Conv3x3 and ReLU	Conv3x3	Add	Total
127µs	122µs	120µs	369µs

Table 6: Time used by the layers of a residual block. The structure of a residual block is illustrated in Figure 5.

Optimization	Down	Res Block 1	Res Block 2	Res Block 3	Res Block 4	Up	Total
None	358µs	369µs	369µs	369µs	369µs	271µs	2105µs
1	358µs	369µs	369µs	369µs	369µs	0µs	1834µs
1 & 2	198µs	369µs	369µs	369µs	369µs	0µs	1674µs

Table 7: Time used by the convolutional neural network, using different optimizations. The structure of the network is illustrated in Figure 5.

Upsampling factor	Optimization	Input Preparation	Network Execution	Output Construction	Total
4x4	None	569µs	2105µs	256µs	2930µs
2x2	None	563µs	2105µs	277µs	2945µs
4x4	1	569µs	1834µs	235µs	2638µs
2x2	1	563µs	1834µs	238µs	2635µs
4x4	1 & 2	320µs	1674µs	152µs	2146µs
2x2	1 & 2	323µs	1674µs	154µs	2151µs

Table 8: Total time used by the implementation, using different upsampling factors and optimizations. The stages and optimizations are described in detail in 4.4.

5.4 Network Training

This subsection covers the network training. First the results from every epoch of training are presented. Then the impact of training with a spatio-temporal loss function is examined.

5.4.1 Training Results

Figure 16 shows the training and validation performance of the three DLCTUS networks during training. The graphs provide insight into the training process and is important for evaluating the efficiency of the training. Table 9 shows the time it takes to train the networks.

f_u	Dataset	Training Time	Validation Time	Total Time
4	1	26.4 h	5.2 h	31.6 h
2	1	33.7 h	7.0 h	40.7 h
2	2	33.7 h	7.0 h	40.7 h

Table 9: Training time split up over training and validation over the 200 epochs of training, for different upsampling factors and on different datasets.

5.4.2 Loss Function

To investigate how the impact of a spatio-temporal loss function could be used to improve temporal stability, the network was trained with $\mathcal{L}_{\theta}^{st-linear}$ from Equation 55 with $\theta = 0.1$, $\theta = 0.5$, and $\theta = 0.9$. The results are compared with the same network trained with L1-loss, and shown in Table 10. The average of $\mathcal{L}_{\theta}^{st-linear}$ for $\theta = 0$ and $\theta = 1$ over the testing dataset are used to compare how the different loss functions prioritize temporal loss over spatial loss. All training and testing are performed with 2x2-upsampling on dataset 2. Specific crops that highlight the difference between the two networks are shown in Table 11.

Training loss function	$\mathcal{L}_{0}^{st-linear}$	$\mathcal{L}_{1}^{st-linear}$	PSNR	SSIM
L1	0.0166	0.00586	35.89 dB	93.65
$\mathcal{L}_{0.1}^{st-linear}$	0.0166	0.00585	35.90 dB	93.60
$\mathcal{L}_{0.5}^{st-linear}$	0.0166	0.00574	35.87 dB	93.58
$\mathcal{L}_{0.9}^{st-linear}$	0.0179	0.00552	35.36 dB	92.78

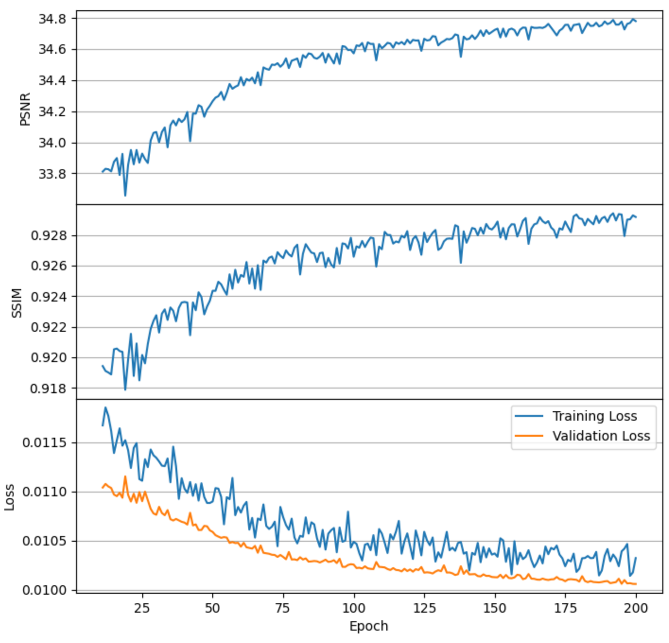
Table 10: Average spatial loss ($\mathcal{L}_{0}^{st-linear}$), temporal loss ($\mathcal{L}_{1}^{st-linear}$), PSNR, and SSIM over the whole testing dataset for DLCTUS(2,2) trained with different loss functions. The loss functions used are L1-loss and spatio-temporal loss, $\mathcal{L}_{\theta}^{st-linear}$, with θ equal to 0.1, 0.5 and 0.9.

5.5 Network Variations

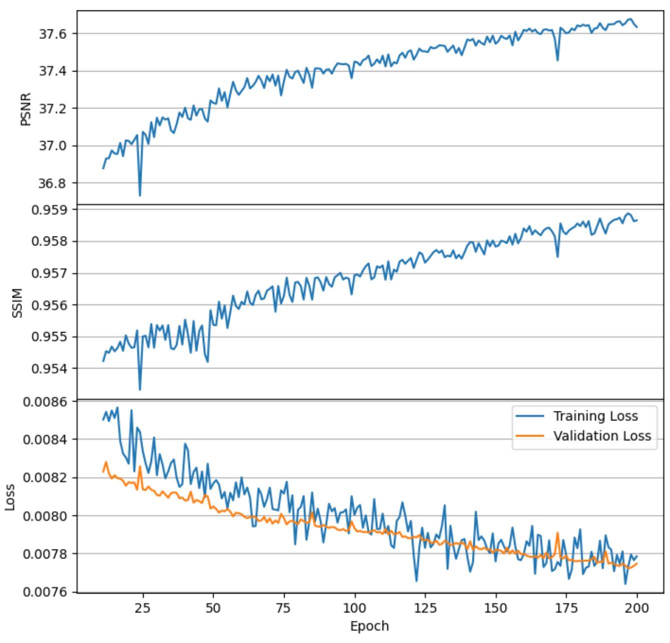
This subsection cover how different changes to the architecture impacts its results. Changes to three different parts of the architecture are considered: reprojection method, motion vector dilation, and accumulation buffer.

5.5.1 Reprojection Method

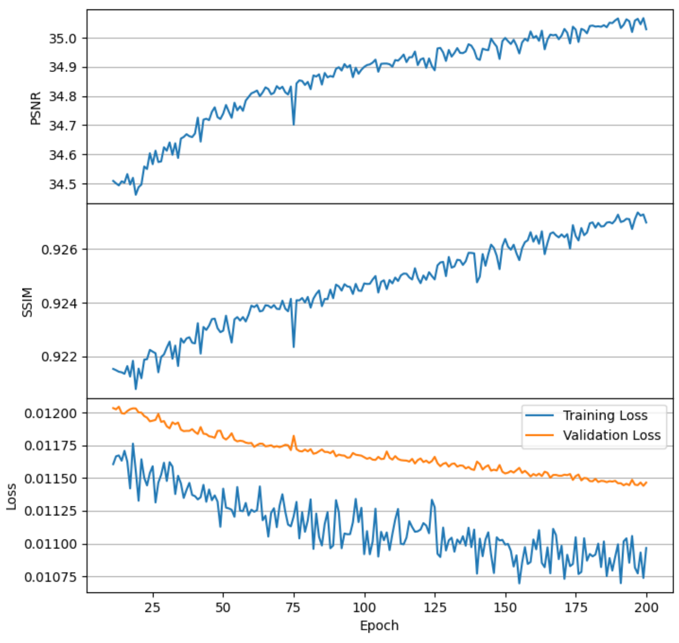
To inspect how the reprojection method effects the runtime and visual quality of the architecture, three different reprojection methods were tested. The first method uses the bicubic convolution algorithm mentioned in 2.2.1, and will be referred to as



(a) DLCTUS(4,1)



(b) DLCTUS(2,1)



(c) DLCTUS(2,2)

Figure 16: Training loss, validation loss, validation PSNR, and validation SSIM for DLCTUS(4,1), DLCTUS(2,1), and DLCTUS(2,2) after each epoch of training. The first 10 epochs are not shown to highlight the important part.

