NTNU

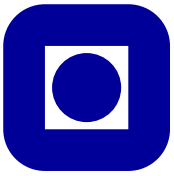
Norwegian University of
Science and Technology

Efficient video scaling algorithms implemented and optimized for FPGA.

Svein Erik Lindø

Master of Science in Electronics
Submission date: June 2011
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Ove Brynestad, Tandberg/Cisco

Norwegian University of Science and Technology
Department of Electronics and Telecommunications



NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

"Efficient video scaling algorithms implemented and optimized for FPGA"

Master Thesis

SVEIN ERIK LINDØ

SUBMISSION DATE: JUNE 13. 2011

SUPERVISORS:

KJETIL SVARSTAD, NTNU
OVE BRYNESTAD, CISCO

Problem Description

A theoretical foundation is presented and discussed in a pre-project as algorithms and basic implementations.

Based on previous theory, this master thesis aims to find more efficient methods for implementing filter structures for use in scaling. We also wish to investigate potential benefits and optimizations by using dynamically reconfigurable FPGA. The improved or new implementations should be compared to the implementation most commonly used today, the FIR-filter based Polyphase implementation.

Efficiency is measured in both hardware area requirements and throughput performance. The implemented design is aimed for real-time video scaling in teleconferencing systems, with close to natural image content.

Assignment given: 17. January 2011
Supervisor (NTNU): Kjetil Svarstad
Supervisor (Cisco): Ove Brynestad

Summary

The goal of this thesis was to find an alternative to a reference video scaler, providing the same or higher visual quality at faster operation, requiring less FPGA area and memory.

The first part of the report is used to lay a theoretical foundation within the field of video and image scalers. It is shown how the problem of video scaling is equivalent to the problem of discrete signal resampling in signal theory, and therefore also bound by the sampling theorem. The report explains why the the sinc function is the ideal interpolation kernel for signal reconstruction, and how the windowed sinc function is utilized in video/image scaling applications.

Three different video scaler algorithms Winscale, Edgeprocessor and Polyphasic FIR-filter Lanczos2 has been purposed. All algorithms are explained in theory and system architectures are suggested. Through visual quality tests based on matlab models, were the conclusion drawn that Winscale provides varying, non-predictable lower visual quality compared to the reference scaler and is not a suitable alternative. The adaptive Edgeprocessor shows potential as lower complexity alternatives providing better visual quality through the use of basic edge-detection and more complex calculations than Winscale. FIR-filter Lanczos2 is still seen as the better choice of implementation, as both Winscale and Edgeprocessor depends on a prescaler when downscaling below scale factor 0.5.

Configurable IP-core scalers from Altera has also been suggested, providing equal visual quality at maximum frequency requiring less FPGA resources. This solution would not provide full customizability and debug properties, as the source-code is not provided. The IP-cores can be configured with the most common interpolation kernels (Nearest, Bilinear, Bicubic, Polyphase).

The use of dynamic reconfigurable FPGAs in video scaling applications are shortly discussed, as the research provided only a limited amount of literature and examples within this specific field. My own thoughts and ideas on how reconfigurability may be utilized are presented.

Allthough no actual implementation of the described algorithms and system architectures is done, the thesis lays a theoretical foundation for future implementation of the three purposed scaler architectures.

Acknowledgments

This master thesis has been performed in the spring of 2011 at NTNU Trondheim and at Cisco at Lysaker. I want to give a special thanks to my two supervisors, Ove Brynestad at Cisco and Kjetil Svarstad at NTNU, for providing good guidance throughout the work on this master thesis.

-
June 13, 2011
Svein Erik Lindø

Contents

1. Introduction

Resizing or scaling of video is performed in a wide variety of electronics. The scaling is done to enlarge or reduce video dimensions to screen-size or even as a method of lossy video-compression. As with any lossy-compression method, scaling may induce noise or distortion of the information in the image. It is very important to reduce such scaling-induced noise in critical applications, such as military and medical applications. Loss of crucial information in these applications could have serious consequences. In handheld devices, such as smart-phones and tablet PCs, effective scaling-algorithms with low complexity have to take battery usage into account. Too complex calculations could seriously reduce battery lifetime. If the handheld device is used for visual video communications, video scaling could be performed to keep transmission within bandwidth limits. As HDTVs are becoming public domain, the entertainment industry uses video scaling to produce high quality transfers of old movies, which where not produced with the current display-resolutions in mind. This report will focus on video scaling in professional videoconferencing applications.

1.1. Video scaling in videoconferencing

Different video scalers are being utilized throughout the video transmission chain in a videoconferencing system. The sensor in the camera may capture a too high resolution video stream. Downscaling the video adapts the video stream for a bandwidth limited video processing system. Downscaling or upscaling at the receiver- or display-end may solve mismatches between video dimension and display resolution. Other applications may be producing compositions of multiple video streams or graphics.

Cisco [Cis11] challenged me in conducting a survey of existing video scaling algorithms, and compare them with a reference design provided by them. The goal of the master thesis would be to determine whether there exists a video scaling algorithm and implementation, providing higher visual quality compared to their reference design. Alternatively, scalers providing same visual quality at reduced hardware requirements could be explored. The scaler should be capable of both upscaling and downscaling with arbitrary rational scaling-factors. Aspect ratio should not necessarily be kept; vertical and horizontal scaling-factors is not necessarily the same. Computation time is very important to keep low, as videoconferencing systems contains strict real-time requirements. Delays in video processing will in the worst case result in the user experiencing the pace of communication as un-natural.

Figure 1 displays a generalization of how the video scaling problem applies to a videoconferencing system and one of the many applications video scalers are being utilized.

1.2. FPGA vs ASIC implementation

When implementing systems on ASICs and FPGAs, we are able to exploit pipelining and massively parallel processing. FPGAs gives us a flexible tool for rapid prototyping when developing digital processing circuits. The circuits are reprogrammable and dynamic reconfigurable, which provide opportunity for development and improvement of firmware after products are sold and shipped, or even in operation. ASICs could be an alternative choice of technology if the requirement of reprogrammability had been left out and the production volume were very high.