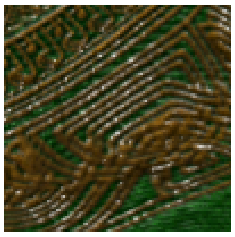
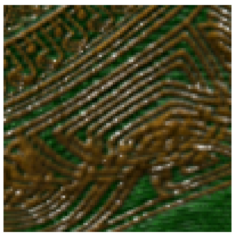
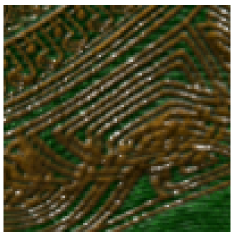
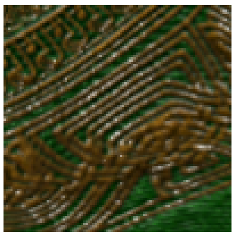
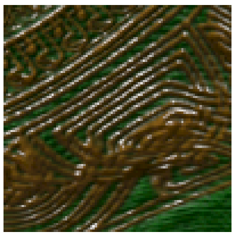
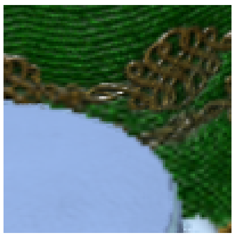
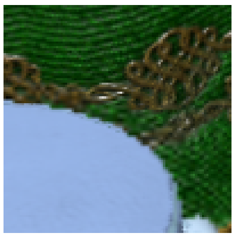
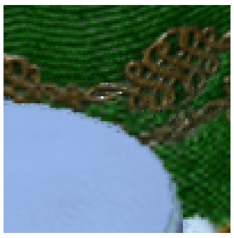
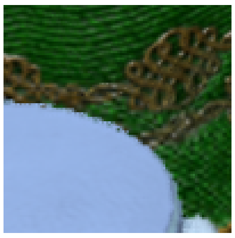
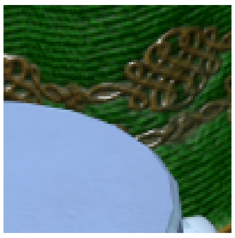
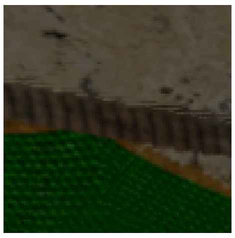
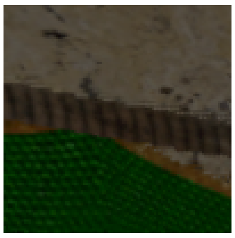
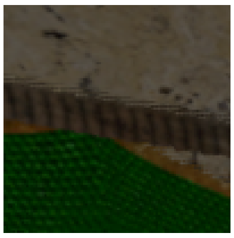
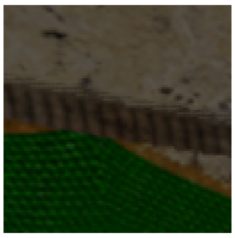
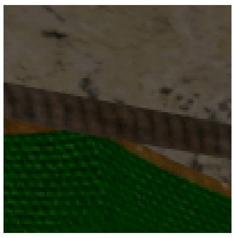
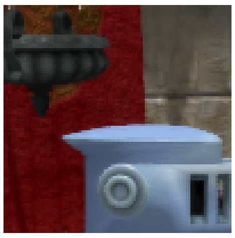
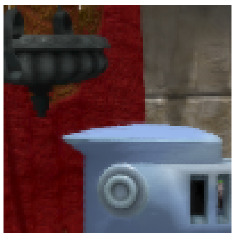
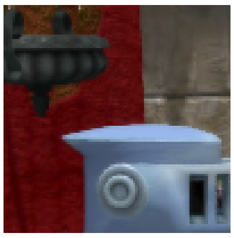
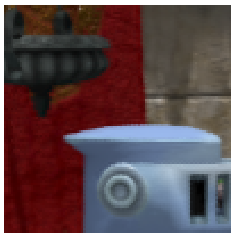
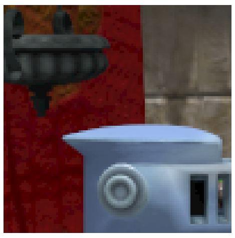
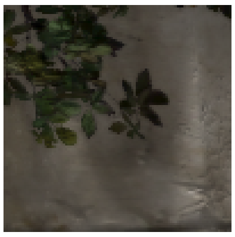
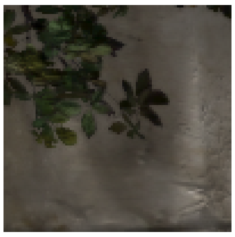
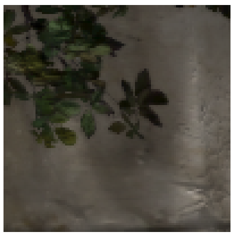
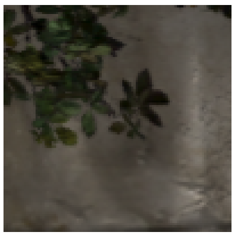
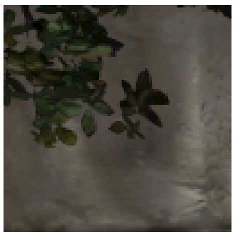
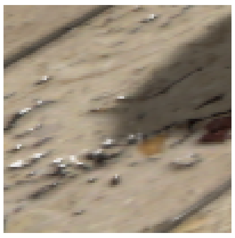
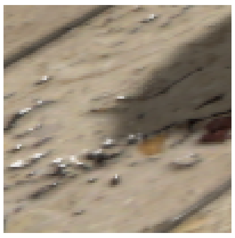
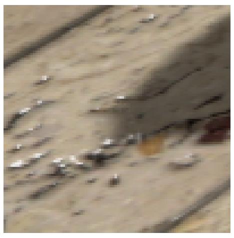
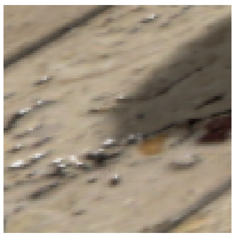
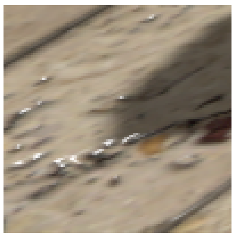
	DLCTUS(2,2) trained with loss function				
Crop	L1	$\mathcal{L}_{0.1}^{st-linear}$	$\mathcal{L}_{0.5}^{st-linear}$	$\mathcal{L}_{0.9}^{st-linear}$	64spp
1					
2					
3					
4					
5					
6					

Table 11: Visual quality comparison between using L1-loss and spatio-temporal loss with θ equal to 0.1, 0.5 and 0.9 for DLCTUS(2,2).

Bicubic9, as it requires 9 bilinear texture fetches. The second method is a bicubic interpolation approximation used by Unreal Engine 4, that relies on 5 bilinear texture fetches and not 9. The Unreal Engine implementation approximates the bicubic convolution algorithm with $a = -0.5$. However, in order to make it more similar to the PyTorch version, the approximation was changed to approximate bicubic interpolation with $a = -0.75$ instead of $a = -0.5$. This method will be referred to as Bicubic5. The last reprojection method considered is bilinear interpolation. Table 12 shows the runtime of input preparation for the DirectX 12 implementation when using the different reprojections methods. Table 13 shows the average PSNR and SSIM over the testing set. During training Bicubic9 was used for all networks.

f_u	Bilinear	Bicubic5	Bicubic9
4	241µs	280µs	320µs
2	243µs	284µs	323µs

Table 12: Time used by the input preparation when using optimization 1 and 2 and different reprojection methods.

f_u	Dataset	Bilinear	Bicubic5	Bicubic9
4	1	34.02 dB / 0.9080	35.27 dB / 0.9370	35.49 dB / 0.9376
2	1	36.54 dB / 0.9429	38.26 dB / 0.9635	38.22 dB / 0.9625
2	2	34.32 dB / 0.9064	35.92 dB / 0.9380	35.89 dB / 0.9364

Table 13: Average PSNR / SSIM for DLCTUS over the testing dataset for different upsampling factors and reprojection methods.

5.5.2 Motion Vector Dilation

Table 14 shows the PSNR and SSIM from testing the networks with and without motion vector dilation. Every motion vector was dilated by using the motion vector of the pixel in the 3x3 neighborhood that is closest to the camera. Table 15 shows specific crops where motion vector dilation had an effect.

f_u	Dataset	With Dilated Motion Vectors	Without Dilated Motion Vectors
4	1	35.14 dB / 0.9303	35.49 dB / 0.9376
2	1	38.26 dB / 0.9620	38.22 dB / 0.9625
2	2	35.89 dB / 0.9356	35.89 dB / 0.9364

Table 14: Average PSNR / SSIM for DLCTUS over the whole testing dataset for different upsampling factors, with and without motion vector dilation.

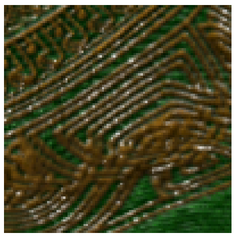
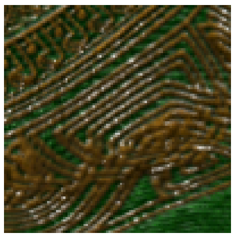
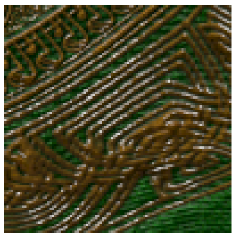
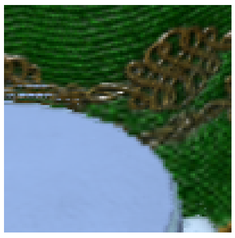
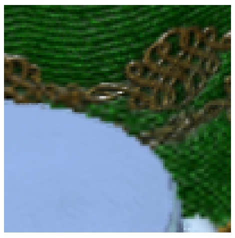
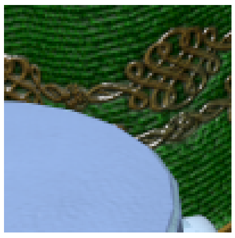
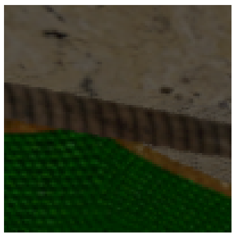
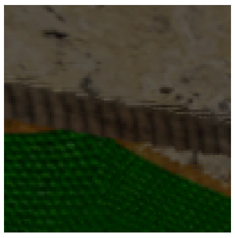
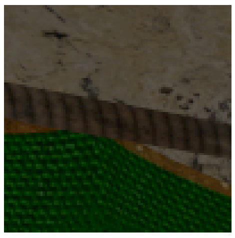
Crop	With Dilated Motion Vectors	Without Dilated Motion Vectors	64spp
1			
2			
3			

Table 15: Visual quality comparison between using and not using motion vector dilation for DLCTUS(2,2).

5.5.3 Accumulation Buffer

To justify the use of an accumulation buffer two networks were trained without an accumulation buffer, and compared to the networks with an accumulation buffer. The first network was trained for 4x4 upsampling on dataset 1, and the other network was trained for 2x2 upsampling on dataset 2. The average PSNR and SSIM is shown in Table 16, and the SSIM for each frame in the testing sets are shown in Figure 17. Table 17 shows a visualisation of how DLCTUS use the accumulation buffer. The table shows the content of the accumulation buffer for each of the crop positions from 5.3.2 for both DLCTUS(4,1) and DLCTUS(2,1) as a gray-scale image.

Upsampling Factor	Test Dataset	With Accumulation Buffer	Without Accumulation Buffer
4x4	1	35.49 dB / 0.9376	35.23 dB / 0.9347
2x2	2	35.89 dB / 0.9365	35.80 dB / 0.9359

Table 16: Average PSNR / SSIM for DLCTUS with and without the use of an accumulation buffer.

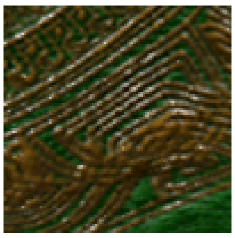
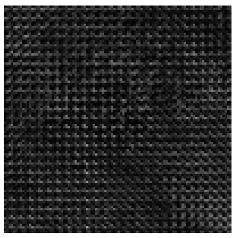
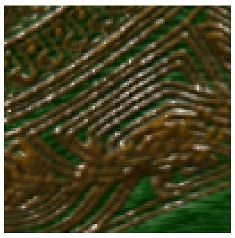
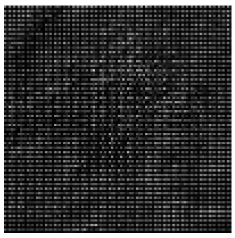
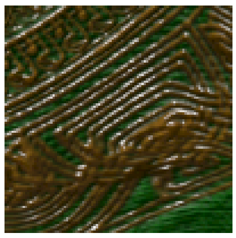
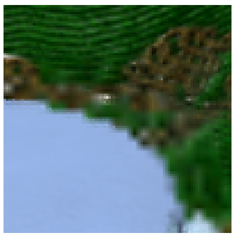
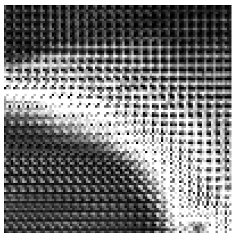
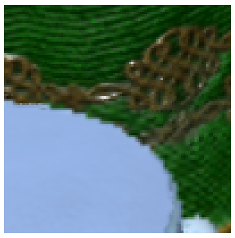
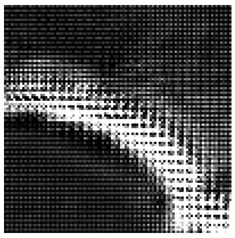
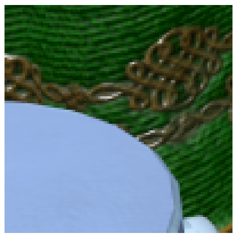
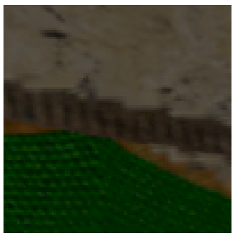
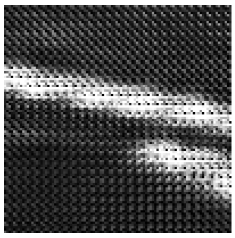
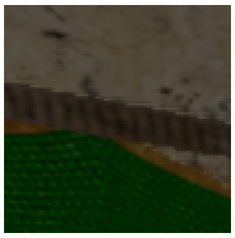
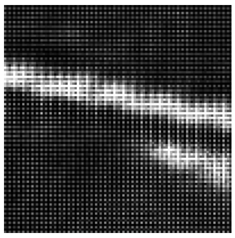
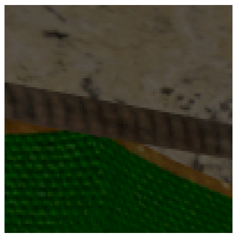
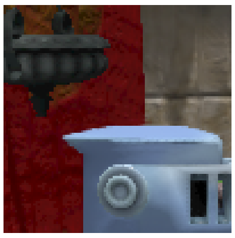
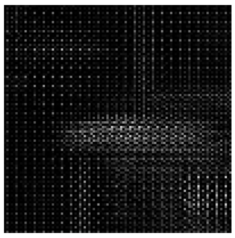
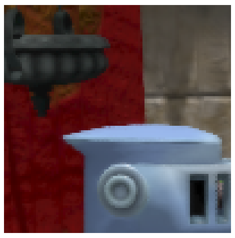
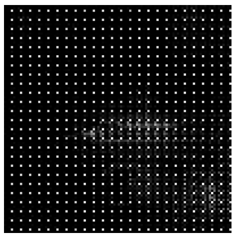
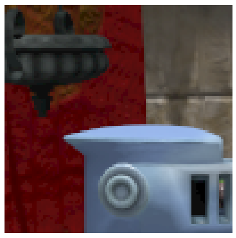
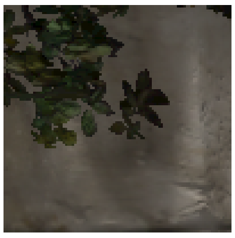
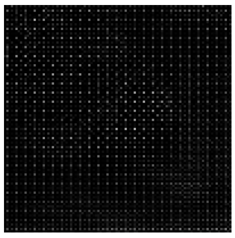
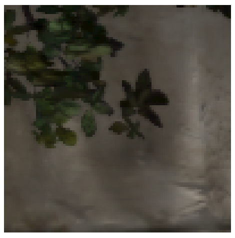
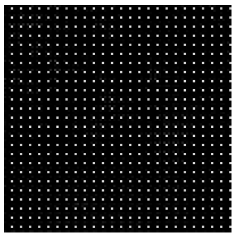
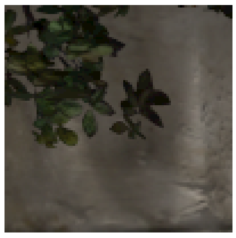
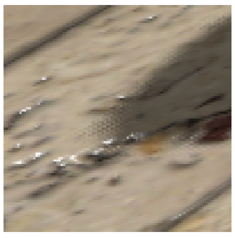
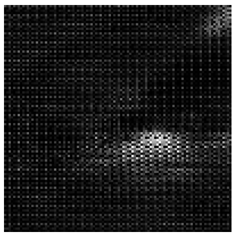
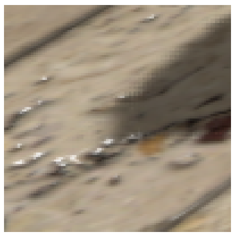
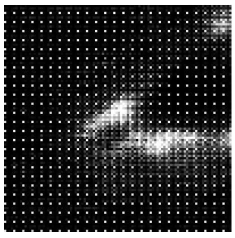
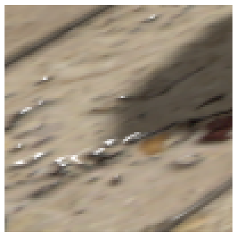
Crop	DLCTUS(4,1)	DLCTUS(4,1) Accumulation buffer	DLCTUS(2,1).	DLCTUS(2,1) Accumulation buffer	64spp
1					
2					
3					
4					
5					
6					

Table 17: Visualisation of the content of the accumulation buffer for DLCTUS(4,1) and DLCTUS(2,1). Bright white represents a 1, and black represents 0.

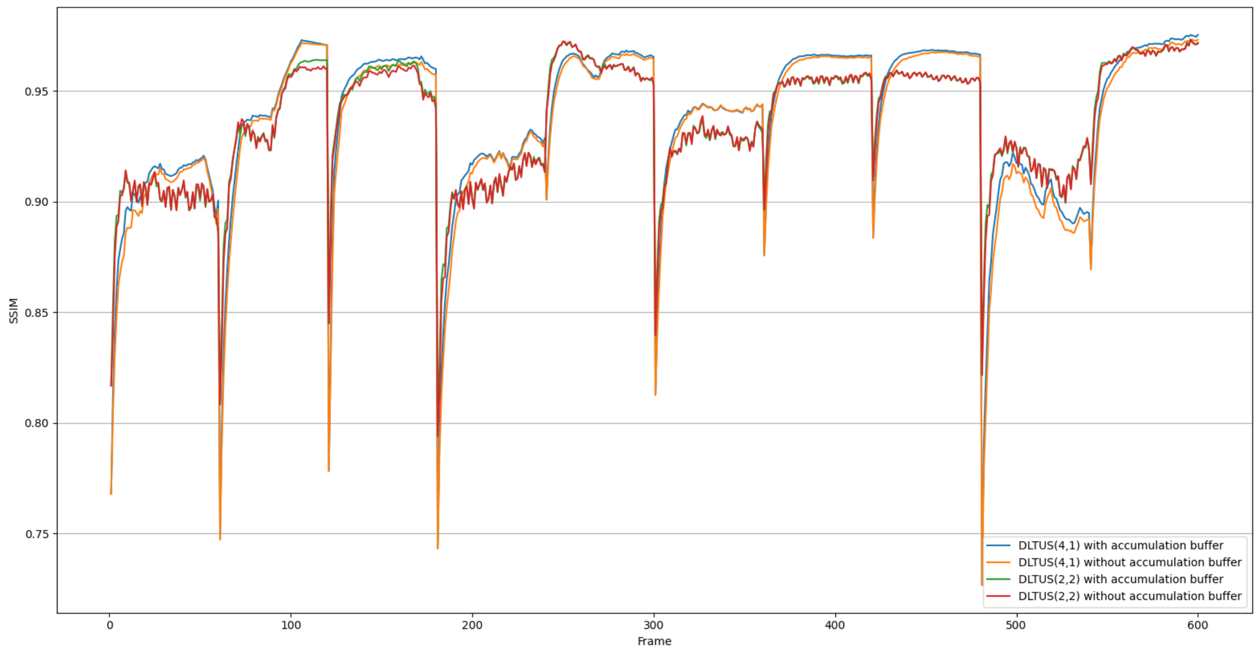


Figure 17: SSIM for the 10 videos in the testing dataset for DLCTUS(4,1) and DLCTUS(2,2) with and without an accumulation buffer.

6 Discussion

This section will discuss the results from section 5. First the visual quality of DLCTUS is analysed and compared to other methods. Then the quality benefits of using an accumulation buffer, training with a spatio-temporal loss function, and lowering the mipmap bias is examined. Finally, the run-time of the DirectX 12 implementation of DLCTUS is discussed.

6.1 The Visual Quality of the Architecture

Table 2 shows that the SSIM of the architecture is close but a little bit lower than Xiao et al. for 2x2 upsampling, but better for 4x4 upsampling. In Figure 9, Xiao et al. is better at around 5 frames into each video, but is quickly surpassed by DLCTUS after that. The fact that Xiao et al. is better at around 5 frames is not surprising, as the network uses a more powerful CNN aimed at getting the best quality out of using exactly 5 frames. However, DLCTUS surpassing Xiao et al. when more than 5 frames has passed indicates that DLCTUS successfully accumulates more than 5 frames when using 4x4 upsampling. This is different for 2x2 upsampling, where Figure 11 and Figure 13 shows that DLCTUS performs very similar to Xiao et al., which indicates that DLCTUS has difficulties with sample accumulation. An exception is towards the end of the second video, where the SSIM of DLCTUS is higher than Xiao et al.. This is the only part of the testing set without any camera motion, which suggests that DLCTUS is able to accumulate samples over a large amount of frames, but not during motion. One possible explanation for this is resampling blur, which is discussed further in 6.1.1.