1.3. Relationship between Pre-Project Report and Master Thesis

As preparation for the master thesis, a pre-project were carried out with the goal of laying a theoretical foundation within *interpolation* and it's relationship to *image and video scaling al-*

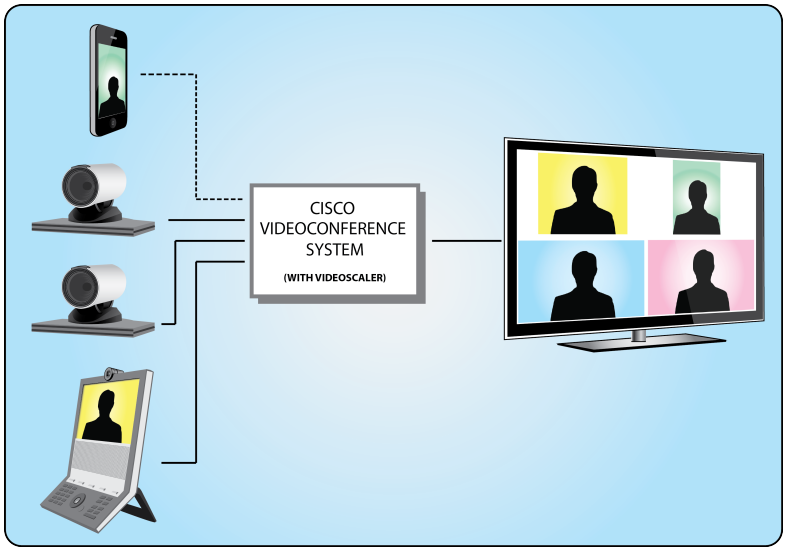


Figure 1: Basic videoconference setup

gorithms. My contribution to the pre-project was an overview and comparison of traditional and new experimental algorithms used in these applications, optimized and implemented on FPGA.

The pre-project and master thesis will have some common content, since the pre-project were aimed to lay the theoretical foundation. Instead of frequent referring to the pre-project, the most relevant theory has been included in this report. This applies to some of the theory in Section 2 - 5. The master thesis should be seen as an extension and an enhancement of the pre-project report, where some mistakes have been corrected, some concepts have been expanded and new ones have been added.

1.4. My contribution

My contribution to this master thesis has been to conduct a survey of already existing theory, algorithms and implementations. Due to time-limitations have analysis, evaluation of relevance and comparison of visual quality and resource requirements been performed based on scientific literature, tests results from own Matlab simulations and implementation documentation.

1.5. Report Outline

The report first presents the basics and definitions of video scaling in Section 2. In Section 3 are the concept and mathematics of signal interpolation explained. The traditional scaling kernels and algorithms are described and compared to the more complex kernels. Video quality and artifacts associated with scaling is presented in Section 4. The most widespread scaler based on the Lanczos2 Polyphase FIR Filter, is derived in Section 5. The implementation statistics of a reference scaler provided by Cisco i shown in Section 6.Two area-pixel based algorithms and implementations, Winscale and Edgeprocessor, is shown in Section 7 and 8. The visual quality and implementation requirements of these algorithms are compared to the reference scaler in Section 9 before FPGA manufacturer Altera's own IP-core solutions are shown in Section 10. Dynamical reconfigurability is shortly discussed in Section 11 before conclusions are drawn and future work discussed in Section 12.

2. Basics of video scaling

A video/image scaler reduces or increases the resolution size of a image or a frame in a video sequence. As a video-sequence basically is a sequence of images, we do not distinguish between video and image scalers. If the scaler have to perform scaling on a video-stream encoded with coding (such as MPEG), we could not use the algorithms and implementations described in this report without performing decoding of the video first. Within the process of scaling, the scaler has to decide which pixels to keep from the original source image, and which pixels have to be recreated through mathematical calculations. The source image has the dimension $X_s \cdot Y_s$, and the destination image $X_d \cdot Y_d$. From these notations, we can define the scaling factors as Equation 1 and 2.

$$S_x = \frac{X_d}{X_s} \tag{1}$$

$$S_y = \frac{Y_d}{Y_s} \tag{2}$$

There are two types of scaling: *pixel replication* and *pixel interpolation*. Pixel Replication, also known as zero order interpolation, replicates the pixels n times, thereby n-doubling the image in size. For the simplest implementation this results in poor image quality and very strong blocking or aliasing artifacts. This method is the oldest, but is still used in many systems because of its simplicity in hardware implementation. It only requires a linestore, a few registers for storing pixels, two multiplexing circuits and a simple control (for implementation details, see [Ber03] page 19.

Pixel Interpolation uses an image-processing filter, which calculates the intermediate pixel values with the use of mathematical interpolation models. These algorithms provides a much smoother scaled image, compared to pixel replication. Such models will be discussed in Section 3.

2.1. Scaling as an geometric operation

A geometric operation on an image transforms the given image I into a new image I' by modifying the coordinates of the image pixels (Equation 3). In transforming the image, a *mapping function* (Equation 4) is needed. The mapping function specifies the target pixel $\mathbf{x'} = (x', y')$ in the new image I' for each of the original pixels $\mathbf{x} = (x, y)$ in the original image.

$$I(x, y) \rightarrow I'(x', y') \tag{3}$$

$$\mathbf{x'} = T(\mathbf{x}) \tag{4}$$

The pixels in the image is defined on a discrete raster, rather than a continuous plane, because of the display's finite resolution. This leads to more strict requirements for the mapping function, in that the only valid resulting pixels from the function may be discrete values on the raster plane. This can be accomplished in two ways, in which differ by the mapping direction; **Source-to-Target** and **Target-to-Source**

Source-to-Target

In the Source-to-Target approach the mapping function compute, for every pixel (u, v) in the original source image I, the corresponding transformed position (x', y') in the new image I'. In

other words, the mapping function has to decide in which pixel in the target image I' the color intensity value from the original pixel $I(u, v)$ should be stored or even which several pixels it should be distributed amongst.

$$(x', y') = T(u, v) \tag{5}$$

The challenge with this approach is that some of the elements in the target image I' may never get assigned a value, depending on the geometric transformation T (Equation 5). This may result in areas or "holes" in the target image without assigned values. A very low-complex example is shown in Figure 2. The algorithm in this example calculates the target pixels in a *for-every-pixel-in-source* fashion. Since the mapping function is rather unintelligent, only selected target pixels get assigned a value. Such an algorithm grows more complex, if equal distribution of pixel intensities is to be ensured. For this reason, the source-to-target is not the ideal approach.

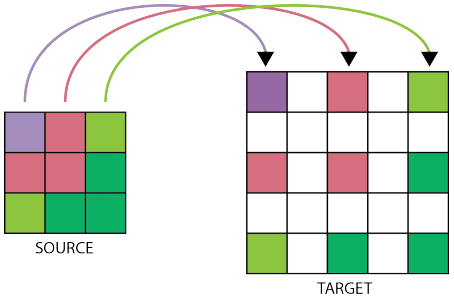


Figure 2: The simplest Source-to-target method

Target-to-Source

The Target-to-Source method takes the opposite approach, where for every discrete pixel position (u', v') in the target image I', it computes the corresponding continuous pixel position (x,y) in the source image plane, using the inverse geometric transformation T^{-1} (Equation (6)). The coordinate (x, y) does not necessarily fall onto a raster point, so the method have to decide from which neighboring source pixel to extract the resulting pixel values. This method solves Source-to-Target's "hole"-problem, but we have to be careful in designing such methods such that all relevant information in the source image is taken in account. We will later see how edge sharpness may suffer from this approach.

$$(x, y) = T^{-1}(u', v') \tag{6}$$

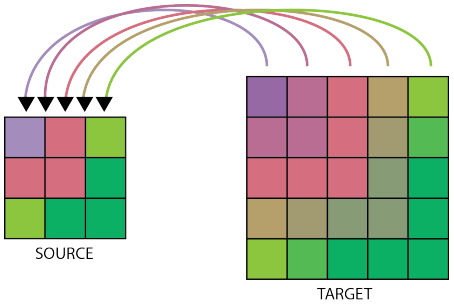


Figure 3: Target-to-source

2.2. Pixel Models

It is important to define how the pixels are modeled, as the mathematical problem of image scaling gets defined. This report uses two different models called the *Point Pixel Model* and the *Area Pixel Model*.