Another important consideration for visual quality is artifacts. Artifacts can be rare, and only cover a small part of the screen, which will have little effect on the PSNR and SSIM of a frame, but can be very noticeable to a human observer. Common artifacts for temporal upsampling are ghosting, flickering, and jaggies, which are discussed in 6.1.2, 6.1.3 and 6.1.4 respectively.

6.1.1 Resampling Blur

Resampling blur is a result of errors from history reprojection accumulating over time, and it dampens high frequencies in the history buffer. The magnitude of the errors depend upon the reprojection method used. Table 13 shows that using bilinear interpolation significantly reduces the PSNR and SSIM of DLCTUS compared to using bicubic interpolation. This is a result of bilinear interpolation having a larger reprojection error than bicubic interpolation. Reprojection errors only occur when the motion vectors are non-zero, and Yang et al. [1] showed that the error is smaller when the interpolation position is closer to a pixel center. An indication of this distance in the testing set is shown in Figure 7, which illustrates the average distance to the closest pixel center for each motion vector in the testing dataset. The figure shows that motion vectors of the frames towards the end of the second video, and in the whole of the third video both have on average a shorter distance to pixel centers compared to other frames. This suggest that the reprojection error is small in these cases. The influence of a small reprojection error is visible in Figure 13 where the SSIM of DLCTUS is about equal to Xiao et al. for every case except the ones where the reprojection error is expected to be small. The difference in SSIM between Xiao et al. and DLCTUS when the reprojection errors are small compared to when the errors are large show how much reprojection errors impact the visual quality of DLCTUS This also show how decreasing this error can lead to a large increase in image quality. The fact that the SSIM of DLCTUS is about equal to Xiao et al. for 2x2-upsampling when the reprojection errors are large, suggest that the reprojection errors prevent DLCTUS from accumulating more than an effective amount of five samples during motion. However, this is different for 4x4-upsampling. Figure 9 shows that 4x4-upsampling performs a lot better than Xiao et al., even when the reprojection errors are large. This is likely due to the lower quality of the history buffer for 4x4-upsampling, which makes it less affected by reprojection errors.

The increased SSIM in Table 13 for Bicubic5 when performing 2x2 upsampling is surprising, as an approximation is expected to perform worse than the actual method. However, when testing the approximation in real-time a visual artifact appears, which is caused by divergent errors. Divergent errors are a result of using the bicubic convolution algorithm with $a = -0.75$ and not $a = -0.5$. Using the algorithm with $a = -0.5$ guarantees third order convergence for the errors, but no such guarantee exist of $a = -0.75$. This results in divergent errors in some rare cases, but this is usually not a big problem as DLCTUS will replace the errors with new samples. However, when using the approximation, divergent errors appear much more frequently, which suggest that the approximation is not well suited for general use. The divergent errors are a sign that using the bicubic convolution algorithm with $a = -0.75$ is not the best choice for temporal upsampling in general, but this is currently the only bicubic interpolation supported by PyTorch.

6.1.2 Ghosting

Ghosting artifacts happen when incorrect pixels in the history buffer are not correctly rectified. This is visible for TUS in crop 2 of Table 3, where the curtain behind the knight's helmet is brighter than it is supposed to be due to the color of the knight's helmet being clamped to the color AABB of the curtain. Ghosting artifacts usually disappear after a short amount of time, as new samples are accumulated and replaces the incorrect pixels. DLCTUS handles this situation much better than TUS, and successfully replaces the incorrect history with the jitter-aligned upsampled color of the input frame. Xiao et al. does an even better job with disocclusions as it uses the CNN to improve the visual quality of the disoccluded area.

Another instance of ghosting is visible only when running the implementation in real-time and happens when the camera is stationary and an object is moving on the screen. In some rare occasions the object can ghost, and the ghosting artifact does not fade away and disappear a short time later. This is a result of DLCTUS setting o_1 to zero for stationary pixels. One possibility for fixing this is to set a small lower limit to o_1, which will cause the ghosting artifacts to disappear after a short amount of time, but might reduce the overall image quality as small values of o_1 might be necessary to properly accumulate samples. Another possibility is to make sure videos that produce this artifact are included in the training set, and that training methods with a long frame sequence are used.

6.1.3 Flickering

Flickering is a temporal artifact where the output changes quickly and incorrectly between frames. Flickering is a problem for TUS in highly detailed areas, as the input is highly aliased. Aliasing leads to moiré patterns (2.3.1) and an example can be seen in Table 3 in crop 2 where the input has a moiré pattern consisting of light and dark green stripes on the curtain. The moiré patterns in the input frame changes between each frame due to the frame jittering, and the pattern disappears when enough samples are accumulated. This may result in a large difference between the input frame and history buffer, which results in a possibility for DLCTUS to wrongly perform history rectification and not sample accumulation. History rectification replaces the history buffer color with the jitter aligned input color, which contains moiré patterns, and the moiré patterns change between frames causing flickering. For DLCTUS this incorrect history rectification occurs in areas where the input is highly aliased. This is especially apparent when the camera is stationary as small patches of flickering can appear that gradually shrink and disappear. It happens very rarely and cannot be seen in the testing set, but is very visually disturbing when it happens.

Flickering also happens in newly disoccluded aliased areas, where samples are accumulated quickly causing a rapid increase in pixel quality that can be perceived as flickering. This is more eye-catching for 4x4-upsampling, where the increase in quality after disocclusion is larger. However, this flickering is not very visually disturbing as it looks a lot like shimmering, and only last for a very short time. A spatio-temporal loss function can be used to reduce flickering, and is discussed further in 6.3.

DLCTUS struggles with another temporal artifact related to aliased input, which is rapidly moving moiré patterns. Rapidly moving moiré patterns are generally not an issue, as the moiré patterns disappear when enough samples are accumulated. However, during motion, DLCTUS tends to rely more on the input than the history due to resampling blur. This causes the moiré patterns in the input to be visible during motion. The rapidly moving moiré patterns are a bigger problem for 2x2-upsampling than for 4x4-upsampling, since the quality of the input in 2x2-upsampling is higher than for 4x4-upsampling.

6.1.4 Jaggies

When the motion vectors are upsampled to the target resolution, errors are introduced in discontinuous areas. One example of such discontinuous areas are the object boundaries. Crop 2 and 3 in Table 3 show clear jaggies at the edge of the knights helmet, and at the edge of the curtain-beam, both being caused by inaccurate motion vectors. For TUS, motion vector dilation is used to prevent these jaggies. Motion vector dilation replaces each pixel's motion vector with the motion vector that is closest to the camera in a specified neighborhood. This can introduce additional small errors during reprojection, but it also ensures that pixels at the edge of objects are reprojected correctly. Table 15 shows that motion vector dilation successfully removes the jaggies at the curtain-beam, but not at the edge of the knight's helmet. This is because the network performs history rejection on the background behind the knight's helmet, and overshoots, causing the edge to be rejected as well. This might be a side effect of performing the convolutions in low resolution, which might reduce the precision of the networks history rejection. Table 14 shows how motion vector dilation effects the overall visual quality of the architecture. A small reduction of visual quality is observed for 2x2-upsampling. However, for 4x4-upsampling the reduction is significantly larger. One explanation for this might be the difference in size of the neighborhood for 2x2-upsampling and 4x4-upsampling. For 2x2-upsampling a 3x3-neighborhood corresponds to a 6x6 pixel region in the HR output, while 4x4-upsampling results in a 12x12 region. A larger region increases the chance of errors occurring.

Table 3, Table 4 and Table 5 show how much better Xiao et al. is at removing jaggies compared to DLCTUS. This is because Xiao et al. utilize a powerful reconstruction network, which is able to correct errors introduced by history reprojection. However, using such a powerful network heavily impacts the network's run-time, making this approach a tradeoff between run-time and jaggies.

6.2 Quality Gain from Accumulation Buffer

Using an accumulation buffer has a positive impact on the visual quality of the network. Table 16 shows an improvement of 0.26 dB in the average PSNR for 4x4-upsampling and an improvement of 0.09 dB for 2x2-upsampling. The reason for the improvement for 4x4-upsampling can be seen in Figure 17, where the absence of an accumulation buffer results in the network using longer time to reach its maximum, and a maximum that is lower than with an accumulation buffer. For 2x2-upsampling the benefit from an accumulation buffer is not as large as for 4x4-upsampling. This is visible in Figure 17, where the image quality is almost exactly equal with and without an accumulation buffer for most frames. An interesting exception is in cases where the resampling blur is small, such as the end of the second video, and the whole third video. One possible explanation is that DLCTUS is able to accumulate more samples when the resampling error is small, and the accumulation can be even more efficient with an accumulation buffer. This benefit is even more prominent when comparing the two approaches in real-time, where the absence of an accumulation buffer leads to very visible flickering when the camera is stationary. This might be because the accumulation buffer can provide information about the quality of each pixel, which makes the network less likely to erroneously replace a high quality pixel with a lower quality pixel, causing flickering.

Table 17 shows the content of the accumulation buffer for different crops. Crop 2, 3, and 6 all show examples of how the accumulation values are higher in areas where less samples are accumulated. The low sample counts are caused by disocclusions in the recent frames, and the low image quality of the disoccluded areas are visible in the images. Crop 2 also show a great example of how the image quality gradually increases for each frame after a disocclusion, as the white area in the crop gradually fades to darker colors. The table also shows an example of how DLCTUS is using the accumulation buffer in an unexpected way. This unexpected behavior is visible in crop 4, 5, and 6, where the camera has been stationary for a long time. A stationary camera give DLCTUS time to accumulate samples, and the accumulation values should be close to zero at this point. However, the crops unexpectedly show a repeatable 4x4-pattern consisting of one bright white value and the rest black. One possible explanation is that the network uses this pattern to detect motion. When the accumulation buffer is reprojected, the network can detect motion by checking if the pattern has moved or not. This can also explain why DLCTUS with an accumulation buffer perform better than DLCTUS without an accumulation buffer when the camera is stationary, as the accumulation buffer can be used to detect when the camera is stationary.