Point pixel model

The point pixel model treats each pixel as a point with a finite placement on a two-dimensional raster. The image scaling is simply a resampling of the two-dimensional function on a new sampling grid. Figure 4 shows the scaling problem in one dimension, where the green pixel is the interpolated pixel used in the target image. The intermediate intensity values can be calculated with the presented interpolation kernels in Section 3. This pixel model is closely connected to the mathematics and statistics of digital/discrete signal theory, which is a well established field of research.

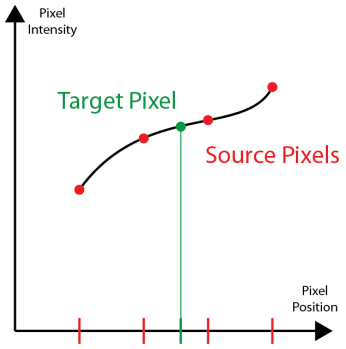
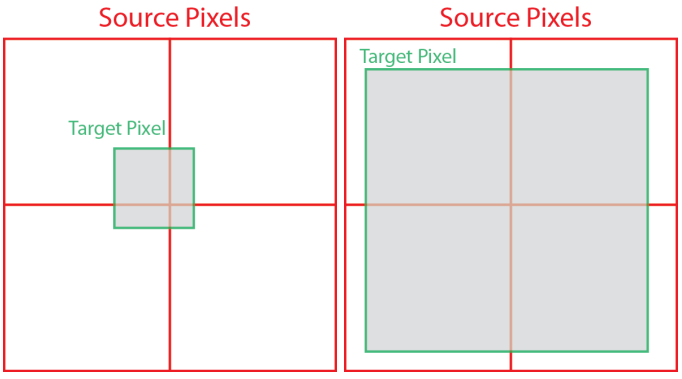


Figure 4: Pixel point model

Area pixel model

Instead of treating each pixel as a point, the area pixel model treats each pixel as a rectangle with the intensity evenly distributed throughout the rectangle. The scaling problem therefore becomes the problem of calculating the size of each overlapping region of the target pixel and each source pixel (shown in Figure 5). The relative size of each overlapping region corresponds to the coefficients in the scaling filter.



(a) Upscaling: Target pixel is smaller than source pixel

(b) Downscaling: Target pixel is larger than source pixel

Figure 5: Area pixel model

3. Interpolation

This section explains how the problem of image scaling can be solved by the concept of two dimensional signal interpolation. We first show how ideal interpolation is derived from the signal sampling theory. Then several different interpolation kernels are presented.

[MB10] defines interpolation as *"the process of estimating the intermediate values of a sampled function or a signal at continuous positions or the attempt to reconstruct the original continuous function from a set of discrete samples"*. Our goal by interpolation, is ultimately to obtain an estimate for the value of the two-dimensional image function I(x,y) at any continuous position (x,y). We should preserve as much detail as possible without visible artifacts such as ringing or moiré patterns (Explained in Section 4).

The interpolation algorithms can be divided into the two catergories *adaptive* and *non-adaptive*. The non adaptive algorithms treat all pixels equal, and are in that way simpler to implement. Nearest Neighbor, bilinear, bicubic, Spline and Lanczos are some non-adaptive algorithms.

The adaptive algorithms change functionality and settings based on the what information is interpolated. These algorithms often detect edges and compensates for this while processing the information. According to [Cam10] are most of these algorithms proprietary algorithms used in licensed software like Qimage[Qim10], PhotoZoom Pro ([Ben10]) and Genuine Fractals[oS10], although an open adaptive algorithm used in the Edgeprocessor is presented in Section 8.

3.1. Sampling Theorem

To fully understand the concept of interpolation, we need to know the basics of sampling and the *Sampling Theorem*. Sampling is the process of converting a continuous function $f(x)$ into a discrete signal $f[n]$. This is done by extracting values at regular intervals, shown in Equation 7. A important feature when sampling, is to construct the discrete signal in such a way that the original continuos signal could be perfectly reconstructed. The sampling theorem (Equation 8) states that in order to keep the property of perfect reconstruction; *the continuos signal must be sampled at a sampling frequency ω_s, at least twice the frequency of the maximum frequency component ω_{max}.*

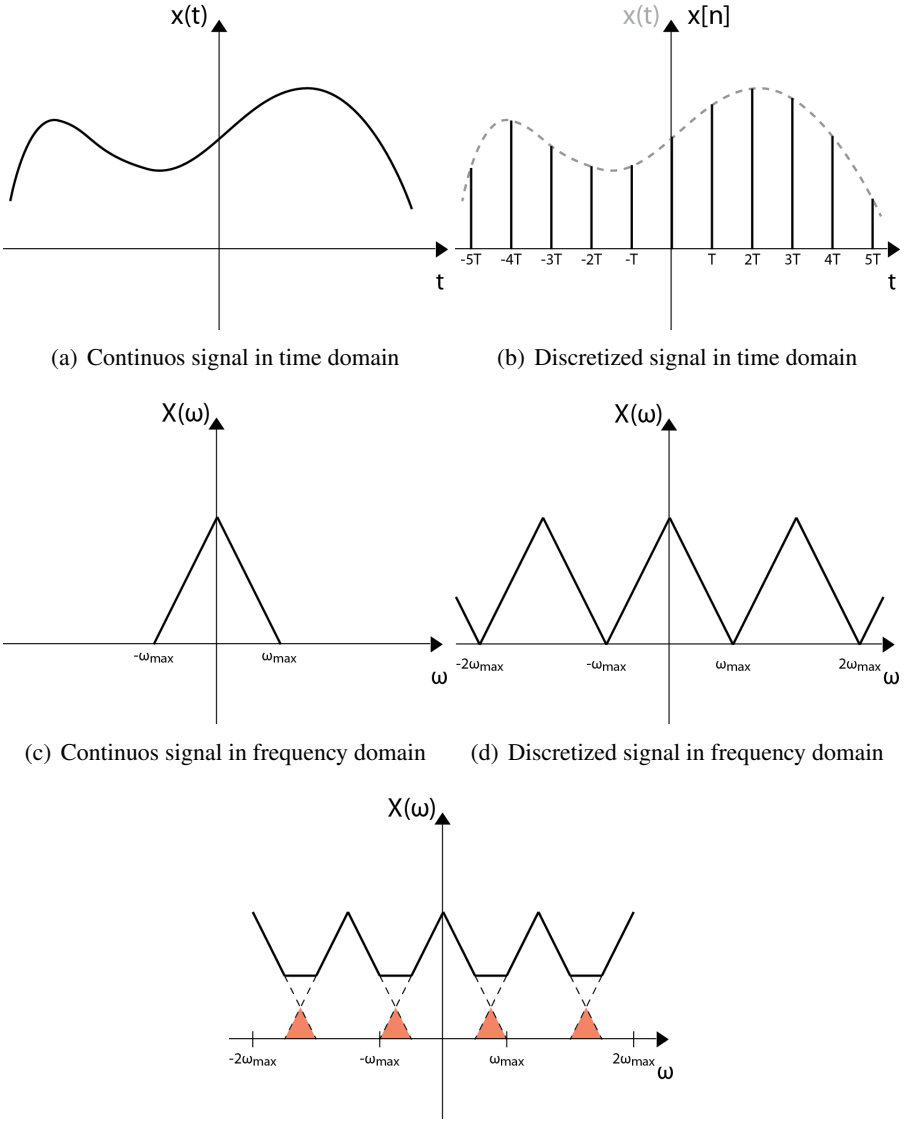
$$f[n] = \sum_{k=-\infty}^{\infty} f(kT) \tag{7}$$

$$\omega_s \geq 2 \cdot \omega_{max} \tag{8}$$

Figure 6 shows the relationships between the continuos signal and the discrete signal in both time and frequency domain. When discretized with a sampling frequency ω_{max}, the frequency response $X_P(\omega)$ will be exact replicas of the frequency response $X(\omega)$ with period ω_{max} (Figure 6(d)). A lower sampling frequency would result in overlapping of the replicas in the frequency domain (shown in Figure 6(e)). This artifact would make it impossible to reproduce the original continuous signal from the discrete signal. More information on the sampling theorem can be found in [Ber03].

The process of interpolation is actually the process of resampling a signal by trying to reproduce the original signal and resample it at a higher sampling rate. In reproducing this signal, we have to we isolate the frequency range $[-\omega_{max}, \omega_{max}]$ in the frequency spectrum (Figure 6(d)). This is done by filtering the signal with a *ideal low-pass filter* given in the frequency domain by Equation (9).

$$\Pi_\pi(\omega) = \begin{cases} 1 & \text{for } -\pi \leq \omega \leq \pi \\ 0 & \text{otherwise} \end{cases} \tag{9}$$



(a) Continuos signal in time domain

(b) Discretized signal in time domain

(c) Continuos signal in frequency domain

(d) Discretized signal in frequency domain

(e) Aliased signal in frequency domain

Figure 6: The Sampling Theorem

The inverse Fourier Transform of the ideal low-pass filter corresponds to *the Sinc function* (Equation (10)) in the time domain. This is the ideal interpolation function for reconstructing a frequency-limited continuous signal.

$$Sinc(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{sin(\pi x)}{\pi x} & \text{for } |x| > 0 \end{cases} \tag{10}$$

The continuous signal $f(x)$ can be perfectly reconstructed from Equation 11. The process is shown visually in Figure 7.

$$x(t) = \sum_{k=-\infty}^{\infty} x(kT) Sinc(\frac{t - kT}{T}) \tag{11}$$

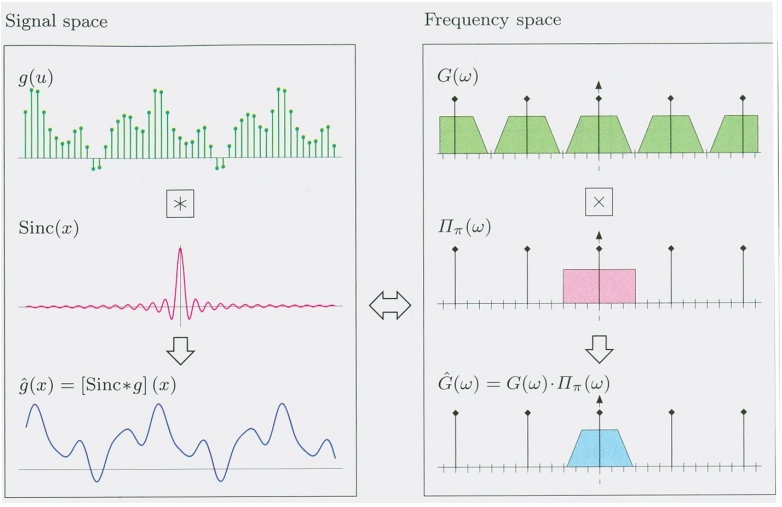


Figure 7: Ideal Interpolation [MB10]

3.2. Ideal Interpolation

In our application of image scaling, we use the concept of interpolation to calculate the value for a discrete function $g(u)$ at an arbitrary position x_0, a position where the discrete image function does not provide any valid values. The Sinc function is therefor centered around x_0, multiplied with all sample values of $g(u)$ and then summed (Equation 12, convolution).

$$\hat{g}(x_0) = Sinc * g = \sum_{u=-\infty}^{\infty} Sinc(x_0 - u)g(u) \tag{12}$$

The ideal interpolation is defined in 2 dimensions as the 2D Sinc function in Equation 13 and Figure 8. A challenge in using the Sinc function occurs around high-frequency signal events, such as rapid transitions or pulses (containing infinite large frequency components), where it causes strong overshooting or "ringing" artifacts. These artifacts may be reduced by doing a lowpass filtering first. An example of such ringing artifacts is shown in shown in Figure 9. The original continuous signal contained a sharp object, like a hair or a line in some clothing. When the original image were tried to be reconstructed from the perfect Sinc interpolation (Equation 12), it resulted in smoother edges with actually higher amplitudes in some regions. This would visually result in wrong pixel intensity values in part of the image, and could in worst case be perceived as quality-degrading. This proves that the ideal interpolation based on the infinite Sinc-function is not perfect. The design of the optimal interpolation kernel is always a tradeoff between high bandwidth (sharpness) and good transient response (low ringing effects).

$$SINCx, y = Sincx \cdot Sincy = \frac{sin(\pi x)}{\pi x} \cdot \frac{sin(\pi y)}{\pi y} \tag{13}$$

The Sinc function has also an infinite extend. For practical applications is the Sinc therefore not suitable as an interpolation kernel. In the next sections, we'll therefore present some interpolation kernels which approximates the Sinc function. These methods all cope with the extent-problem by windowing the approximation or the sampled version of the Sinc function within a finite interval.