One disadvantage to using an accumulation buffer is the increase in run-time due to the extra reading and writing to and from the buffer. However, this only impacts the input preparation and output construction stage, as the neural network is required to use 8 input channels to execute on tensor cores. Table 8 shows that input preparation and output construction use a small amount of time compared to network execution, which implies that the reduction in run-time from not using an accumulation buffer is small.

6.3 Spatio-temporal Loss

Table 10 shows how the spatio-temporal loss function from 4.3.3 can be used to trade spatial loss for temporal loss, and compares it to a network trained with a purely spatial L1-loss function. The table shows that using a spatio-temporal loss function with θ equal to 0.1 or 0.5 reduces the spatial quality slightly when compared to L1-loss. For $\theta = 0.1$ the SSIM is reduced by 0.05 and for $\theta = 0.5$ the SSIM is reduced by 0.07. However, when θ is equal to 0.9 the spatial quality is reduced drastically with a reduction of 0.87 in the SSIM. At the same time the temporal loss is reduced slightly, where the reduction is larger for larger θ.

It is also important to consider how a spatio-temporal loss function impacts artifacts. The difference between a network trained with a spatial loss function and a network trained with a spatio-temporal loss function lies in their different handling of errors that are consistent between frames and errors that are inconsistent between frames. A spatio-temporal loss function prioritises to reduce pixels-wise errors that are inconsistent between frames, while it down prioritises errors that are consistent between frames. A spatial loss function on the other hand, weights the two losses equally. One type of error where the loss is consistent between frames is ghosting, as ghosting artifacts tend to be similar over multiple frames, and slowly vanish. The loss functions' impact on ghosting is visible in Table 11 in crop 3. For both $\theta = 0.5$ and $\theta = 0.9$, the ghosting trail over the curtain is more apparent than when using

L1-loss. An example of inconsistent errors are flickering, where the error is usually very different form one frame to the next. For a network trained with the spatial L1-loss function, flickering is rare, and can only be observed when running the networks in real-time. However, the difference in flickering when using a spatio-temporal loss function is large. For all the tested spatio-temporal loss functions, flickering disappears completely. Another temporal artifact that spatio-temporal could influence is quickly moving moiré patterns. Unfortunately, when inspecting the networks running in real-time, little difference can be observed when the network is trained with a spatio-temporal loss function.

Both 0.1 and 0.5 look like good choices for θ, as both of them remove flickering, and have little impact on the image quality. The biggest difference between the two choices is the increase in ghosting. Using $\theta = 0.5$ increases ghosting more than $\theta = 0.1$, which indicates that 0.1 might be the best choice for θ. Using $\theta = 0.9$ on the other hand has a significant impact on image quality, which suggest that this is a poor value for θ.

6.4 Best Mipmap Bias

Reducing the mipmap bias had a large influence on the image quality. This can be seen in Table 2 where DLCTUS(2,1) achieved a SSIM of 0.9625 and DLCTUS(2,2) achieved a SSIM of 0.9364. Directly comparing the two results are difficult as the target images are different. The target images from dataset 2 have a lower mipmap bias and as a result contains more detail and have a higher visual quality. This makes it possible for DLCTUS(2,2) to look better than DLCTUS(2,1), even though DLCTUS(2,1) has a higher average SSIM. This is visible when comparing crop 1 in Table 4 with crop 1 in Table 5, where DLCTUS(2,2) looks better than DLCTUS(2,1) but DLCTUS(2,1) is closer to the ground truth. The greatest downside to DLCTUS(2,2) is the increase in fast moving moiré patterns, which are caused by the additional aliasing due to the lower mipmap bias. This makes DLCTUS(2,2) more visually disturbing than DLCTUS(2,1), suggesting that using a higher mipmap bias is better until better techniques for preventing fast moving moiré patterns are introduced.

6.5 Training Analysis

Figure 16 shows that the network is slow to converge towards a maximum. For example, the PSNR for DLCTUS(2,2) is continuously increasing all the way up to 200 epochs. This indicates that the network can benefit from more epochs during training as maximum is not yet reached after 200. On the other hand this will increase the training time, which already is considerably long. One possible reason behind this slow convergence is the fact that the network is recurrent. A small improvement for a recurrent neural network can lead to a large effect on the overall quality. This small change can lead to a snowball effect where the network increases the quality of one frame, which is used as the input for the next iteration where the network once again increases the quality and so on.

Figure 16 also shows a large difference between training loss and validation loss. DLCTUS(4,1) and DLCTUS(2,1) have a training loss that is on average larger than the validation loss, while DLCTUS(2,2) has a training loss that is smaller than the validation loss. The difference between training loss and validation loss is used to detect overfitting during the training. A training loss smaller than the validation loss indicates that the network has overfitted to the training set, which might reduce the networks ability to generalize. However, for this to work, the training set must on average be very similar to the validation set. This is usually ensured by making the training set and validation set consist of random elements from a larger set. In this case, the random elements are videos, and the dataset only contains a total of 100 videos. This amount is too small to ensure that the training set and validation set are similar enough for their averages to be the same.

6.6 DirectX Implementation Run-time Performance

The network had a performance goal of running in conjunction with a graphics renderer in real-time. Table 8 shows that the DirectX 12 implementation uses a total of 2.146 ms for 4x4 upsampling. This is a small fraction of the 16.67 ms available when rendering at 60 frames per second, leaving 14.524 ms for rendering. This runtime is low enough to satisfy the goal of running the network and rendering in conjunction.

Table 8 also shows that the two optimizations from 4.4.4 have a positive impact on the performance. This impact is larger than initially expected, as duration of both input preparation and output construction is reduced significantly, while only the memory access pattern is changed. The same holds for the downsampling, as the 4x4 convolution with stride 4 is computationally equivalent with the 1x1 convolution with stride 1. A possible explanation for this is that the new memory access pattern has better caching properties, which reduces cache misses during execution and saves time on memory fetches.

A large portion of the implementations' execution is spent on the residual net. The residual net consists of four residual blocks, each of them using 369 µs. Adding these up, results in a total of 1.476 ms for the residual net, which is 68.8% of the total run-time. This shows that optimizing the residual block can lead to a large performance boost. An interesting observation about the run-time of the residual block shown in Table 6 is that the addition layer uses 120 µs This is very close to the time used by the 3x3 convolution, which is 122 µs. One might expect the addition layer to be significantly faster than a convolution layer, as a convolution requires significantly more mathematical computations than an addition. However, the tensor addition operation in the residual connections are limited by GPU memory bandwidth, while the convolution layer is limited by mathematical operation bandwidth, which significantly slows down the addition layer. One possibility to reduce the run-time of the addition layer is to fuse it with the preceding convolution layer, which would remove the need for one of the read operations and the write operation of the addition layer. Another possibility is to use another network architecture that does not rely on residual connections, arguing that the time spent on the addition layer is better spent on something else.

Directly comparing the run-time of DLCTUS to Xiao et al. and DLSS 2.0 is difficult as the run-times are tested on different GPUs. For 4x4-upsampling the different run-times are: 2.15 ms for DLCTUS on a NVIDIA RTX 2060 GPU, 24.4 ms for Xiao et al. on a Titan V GPU, and 0.736 ms for DLSS 2.0 on a NVIDIA RTX 2060 Super GPU. When looking at the specifications of the GPUs [32][33][34], it is clear that the Titan V can be expected to be significantly faster than the RTX 2060, while the RTX 2060 Super can be expected to be only slightly faster than the RTX 2060. With this information is it possible to state that DLCTUS is at least an order of magnitude faster than Xiao et al., and that DLSS 2.0 is no more than 2.9 times faster than DLCTUS.

7 Conclusion

This thesis introduced a novel architecture for deep learning temporal upsampling. The architecture's performance was measured in SSIM and compared to state-of-the-art methods. The comparison showed that the architecture performed slightly worse for 2x2-upsampling, but significantly better for 4x4-upsampling. The architecture's run-time was measured by creating an DirectX 12 implementation, and was found to be an order of magnitude faster than state-of-the-art. However, the architecture suffers from artifacts such as jagged edges and rapidly moving moiré patterns, which can impact the perceived visual quality. The thesis also formulated a spatio-temoporal loss function, which was shown to significantly reduce flickering. In addition, an application was made for the purpose of generating training data, and a method for training the network was presented.

7.1 Research Questions

This section attempts to answer the research questions from 1.1, based on the findings of this thesis.

- **RQ1**: *What are the main challenges of training and applying a recurrent neural network for TUS?*

  TUS benefits from reusing information from iterations as far back as is possible. This makes it necessary to use a long sequence length when training the RNN with BPTT. This long sequence length proved to be a challenge because it increased both training time and memory usage. The main cause of the training time increase was the large amount of data that had to be loaded at each iteration of the training. This training time was reduced by storing the data in the HDF5 format, but was still large. The large memory usage limited network size, batch size, and crop size during training. The small network size was not a big problem, as a small network was already necessary for fast inference. However, reducing the batch size and crop size can influence the training performance, and finding a good balance between sequence length, batch size, and crop size proved challenging.

  Another challenge was ensuring that the RNN generalized to sequences longer than the training sequences. The two main steps taken to increase generalizability were to use long sequences for training and reducing the networks influence over the output.

- **RQ2**: *How can methods originally used to enhance TUS fit in a neural network approach, and how do they affect the visual quality?*

  Two methods from TUS investigated in this thesis were motion vector dilation and the use of an accumulation buffer.

  Motion vector dilation was shown to work well for 2x2-upsampling, where it resulted in no significant reduction in overall image quality, and a reduction of jaggies. However, it was not able to remove jaggies on trailing edges during motion. Motion vector dilation was less effective for 4x4-upsampling, where it resulted in a larger reduction in overall image quality. This might be a sign that different methods have to be explored to remove jaggies for large upsampling factors.

The accumulation buffer was shown to work well in a neural network approach when the network has influence over what to store in it. Using an accumulation buffer increased the image quality for both 2x2-upsampling and 4x4-upsampling. The greatest benefit was the increased stability it provided for stationary pixels. In addition, it increased the quality during movement for 4x4-upsampling, but not for 2x2-upsampling.