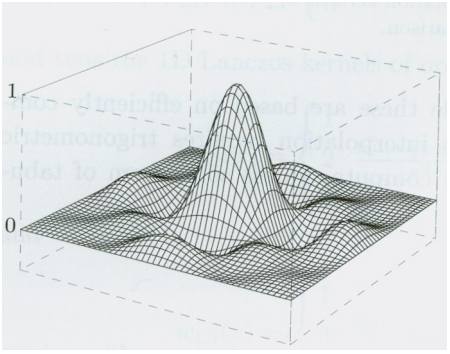


Figure 8: Ideal Interpolation 2D [MB10]

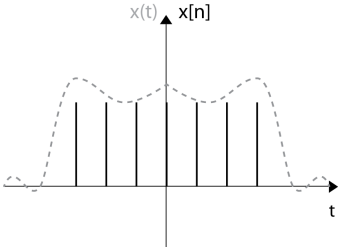


Figure 9: "Ringing" effects in high frequency areas caused by Sinc based interpolation

3.3. Nearest-neighbor Interpolation

The simplest way of interpolation is called nearest-neighbor interpolation. This method rounds up the continuous coordinate x to the closest integer u_0 and use the sample $g(u_0)$ as the estimated function value in Equation (14). The Nearest-neighbor Kernel is given by Equation (15)

$$\begin{aligned} \hat{g}(x) &= g(u_0) \\ u_0 = round(x) &= \lfloor x + 0.5 \rfloor \end{aligned} \tag{14}$$

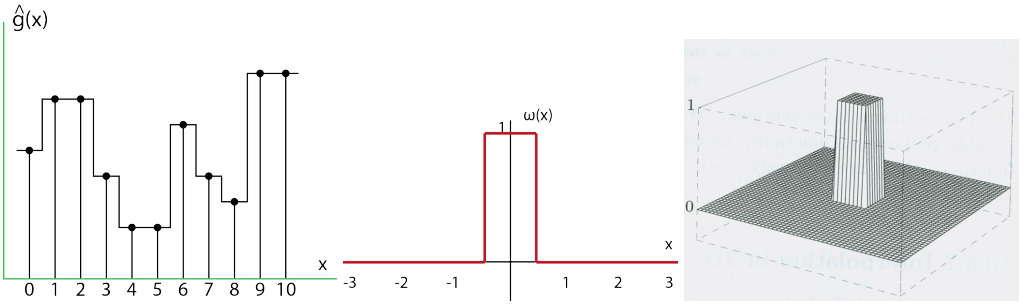
$$\omega_{nn}(x) = \begin{cases} 1 & \text{for } -0.5 \leq x \leq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

Nearest-neighbor in 2D

In the 2D case, the pixel closest to a given continuous point (x_0, y_0) is found by rounding the x and the y coordinates independently to integer values in Equation (16). The 2D kernel of the nearest-neighbor interpolation is defined as Equation (17). A visual representation of the 1D and 2D kernel is shown in Figure 10(b) and Figure 10(c). This method is rarely used because of its blocking effects on the reconstructed signal. An example of such effects is shown in Figure 10(a).

$$\begin{aligned} \hat{I}(x_0, y_0) &= I(u_0, v_0) \\ u_0 = round(x_0) &= \lfloor x_0 + 0.5 \rfloor \\ v_0 = round(y_0) &= \lfloor y_0 + 0.5 \rfloor \end{aligned} \tag{16}$$

$$W_{nn}(x, y) = \begin{cases} 1 & \text{for } \quad -0.5 \leq x, y < 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$



(a) Example (b) 1D Kernel (c) 2D Kernel [MB10]

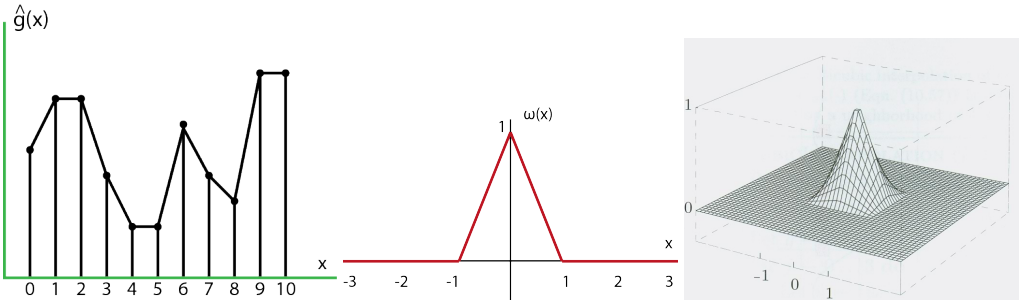
Figure 10: Nearest-neighbor Interpolation

3.4. Linear Interpolation

As nearest neighbor produces very blocky images, the need for some smoothening of the edges arrises. Linear interpolation estimates the intermediate value $\hat{g}(x)$ as the sum of the two closest samples $g(u_0)$ and $g(u_0 + 1)$, where $u_0 = |x|$ (Equation (18)). The weight of each sample is proportional to its closeness to the continuous position x. This provides much smoother transitions between the existing samples in our example in Figure 11(a). The Linear Kernel is given by Equation (19).

$$\hat{g}(x) = g(u_0) + (x - u_0)(g(u_0 + 1) - g(u_0)) \tag{18}$$

$$\omega_{lin}(x) = \begin{cases} 1 - x & \text{for } |x| < 1 \\ 0 & \text{for } |x| \geq 1 \end{cases} \tag{19}$$



(a) Example (b) 1D Kernel (c) 2D Kernel [MB10]

Figure 11: Linear Interpolation

[zip09] suggests that the linear kernel could be modified with what is called *successive smoothing*, which is implemented by raising the kernel near zero and lowering it more near the end. Figure 12 shows two examples of such smoothening. We can clearly see from this illustration that added complexity of this kernel with successive smoothening, would have the shape of the sinc-based kernels.

Bilinear Interpolation in 2D

In the two-dimensional space, the linear interpolation is called Bilinear Interpolation. For the given interpolation point (x_0, y_0) we first find the four closest pixels A, B, C, D in the image I defined by Equation (20).

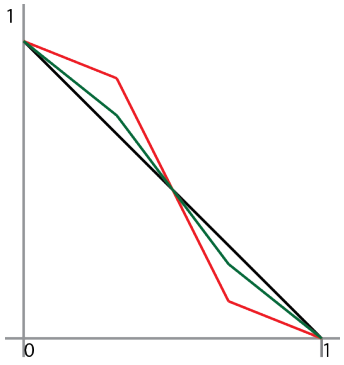


Figure 12: Successive Smoothing

$$\begin{aligned} A &= I(u_0, v_0) \\ B &= I(u_0 + 1, v_0) \\ C &= I(u_0, v_0 + 1) \\ D &= I(u_0 + 1, v_0 + 1) \\ u_0 &= \lfloor x_0 \rfloor \\ v_0 &= \lfloor y_0 \rfloor \end{aligned} \tag{20}$$

The distance from these pixels determines the weighing of the different pixel values. The calculations needed is defined in Equation (21) and summarized as the *linear convolution filter kernel* in Equation (22) and Figure 11(c).

$$\begin{aligned} &\text{Horizontal:} \\ a &= (x_0 - u_0) \\ E &= A + (x_0 - u_0)(B - A) \\ F &= C + (x_0 - u_0)(D - C) \\ &\text{Vertical:} \\ b &= (y_0 - v_0) \\ \hat{I}(x_0, y_0) &= E + (y_0, v_0)(F - E) \\ \hat{I}(x_0, y_0) &= (a-1)(b-1)A + a(1-b)B + (1-a)bC + abD \end{aligned} \tag{21}$$

$$\begin{aligned} W_{bil}(x, y) &= w_{in}(x) + w_{lin}(y) \\ W_{bil}(x, y) &= \begin{cases} 1 - x - y - x \cdot y & \text{for} \quad 0 \leq |x|, |y| < 1 \\ 0 \quad otherwise \end{cases} \end{aligned} \tag{22}$$

3.5. Cubic Interpolation

Cubic Interpolation is an approximation of the Sinc function and defined as the piecewise polynomial in Equation (23). The control parameter a adjusts the slope of the Spline function, which affects the amount of overshoot and perceived "sharpness" of the interpolated signal. Compared to the Sinc function, the cubic interpolation kernel has a very small extent and is therefor efficient to compute.

- $a = 1$ is often the recommended standard setting
- $a = 0.5$ is called a Catmull-Rom Spline, and gives better results than $a = 1$, particulary in smooth regions

$$\omega_{cub}(x,a) = \begin{cases} (-a+2)|x|^3 + (a-3)|x|^2 + 1 & \text{for} \quad 0 \leq |x| < 1 \\ -a|x|^3 + 5a|x|^2 - 8a|x| + 4a & \text{for} \quad 1 \leq |x| < 2 \\ 0 & \text{for} \quad |x| \geq 2 \end{cases} \tag{23}$$

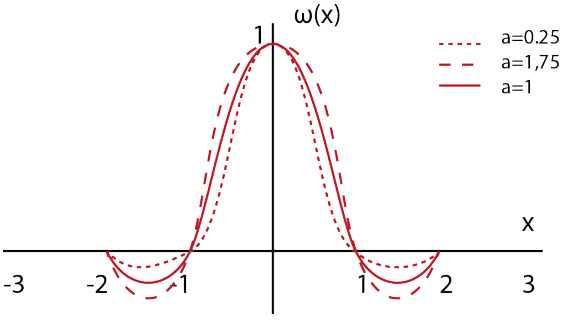


Figure 13: Cubic Interpolation Kernel

3.6. Spline Interpolation

Another more general approximation of the Sinc function is called Spline interpolation (Figure 14).

$$\omega_{cs}(x,a,b) = \begin{cases} (-6a-9b+12)|x|^3 + (-6a+12b-18)|x|^2 - 2b + 6 & \text{for} \quad 0 \leq |x| < 1 \\ (-6a-b)|x|^3 + (30a+6b)|x|^2 + (-48a-12b)|x| + 24a + 8b & \text{for} \quad 1 \leq |x| < 2 \\ 0 & \text{for} \quad |x| \geq 2 \end{cases} \tag{24}$$

- $\omega_{cs}(x,a,0)$ is equal to the **Cubic Interpolation**
- $\omega_{cs}(x,0.5,0)$ is called **Catmull-Rom Interpolation**. This method emphasizes high sharpness and provides good performance in smooth signal regions.
- $\omega_{cs}(x,0,1)$ is called **Cubic B-spline Approximation**. The approximation does not go through all points, but creates no ringing effects.
- $\omega_{cs}(x,\frac{1}{3},\frac{1}{3})$ is called **Mitchell-Netravali Approximation**. This method is a weighted sum of Catmull and Cubic B-Spline. It provides less overshoot, high edge sharpness and good signal continuity in smooth signal regions.

Detailed polynomial for each function can be found in [MB10].

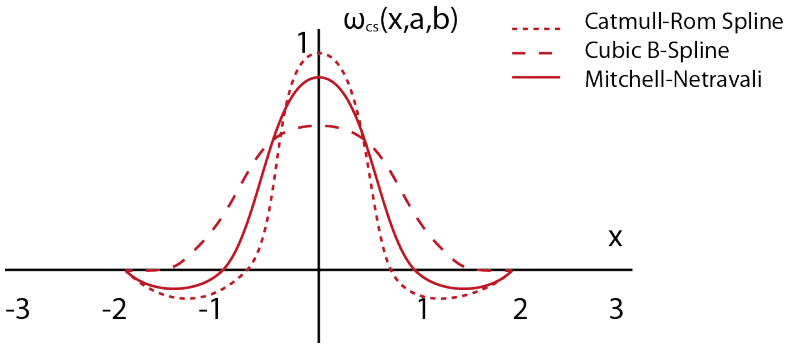


Figure 14: Examples of Spline Interpolation functions

Bicubic and Spline Interpolation in 2D

The Bicubic and Spline 2D interpolation kernels are defined in Equation (25) by the 1D kernels. The interpolated value $\hat{I}(x_0, y_0)$ can be calculated using Equations (26). These equations is based on a 4x4 neighborhood of pixels where p_j denotes the intermediate result for the cubic interpolation in the horizontal direction in line j. This example requires 20 additions and multiplications (16 + 4). [MB10](p.233) presents a pseudocode for the General Spline Interpolation.

$$W_{bic}(x, y) = w_{cub}(x) \cdot w_{cub}(y) \tag{25}$$

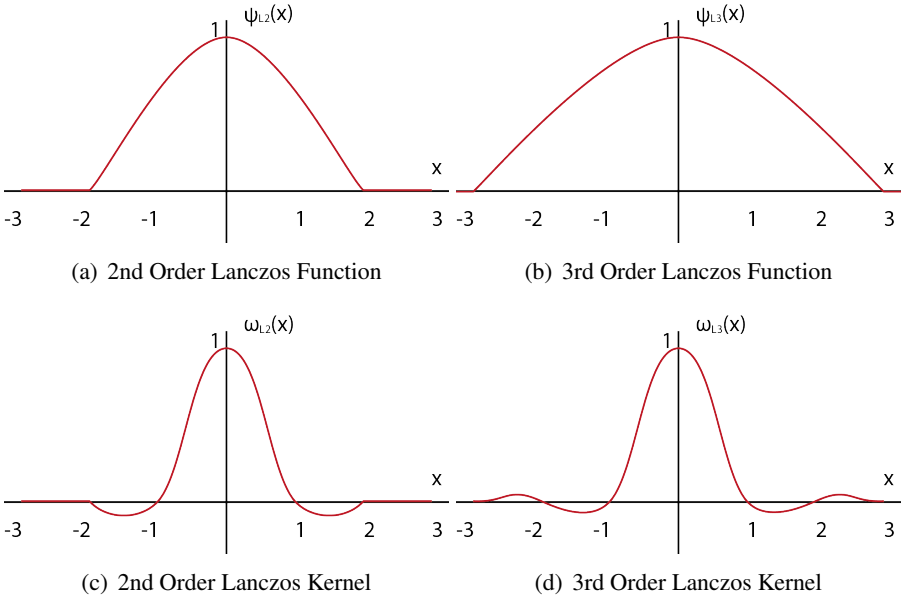
$$\begin{aligned}
\hat{I}(x_0, y_0) &= \sum_{v=\lfloor y_0 \rfloor - 1}^{\lfloor y_0 \rfloor + 2} [\sum_{v=\lfloor x_0 \rfloor - 1}^{\lfloor x_0 \rfloor + 2} [I(u, v) \cdot W_{bic}(x_0 - u, y_0 - v)]] \\
\hat{I}(x_0, y_0) &= \sum_{j=0}^{3} [w_{cub}(y_0 - v_j) \cdot \sum_{i=0}^{3} [I(u_i, v_j) \cdot w_{cub}(x_0 - u_i)]] \\
p_j &= \sum_{i=0}^{3} [I(u_i, v_j) \cdot w_{cub}(x_0 - u_i)] \\
u_i &= \lfloor x_0 \rfloor - 1 + i \\
v_j &= \lfloor y_0 \rfloor - 1 + j
\end{aligned} \tag{26}$$