- **RQ3**: *How can a spatio-temporal loss function be formulated, and how does it impact the visual quality and temporal stability of the network?*

  A spatio-temporal loss function was formulated based on the spatio-temporal loss function from Salvi [25]. The formulation used mean absolute error over mean square error, and was parameterized by a parameter θ. This parameter could be used to trade spatial error for temporal error, where $\theta = 0$ only focused on spatial error and $\theta = 1$ only focused on temporal error. It was shown that a large θ of 0.9 significantly reduced the average image quality, and thus is not a good value. However, both $\theta = 0.1$ and $\theta = 0.5$ were shown to only reduce the image quality by a small amount. Both 0.1 and 0.5 significantly reduced flickering, but also increased ghosting. The increase in ghosting were larger for 0.5 than for 0.1, which indicated that 0.1 is a good value for θ.

- **RQ4**: *What are the difficulties and limitations behind creating a neural network that runs in real-time on a modern GPU?*

  Even when using tensor cores, high-resolution convolution operations were found to be too time consuming to be useable in a real-time neural network. This limited the convolutions to low-resolution tensors, which were faster, but reduced the networks ability to rectify and accumulate samples. However, a low-resolution network was found to be sufficient when the complexity of the task performed by the network was reduced.

7.2 Further Work

This subsection proposes different ways to improve upon the work of this thesis. These propositions were not implemented in the thesis due to time constraints and some being outside the scope of the thesis.

7.2.1 Architecture

The proposed architecture is inspired by temporal upsampling and state-of-the-art image upsampling neural networks. Many different architectures for deep learning temporal upsampling are possible by combining the two research areas in different ways. One possible improvement is to give the neural network more control over the output, either by letting it directly produce it, which was done by Salvi [25], or by other means. This enables the network to do error correction, which can remove different errors that arise during execution, but it also increases the complexity of the task the network performs, which can lead to poor performance for small networks. Another improvement is testing out other network architectures, which can result in faster inference and better results.

The architecture can also be improved by optimizing the inputs. One possible input the architecture could benefit from is the distance to the closest pixel center for each motion vector. This can be calculated from the motion vectors, and gives the network information about how large it can expect the resampling error to be, and it is possible that the network can perform better with this information.

7.2.2 Reprojection Method

Reprojection errors heavily impacts the visual quality during movement, especially for 2x2 upsampling. Bicubic interpolation is commonly used because of its efficiency and small interpolation error. However, the fact that this small interpolation error is a limiting factor, might indicate that it is time to move on to better reprojection methods. One such method is Sacht-Nehab [35]. Sacht-Nehab has an interpolation error that is significantly smaller than bicubic interpolation, but comes at a cost of being more computationally complex. However, with the neural network dominating the run-time of the implementation, an increase in reprojection time will have a smaller effect on the total run-time. This is different from traditional temporal upsampling, where reprojection takes up a large portion of the total run-time. One difficulty with using Sacht-Nehab for deep learning temporal upsampling is implementing it in a machine learning framework such as PyTorch. This is necessary for training the network, and an inefficient implementation can lead to a large increase in training time.

7.2.3 Network Training

The training of the network can be improved in two different ways: improving the dataset, and improving the training methodology. The dataset can be improved by including more scenes. This will make it possible to test how well the network generalizes to unseen input, as well as improve generalization by increasing the variety in the training set. Further research can also be done on the optimal video length in the dataset. This thesis has consistently used a video length of 60 frames, and back-propagated through 30 frames at a time. Increasing the amount of frames used for back-propagating is difficult as it is limited by GPU memory. However, by using TBPTT, it is possible to better learn longer sequences because the next training iteration is initialized with the output of the previous iteration.

7.2.4 Optimizations

Optimizing the implementation of the architecture is important. Reducing the run-time can either open up for higher quality rendering, or a more complex network, both resulting in higher visual quality. An optimization that can be researched further is the usage of 8-bit integer convolutions. This optimization can significantly speed up the network, as 8-bit integer operations are faster than 16-bit floating point operations. However, it can lead to issues due to the reduced precision, but how much this impacts the visual quality is difficult to say without further research.

Bibliography

[1] L. Yang, D. Nehab, P. V. Sander, P. Sitthi-amorn, J. Lawrence and H. Hoppe, 'Amortized supersampling', *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, pp. 1–12, 2009.

[2] E. Games. (2021). 'Unreal engine 4, taau', [Online]. Available: https://docs.unrealengine.com/en-US/RenderingAndGraphics/ScreenPercentage/index.html (visited on 5th May 2021).

[3] C. Dong, C. C. Loy, K. He and X. Tang, 'Image super-resolution using deep convolutional networks', *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 295–307, 2015.

[4] NVIDIA. (2021). 'Discover how tensor cores accelerate your mixed precision models', [Online]. Available: https://developer.nvidia.com/tensor-cores (visited on 29th May 2021).

[5] A. Edelsten, P. Jukarainen and A. Patney, 'Truly next-gen: Adding deep learning to games & graphics', in *NVIDIA Sponsored Sessions (Game Developers Conference)*, 2019.

[6] NVIDIA. (2020). 'Nvidia dlss 2.0: A big leap in ai rendering', [Online]. Available: https://www.nvidia.com/en-gb/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/ (visited on 6th Dec. 2020).

[7] K. Fukushima, 'Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron', *IEICE Technical Report, A*, vol. 62, no. 10, pp. 658–665, 1979.

[8] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[9] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, 'Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1874–1883.

[10] A. L. Maas, A. Y. Hannun and A. Y. Ng, 'Rectifier nonlinearities improve neural network acoustic models', in *Proc. icml*, Citeseer, vol. 30, 2013, p. 3.

[11] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[12] D. P. Kingma and J. Ba, 'Adam: A method for stochastic optimization', *arXiv preprint arXiv:1412.6980*, 2014.

[13] R. J. Williams and J. Peng, 'An efficient gradient-based algorithm for on-line training of recurrent network trajectories', *Neural computation*, vol. 2, no. 4, pp. 490–501, 1990.

[14] Z. Wang, J. Chen and S. C. Hoi, 'Deep learning for image super-resolution: A survey', *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[15] R. Keys, 'Cubic convolution interpolation for digital image processing', *IEEE transactions on acoustics, speech, and signal processing*, vol. 29, no. 6, pp. 1153–1160, 1981.

[16] B. Lim, S. Son, H. Kim, S. Nah and K. Mu Lee, 'Enhanced deep residual networks for single image super-resolution', in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 136–144.

[17] T. Vu, C. Van Nguyen, T. X. Pham, T. M. Luu and C. D. Yoo, 'Fast and efficient image quality enhancement via desubpixel convolutional neural networks', in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018.

[18] J. Johnson, A. Alahi and L. Fei-Fei, 'Perceptual losses for real-time style transfer and super-resolution', in *European conference on computer vision*, Springer, 2016, pp. 694–711.

[19] E. T. Swensen, *A study of anti-aliasing and upsampling techniques for deferred rendering and real-time ray-tracing*, 2020.

[20] L. Williams, 'Pyramidal parametrics', in *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, 1983, pp. 1–11.

[21] L. Yang, S. Liu and M. Salvi, 'A survey of temporal antialiasing techniques', *STAR*, vol. 39, no. 2, 2020.

[22] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk and J. R. Isidoro, 'Accelerating real-time shading with reverse reprojection caching', in *Graphics hardware*, vol. 41, 2007, pp. 61–62.

[23] B. Karis, 'High-quality temporal supersampling', *Advances in Real-Time Rendering in Games, SIGGRAPH Courses*, 2014.

[24] R. Herzog, E. Eisemann, K. Myszkowski and H.-P. Seidel, 'Spatio-temporal upsampling on the gpu', in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010, pp. 91–98.

[25] M. Salvi, 'Deep learning: The future of real-time rendering', *ACM SIGGRAPH Courses: Open Problems in Real-Time Rendering*, vol. 12, 2017.

[26] L. Xiao, S. Nouri, M. Chapman, A. Fix, D. Lanman and A. Kaplanyan, 'Neural supersampling for real-time rendering', *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 142–1, 2020.

[27] E. Liu, 'Dlss 2.0 – image reconstruction for real-time rendering with deep learning', in *GPU Technology Conference*, NVIDIA, 2020.

[28] E. T. Swensen. (2021). 'Thesis' repository', [Online]. Available: https://github.com/EilifTS/Anti-Aliasing (visited on 24th May 2021).

[29] PyTorch. (2021). 'Pytorch', [Online]. Available: https://pytorch.org/ (visited on 3rd Jun. 2021).

[30] NVIDIA. (2021). 'Convolutional layers user guide', [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html (visited on 24th May 2021).

[31] Microsoft. (2020). 'Pix on windows', [Online]. Available: https://devblogs.microsoft.com/pix/ (visited on 6th Dec. 2020).

[32] NVIDIA. (2021). ‘Geforce rtx 2060’, [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060/ (visited on 20th Jun. 2021).

[33] ——, (2021). ‘Nvidia titan v’, [Online]. Available: https://www.nvidia.com/en-us/titan/titan-v/#specs (visited on 20th Jun. 2021).

[34] ——, (2021). ‘Geforce rtx 2060 super’, [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060-super/ (visited on 20th Jun. 2021).

[35] L. Sacht and D. Nehab, ‘Optimized quasi-interpolators for image reconstruction’, *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5249–5259, 2015.

Appendix

A Code Examples

A.1 PyTorch Model for DLCTUS

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class ZeroUpsampling(nn.Module):
    def __init__(self, factor, channels):
        super(ZeroUpsampling, self).__init__()
        self.factor = factor
        self.channels = channels
        self.w = torch.zeros(size=(self.factor, self.factor))
        self.w[0, 0] = 1
        self.w = self.w.expand(channels, 1, self.factor, self.factor).cuda()

    def forward(self, x):
        return F.conv_transpose2d(x, self.w, stride=self.factor,
        ↪  groups=self.channels)

def JitterAlign(img, jitter, factor, mode='constant'):
    N, C, H, W = img.shape
    jitter_index = torch.floor(jitter*factor).int()
    img = F.pad(img, (factor,factor,factor,factor), mode=mode)
    out = []
    for b in range(N):
        i, j = jitter_index[b,1], jitter_index[b,0]
        out.append(img[b,:,factor-i:-factor-i,factor-j:-factor-j])
    img = torch.stack(out)
    return img

def JitterAlignedInterpolation(img, jitter, factor, mode='bilinear'):
    N, C, H, W = img.shape
    theta = []
    for i in range(N):
        theta.append(torch.tensor(
            [   [1,0,2.0*(0.5-jitter[i,0]) / W],
                [0,1,2.0*(0.5-jitter[i,1]) / H]], device='cuda'))
    theta = torch.stack(theta)
    grid = F.affine_grid(theta, (N,C,H*factor,W*factor),
    ↪  align_corners=False).float()

    return F.grid_sample(img, grid, mode=mode, align_corners=False,
    ↪  padding_mode='border')

def IdentityGrid(shape):
    N, H, W, C = shape
    theta = torch.tensor(
        [   [1.0,0.0,0.0],
            [0.0,1.0,0.0]], device='cuda')
    theta = theta.unsqueeze(0).expand(N, -1, -1)
    grid = F.affine_grid(theta, (N,C,H,W), align_corners=False).float()
    return grid

def DepthToLinear(depth):
    far, near = 100.0, 0.1
    depth = near * far / (far - depth * (far - near))
    return (depth - near) / (far - near)

class MasterNet(nn.Module):
    def __init__(self, factor):
        super(MasterNet, self).__init__()
        self.factor = factor
        self.history_channels = 4

        self.zero_up = ZeroUpsampling(factor, 4)

        self.down = nn.Sequential(
            nn.Conv2d(8, 32, 4, stride=4),
            )
        self.cnn1 = nn.Sequential(
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            )
        self.cnn2 = nn.Sequential(
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            )
        self.cnn3 = nn.Sequential(
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            )
        self.cnn4 = nn.Sequential(
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, stride=1, padding=1, padding_mode='replicate'),
            )

        self.up = nn.Sequential(
            nn.PixelShuffle(4)
            )

    def forward(self, frame, depth, mv, jitter, history):
        mini_batch, channels, height, width = frame.shape

        # Linearizing depth
        depth = DepthToLinear(depth)

        # Getting frame info
        frame_ones = torch.cat((frame, torch.ones(size=( mini_batch, 1, height,
        ↪  width), device='cuda')), dim=1)

        # Frame upsampling
        frame_bilinear = JitterAlignedInterpolation(frame_ones, jitter,
        ↪  self.factor, mode='bilinear')
        frame = torch.cat((frame, depth), dim=1)
        frame = self.zero_up(frame)
        frame = JitterAlign(frame, jitter, self.factor)

        if(history == None): # First frame is handled by its own
            history = frame_bilinear
        else:
            # Upscaling motion vector
            mv -= IdentityGrid(mv.shape)
            mv = torch.movedim(mv, 3, 1)
            mv = F.interpolate(mv, scale_factor=self.factor, mode='bilinear',
            ↪  align_corners=False)
            mv = torch.movedim(mv, 1, 3)
            mv += IdentityGrid(mv.shape)

            # History reprojection
            history = F.grid_sample(history, mv, mode='bicubic',
            ↪  align_corners=False, padding_mode='border')

            ## Special case for padding
            mask = torch.logical_or(torch.greater(mv, 1.0), torch.less(mv,
            ↪  -1.0))
            mask = torch.logical_or(mask[:,:,:,0], mask[:,:,:,1])
            mask = mask.unsqueeze(1)
            mask = mask.expand(mini_batch, 4, height*self.factor,
            ↪  width*self.factor)

            history[mask] = frame_bilinear[mask]

        # Execute network
        small_input = torch.cat((frame, history), dim=1)
        small_input = self.down(small_input)
        small_input = small_input + self.cnn1(small_input)
        small_input = small_input + self.cnn2(small_input)
        small_input = small_input + self.cnn3(small_input)
        small_input = small_input + self.cnn4(small_input)
        residual = self.up(small_input)

        # Output Construction
        residual = torch.clamp(residual, 0.0, 1.0)

        # Lerp
        alpha = residual[:,0:1,:,:]
        history = history*(1.0 - alpha) + frame_bilinear * alpha

        # Mul
        history_start = history[:,0:3,:,:]
        history_end = history[:,3:4,:,:]*residual[:,1:2,:,:]
        history = torch.cat((history_start, history_end), dim=1)

        history = torch.clamp(history, 0.0, 1.0)
        # Reconstruction
        return history[:,:3,:,:], history
```

A.2 HLSL Code for Input Preparation

```
#ifndef UPSAMPLE_FACTOR
#define UPSAMPLE_FACTOR 4
#endif

#ifndef OPTIM
#define OPTIM 2
#endif

Texture2D                        input_texture            : register(t0);
Texture2D<half4>        history_buffer               : register(t1);
Texture2D<float2>        motion_vectors               : register(t2);
Texture2D<float>        depth_buffer                : register(t3);
SamplerState                linear_clamp                : register(s0);
SamplerState                point_clamp                : register(s1);

cbuffer constants : register(b0)
{
    float2 window_size;
    float2 rec_window_size;
    float2 current_jitter;
}

RWBuffer<half> out_tensor   : register(u0);

half4 catmullSample(float2 uv)
{
    half4 c = history_buffer.SampleLevel(linear_clamp, uv, 0);
    return c;
}

half4 catmullRom(float2 uv)
{
    float2 position = uv * window_size;
    float2 center_position = floor(position - 0.5) + 0.5;
    float2 f = position - center_position;
    float2 f2 = f * f;
    float2 f3 = f2 * f;

    half2 w0 = half2(0.25 * (-3.0 * f3 + 6.0 * f2 - 3.0 * f));
    half2 w1 = half2(0.25 * (5.0 * f3 - 9.0 * f2 + 4.0));
    half2 w2 = half2(0.25 * (-5.0 * f3 + 6.0 * f2 + 3.0 * f));
    half2 w3 = half2(0.25 * (3.0 * f3 - 3.0 * f2));
    //float2 w2 = 1.0 - w0 - w1 - w3

    half2 w12 = w1 + w2;
    float2 tc12 = rec_window_size * (center_position + float2(w2) /
    ↪ float2(w12));
    half4 center_color = catmullSample(tc12);

    float2 tc0 = rec_window_size * (center_position - 1.0);
    float2 tc3 = rec_window_size * (center_position + 2.0);
    half4 color =
        catmullSample(float2(tc0.x, tc0.y))  * (w0.x  * w0.y) +
        catmullSample(float2(tc3.x, tc0.y))  * (w3.x  * w0.y) +
        catmullSample(float2(tc12.x, tc0.y)) * (w12.x * w0.y) +
        catmullSample(float2(tc0.x, tc12.y)) * (w0.x  * w12.y) +
        center_color * (w12.x * w12.y) +
        catmullSample(float2(tc3.x, tc12.y)) * (w3.x  * w12.y) +
        catmullSample(float2(tc12.x, tc3.y)) * (w12.x * w3.y) +
        catmullSample(float2(tc3.x, tc3.y))  * (w3.x  * w3.y) +
        catmullSample(float2(tc0.x, tc3.y))  * (w0.x  * w3.y);
    return color;
}

float linear_depth(float depth)
{
    float far = 100.0;
    float near = 0.1;
    depth = near * far / (far - depth * (far - near));
    return (depth - near) / (far - near);
}

uint get_pixel_shuffle_index(uint x, uint y, uint r, uint W, uint C)
{
    uint ym = y % r;
    uint xm = x % r;

    return (y - ym) * W * C + (x - xm) * r * C + r * ym + xm;
}

[numthreads(8, 8, 1)]
void CS(uint3 block_id : SV_GroupID, uint3 thread_id : SV_GroupThreadID)
{
    uint2 window_size_int = (uint2)window_size;
    uint2 pixel_pos = uint2(block_id.x * 8 + thread_id.x, block_id.y * 8 +
    ↪  thread_id.y);

    if (pixel_pos.x < window_size_int.x && pixel_pos.y < window_size_int.y)
    {
        uint2 lr_pixel_pos = pixel_pos / UPSAMPLE_FACTOR; // Position of pixel
        ↪  in low res image
        float2 jitter_offset = current_jitter; // Pixel offset of jitter in low
        ↪  res image

        // Zero upsample rgbd
        float2 lr_jitter_pos = (float2)lr_pixel_pos + jitter_offset;
        uint2 hr_jitter_pos_int = (uint2)(lr_jitter_pos * UPSAMPLE_FACTOR);
        float beta = 0.0;
        if (hr_jitter_pos_int.x == pixel_pos.x && hr_jitter_pos_int.y ==
        ↪  pixel_pos.y) beta = 1.0;
        float4 zero_up_rgbd = float4(0.0, 0.0, 0.0, 0.0);
        zero_up_rgbd.rgb = input_texture[lr_pixel_pos].rgb;
        zero_up_rgbd.a = linear_depth(depth_buffer[lr_pixel_pos]);
        zero_up_rgbd = zero_up_rgbd * beta;

        // JAU rgbd
        float2 hr_jitter_pos = (float2(0.5, 0.5) + (float2)pixel_pos) +
        ↪  (float2(0.5, 0.5) - jitter_offset) * UPSAMPLE_FACTOR;
        float2 hr_jitter_uv = hr_jitter_pos * rec_window_size;
        float4 jau_rgbd = float4(0.0, 0.0, 0.0, 0.0);
        jau_rgbd.rgb = input_texture.SampleLevel(linear_clamp, hr_jitter_uv,
        ↪  0).rgb;
        jau_rgbd.a = 1.0;

        // reproject history
        float2 hr_pixel_uv = (float2(0.5, 0.5) + (float2)pixel_pos) *
        ↪  rec_window_size;
        float2 motion_vector = motion_vectors.SampleLevel(linear_clamp,
        ↪  hr_pixel_uv, 0);
        float2 prev_frame_uv = hr_pixel_uv + motion_vector;

        half4 history = half4(0.0, 0.0, 0.0, 0.0);
        if (prev_frame_uv.x > 0.0 && prev_frame_uv.x <= 1.0 && prev_frame_uv.y >
        ↪  0.0 && prev_frame_uv.y <= 1.0) // Check if uv is inside history
        ↪  buffer
            history = catmullRom(prev_frame_uv);
        else
            history = half4(jau_rgbd);