3.7. Lanczos Interpolation

The Lanzos Interpolation (Equation (27)) method utilizes the actual Sinc function combined with a suitable windowing function $\psi(x)$. Equation (28) defines the Lanczos window function, where n denotes the *order* of the filter. The most commonly used orders are 2 and 3. Lanczos interpolation requires trigonometric functions, which are relative costly to compute. They are for this reason sampled, and stored in memory in the implemented architecture, rather than computed on-the-fly. Another alternative is computation in software processors with accurate trigonometric functions.

$$\omega(x) = \psi(x) \cdot Sinc(x) \tag{27}$$

$$\psi_{Ln}(x) = \begin{cases} 1 & \text{for} \quad |x| = 0 \\ \frac{sin(\pi \frac{x}{n})}{\pi \frac{x}{n}} & \text{for} \quad 0 \leq |x| < n \\ 0 & \text{for} \quad |x| \geq n \end{cases} \tag{28}$$



(a) 2nd Order Lanczos Function (b) 3rd Order Lanczos Function

(c) 2nd Order Lanczos Kernel (d) 3rd Order Lanczos Kernel

Figure 15: Lanczos Interpolation

Lanczos Interpolation in 2D

Equation (29) is the general equation for a 2D Lanczos Interpolator of order n. The L3 (n=3) Lanczos interpolation in 2D uses the support of $6x6 = 36$ pixels, 20 more than bicubic interpolation. It is important to include enough pixels when scaling down, to ensure that all relevant information is transferred to the target image.

$$\begin{aligned} \hat{I}(x_0, y_0) &= \sum_{v=\lfloor y_0 \rfloor - n + 1}^{\lfloor y_0 \rfloor + n} [\sum_{v=\lfloor x_0 \rfloor - n + 1}^{\lfloor x_0 \rfloor + n} [I(u, v) \cdot W_{Ln}(x_0 - u, y_0 - v)]] \\ \hat{I}(x_0, y_0) &= \sum_{j=0}^{2n-1} [w_{Ln}(y_0 - v_j) \cdot \sum_{i=0}^{2n-1} [I(u_i, v_j) \cdot w_{Ln}(x_0 - u_i)]] \\ u_i &= \lfloor x_0 \rfloor - n + 1 + i \\ v_j &= \lfloor y_0 \rfloor - n + 1 + j \end{aligned} \tag{29}$$

4. Video Quality and Scaling Artifacts

To make good quality comparisons between the different video scaler implementations, it is important to have knowledge of what artifacts and picture distortions we could expect. These concepts will be very important when the algorithms are implemented. They are presented now as they may already help us to distinguish between the algorithms.

Video quality is subjective and cannot be measured accurately as a measurable number. Since we essentially construct information, we cannot draw adequate visual quality measures by comparing the statistics and mathematics of the source and target images. The most common way of measuring image quality is to look for known types of artifacts that might have appeared during the scaling process. This is still a manual and subjective approach.

There are several types of artifacts associated with non-adaptive scaling algorithms. These artifacts will degrade the perceptual quality of the image frame. [Cam10] describes three such artifacts; Edge Halo, Blurring and Aliasing (Figure 16). The resulting image will always contain a combination of these artifacts, as they are closely connected. Suppressing one of them comes at the expense of the other two. [Cam10] presents a diagram (Figure 17) in which they place different scaling algorithms according to their artifact susceptibility. The diagram shows how there is a tradeoff between the artifact types. (Algorithms 3,5 and 7 will not be described in this report). The nearest neighbor is most aliased, and along with bilinear, these two are the only two that have no edge haloing. The edge sharpness gradually increases from 3-5, but at the expense of both increased aliasing and edge halos. Lanczos and bicubic are most often used since they perform better than the other algorithms with reference to these artifacts. They are however more complex in implementation compared to nearest neighbor and bilinear.

1. Nearest Neighbor
2. Bilinear
3. Bicubic Smoother
4. Bicubic Sharper
5. Bicubic
6. Lanczos
7. Bilinear w/blur

(a) Aliasing (b) Blurring Edge (c) Edge Halo

Figure 16: Scaling Artifacts [Cam10]

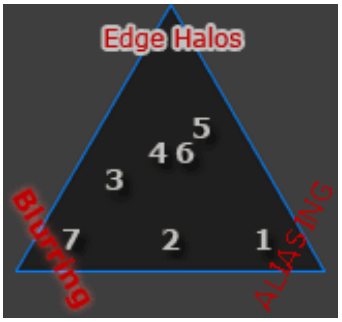
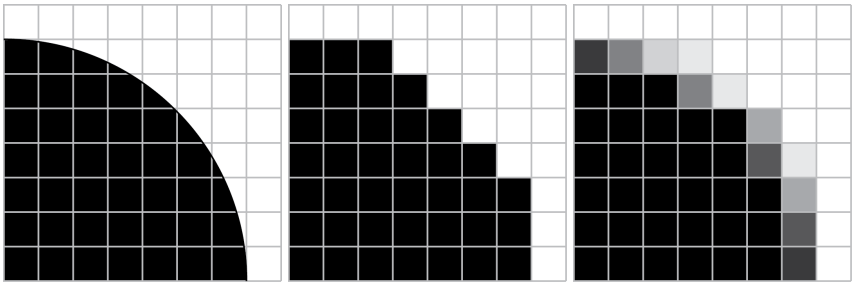


Figure 17: Artifact Trade-offs [Cam10]

Anti-Aliasing

Anti-aliasing is the process of reducing or/and minimizing the amount of aliased or "jagged" diagonal edges. The edges may be smoothened by calculating how much an ideal edge overlaps the actual pixel-area. From this it can be determined how much of the pixel value should be taken in account. Figure 18(c) shows how the edge becomes more smooth by assigning a more gradient distribution of color values to the edge. The challenge in using this method, is to find a trade-off between the extremely jagged edge, and the blurry, unshaped edge. This may vary between applications and image sources.