        // Fill output tensor
#if OPTIM == 2
        uint index0 = get_pixel_shuffle_index(pixel_pos.x, pixel_pos.y, 4,
        ↪  window_size_int.x, 8);
        uint index1 = index0 + 4 * 4 * 1;
        uint index2 = index0 + 4 * 4 * 2;
        uint index3 = index0 + 4 * 4 * 3;
        uint index4 = index0 + 4 * 4 * 4;
        uint index5 = index0 + 4 * 4 * 5;
        uint index6 = index0 + 4 * 4 * 6;
        uint index7 = index0 + 4 * 4 * 7;
#else
        uint index = window_size_int.x * pixel_pos.y + pixel_pos.x;
        uint index0 = 8 * index + 0;
        uint index1 = 8 * index + 1;
        uint index2 = 8 * index + 2;
        uint index3 = 8 * index + 3;
        uint index4 = 8 * index + 4;
        uint index5 = 8 * index + 5;
        uint index6 = 8 * index + 6;
        uint index7 = 8 * index + 7;
#endif
        out_tensor[index0] = half(zero_up_rgbd.r);
        out_tensor[index1] = half(zero_up_rgbd.g);
        out_tensor[index2] = half(zero_up_rgbd.b);
        out_tensor[index3] = half(zero_up_rgbd.a);
        out_tensor[index4] = history.r;
        out_tensor[index5] = history.g;
        out_tensor[index6] = history.b;
        out_tensor[index7] = history.a;
    }
}
```

A.3 HLSL Code for Output Construction

```
Buffer<half> history_buffer : register(t0);
Buffer<half> cnn_res : register(t1);

Texture2D input_texture        : register(t2);
Texture2D<float> depth_buffer        : register(t3);
SamplerState linear_clamp : register(s0);

#ifndef UPSAMPLE_FACTOR
#define UPSAMPLE_FACTOR 4
#endif

#ifndef OPTIM
#define OPTIM 2
#endif

// Use this as index + c+r*r to get actual index
uint get_pixel_shuffle_index(uint x, uint y, uint r, uint W, uint C)
{
        uint ym = y % r;
        uint xm = x % r;

        return (y - ym) * W * C + (x - xm) * r * C + r * ym + xm;
}

cbuffer constants : register(b0)
{
        float2 window_size;
        float2 rec_window_size;
        float2 current_jitter;
}

struct PSInput
{
        float4 position : SV_POSITION;
        float2 clip_position : TEXCOORD0;
        float2 uv : TEXCOORD1;
};

half4 PS(PSInput input) : SV_TARGET
{
        uint2 hr_pixel_pos = (uint2)input.position.xy; // Position in pixels of
        ↪  pixel in high res image
        uint2 lr_pixel_pos = hr_pixel_pos / UPSAMPLE_FACTOR; // Position of
        ↪  pixel in low res image
        float2 jitter_offset = current_jitter; // Pixel offset of jitter in low
        ↪  res image

        // JAU input
        float2 hr_jitter_pos = input.position.xy + (float2(0.5, 0.5) -
        ↪  jitter_offset) * UPSAMPLE_FACTOR;
        float2 hr_jitter_uv = hr_jitter_pos * rec_window_size;
        half4 jau_rgbd = half4(0.0, 0.0, 0.0, 0.0);
        jau_rgbd.rgb = half3(input_texture.Sample(linear_clamp,
        ↪  hr_jitter_uv).rgb);
        jau_rgbd.a = 1.0;

        // Load history
        uint2 window_size_int = (uint2)window_size;

#if OPTIM == 2
        uint index = get_pixel_shuffle_index(hr_pixel_pos.x, hr_pixel_pos.y, 4,
        ↪  window_size.x, 8);
        uint index0 = index + 4*4 * 4;
        uint index1 = index + 4*4 * 5;
        uint index2 = index + 4*4 * 6;
        uint index3 = index + 4*4 * 7;
#else
        uint index = hr_pixel_pos.y * window_size.x + hr_pixel_pos.x;
        uint index0 = index * 8 + 4;
        uint index1 = index * 8 + 5;
        uint index2 = index * 8 + 6;
        uint index3 = index * 8 + 7;
#endif

        half4 history = half4(0.0, 0.0, 0.0, 0.0);
        history.r = history_buffer[index0];
        history.g = history_buffer[index1];
        history.b = history_buffer[index2];
        history.a = history_buffer[index3];

        // Pixel shuffle
#if OPTIM == 0
        index0 = 2 * index + 0;
        index1 = 2 * index + 1;
#else
        index0 = get_pixel_shuffle_index(hr_pixel_pos.x, hr_pixel_pos.y, 4,
        ↪  window_size.x, 2);
        index1 = index0 + 4 * 4;
#endif

        // Load alpha and depth_res
        half alpha = clamp(cnn_res[index0], 0.0, 1.0);
        half depth_res = clamp(cnn_res[index1], 0.0, 1.0);

        // Combine
        half4 output = history * (1.0 - alpha) + jau_rgbd * alpha;
        output.a *= depth_res;

        // Return
        return saturate(output);
}
```

B Application Manual

1 Anti-Aliasing

Anti-Aliasing is a Windows 10 desktop application with implementations of several different anti-aliasing methods.

2 Installation

This project consists of three sub projects, two C++ projects and one Python project. The C++ projects require Micosoft Visual Studio 2019 with the ”Desktop development with C++” option and the latest Windows SDK. In addition is the DirectML NuGet package required.

To run the python project, the following python packages are required: PyTorch, torchvision, CV2, Matplotlib, Numpy, h5py and PIL.

3 Hardware requirement

Certain hardware is necessary to run the neural network efficiently. A GPU which supports native 16bit operations is necessary to avoid a bug where errors from 32bit to 16bit conversions accumulate, but the network is able to run without. A GPU with tensor cores is necessary to run the network efficiently. A GPU using the NHWC format is necessary to run the network efficiently. From NVIDIA should a GTX 1060 or newer work.

4 Description

`Anti-Aliasing/app.cpp` : Main file for running the real-time application

`DatasetGenetator/DatasetGenerator.cpp` : Main file for dataset generation

`ELib/graphics/` : Everything related to DirectX 12

`ELib/math/` : Some helper classes for math

`ELib/window/` : Everything related to Windows

`ELib/misc/` : Helper functions and classes for various purposes

`Rendering/deep_learning` : Everything related to DirectML and the execution of DLCTUS

`Rendering/deferred_rendering` : Classes used for deferred rendering

`Rendering/ray_tracer` : Class for using ray tracing

`Rendering/models` : .obj files for meshes used

`Rendering/textures` : Texture files for mesh textures

`Rendering/aa` : Classes for applying anti-aliasing

`Rendering/shaders` : Contains all shaders used in this project

`Network` : Contains python code related to the project

5 Usage

Keyboard and Mouse Input	Action
WASD	Movement control
Mouse movement	Rotation
1	Set Anti-aliasing mode off
2	Set Anti-aliasing mode to FXAA
3	Set Anti-aliasing mode to TAA
4	Set Anti-aliasing mode to SSAA
5	Toggle freeze frame (The right arrow to skip frames)
6	Toggle ray tracing
7	Toggle DLCTUS

When in TAA mode:

Keyboard and Mouse Input	Action
R	Toggle between bilinear and bicubic interpolation
T	Toggle history rectification
Y	Toggle between YCoCg space and RGB space
U	Toggle between history clipping and clamping

Temporal upsampling is used by changing the constants `upsample_numerator` and `upsample_denominator` in `Anti-Aliasing/app.cpp` at lines 19 and 20

6 Deep Learning Pipeline

The deep learning pipeline has several steps: 1. Camera Motion Capture 2. Dataset Generation 3. Dataset Conversion 4. Network Training 5. Network Conversion 6. Network Loading

All steps are not necessary to run the network, as several pretrained networks are included in the project.

6.1 Camera Motion Capture

Camera motion capure is done by starting the main application `Anti-Aliasing`. Then pressing the "Q"-button will record a 60-frame sequence video. The video is stored as a .txt file in `DatasetGenerator/camera_positions/`, and the file contains world time, camera position and rotation. 100 prerecorded .txt files are allready stored in the folder.

6.2 Dataset Generation

Dataset generation is done by exectuing the `DatasetGenerator` project. The project will iterate over all video files in `DatasetGenerator/camera_positions/` and generate input color, motion vector and depth buffers and save them as .png files to `DatasetGenerator/data`. In addition is the frames jitter offset saved to a .txt file. A number of control variables are defined at the start of `DatasetGenerator/DataGenerator.cpp` which control the generation process. `super_sample_options` define how many samples are used for the target image, and `upsample_factor_options` define the upsampling factor used for the input images. `mipmap_bias` can be used to add an additional mipmap bias for both input and target images.

6.3 Dataset Conversion

This stage converts the dataset from .png to .hdf5. This is done in Python by running the `Network.py` file, and making sure that the `dataset.ConvertPNGDatasetToH5` function is ran at the start of the file.

6.4 Network Training

Network training is perfromed but the `Network.py`-file. It is important that the right values are set for the `model`, `load_model` and `loss_function` variables before starting the training. The trained network is saved to the `Network/modelMaster/` folder for each epoch of training. Pretrained networks can be found in `Network/MasterNet2x2/` and `Network/MasterNet4x4/`.

6.5 Network Conversion

This step converts the PyTorch model to a file that is easier read in C++. This is done in the `Network.py`-file by executing the `utils.SaveModelWeights` function after the trained model is loaded. This saves the network as a .bin file to `Network/nn_weights.bin`.

6.6 Network Loading

The last step is to make sure the right model is loaded when the `Anti-Aliasing/main.cpp`-file is ran. This is done by changing the file path for network loading. The variable containing the file path is located in `Rendering/deep_learning/master_net.cpp` and is named `weight_path`. When this variable is set correctly, and the upsampling factor in `Anti-Aliasing/app.cpp` is set correctly. The network can be ran in real-time by starting the `Anti-Aliasing`-project and pressing `7` on the keyboard.