(a) Ideal Edge (b) Aliased Edge (c) Anti-Aliased Edge

Figure 18: Anti-Aliasing

Moiré Artifacts

Another aliasing artifact, when downsizing an image, is called moiré (shown in Figure 19(b)). This effect comes to display in images containing fine grained repeating pattern textures. After downsizing such an image, the size of details in the texture may be below the new lower pixel size resulting in only selective records of the repeating pattern. Images with fine geometric patterns, such as roof tiles, distant bricks and line texture in a suit, are at high risk of generating this artifact. In these cases, It may often be useful to simulate the blurry properties the human sight has, when observing objects at a distance. Interpolation algorithms that preserve best sharpness are more susceptible to moiré, where as those that avoid moiré typically produce a softer result.

Several simple, but good interactive demonstrations on artifact appearance and anti-aliasing can be found at [Cam10].

(a) Source Image

(b) Downsized Image with moiré

Figure 19: Moiré Artifacts

5. FIR-filter Image Scaling

The most common way of implementing a digital scaling filter, is by using a FIR filter. Such a scaler will be presented in the following section.

Years of research and experience has led the industry to believe that the best video scalers are implemented as *finite impulse response* (FIR) filter structures (Figure 20). The filter is characterized by its filter transfer function $H(z)$ (Equation 30) or its impulse response response $h[n]$ (Equation 31), which the input signal $x[n]$ is convoluted by, to produce the output signal $y[n]$ (by Equation (32)). In this case $x[n]$ is the original signal extended by inserting zero-value pixels in between the original pixels. By sending this signal through the filter, all target pixel are based on source pixels.

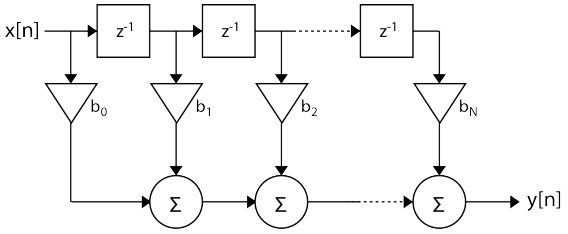


Figure 20: FIR filter structure

$$H(z) = \sum_{n=0}^{N} b_n z^{-n} \tag{30}$$

$$h[n] = \sum_{i=0}^{N} b_i \delta[n-i] \tag{31}$$

$$y[n] = \sum_{k} x[n+k]h[-k] \tag{32}$$

A common mistake when talking about upsampling processes using FIR filters, is calling the filters for *polyphase filters*. If an ordinary FIR filter is used to process a signal with inserted zero-values prepared for upsampling, and the multiplications by zeroes is removed, the implementation is *polyphasic*.

The straight forward way to implement a *n*-phase filter, is to implement *n* discrete filters and route the input pixels to the needed filter. This creates large overhead, since each filter contains several multiplications and additions. Instead we implement a single filter where coefficients are multiplexed from memory, using the phase as the selector. The general design is shown in Figure 21. The number of phases a filter is split into depends only on the upsampling factor. Each phase consists of several coefficients called *taps*. In other words, taps is the number of neighboring pixels the new pixel is dependent on.

5.1. Integer Upscaling Factors

To scale a signal by an integer factor of m, where m is a positive integer, m-1 zeroes are inserted between all samples and a low pass filter is used to remove the mirrored spectra generated by the insertion of zeroes. The filter is divided into m phases, shown in Figure 22, while the hardware architecture for a filter with three taps per phase is shown in Figure 23. The filter generates m output samples for a given set of input samples. The output sequence $y[n]$ is collected from

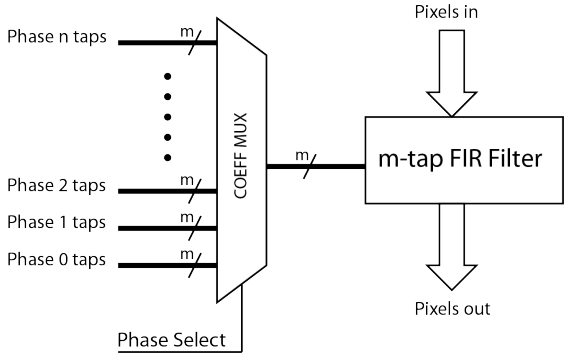


Figure 21: A N-phase, M-tap polyphase FIR filter architecture

the filter by "rotating" between the phases with a frequency of m. From Figure 23 we see that we require two linestores in addition to registers to make three pixels available in the vertical direction. Each pixel also requires its own phase in multiplication.

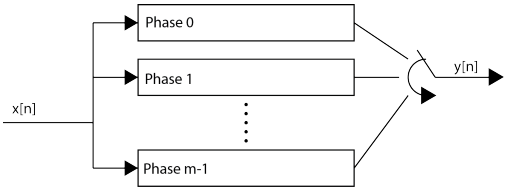


Figure 22: A generalization of a FIR filter divided into different phases

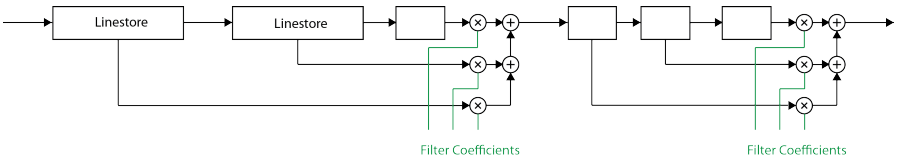


Figure 23: Hardware Architecture of a filter with three taps per phase

5.2. Rational Up- and Downscaling Factors

A rather easy way to obtain scaling with a rational factor m/n (where both are positive integers), is first to upsample the signal with a factor of m, then downsample it with a factor n. The downsampling process is the easiest, as it only requires extracting every n*th* pixel. Following this, we see from Figure 24 that a lot of redundant calculated pixel values are discarded during the downsampling. Every pixel value from a redundant calculation is highlighted in yellow. To make the overall scaling-process more effective, it is crucial that these redundant calculations are not performed. Our goal for the upsampling process is therefore to calculate only pixel values for which is needed in the downsampling process. From the illustration we observe that the extracted pixels are numbered 0, 4, 3, 2, 1... To select which calculations to preform, a algorithm using modulo 5 is used, since modulo 5 of 0, 4, 8, 12 and 16 is 0, 4, 3, 2, 1. As scaling factors increase in precision, the process requires increasingly larger amount of phases. As an example would a scalingfactor of 1000/1009 require as much as 1009 phases. To solve this challenge, it becomes necessary to perform some approxmiations. Figure 25 shows an example of a non-uniform sampling with a maximum deviation of $1/8$. From this, we see that a filter with

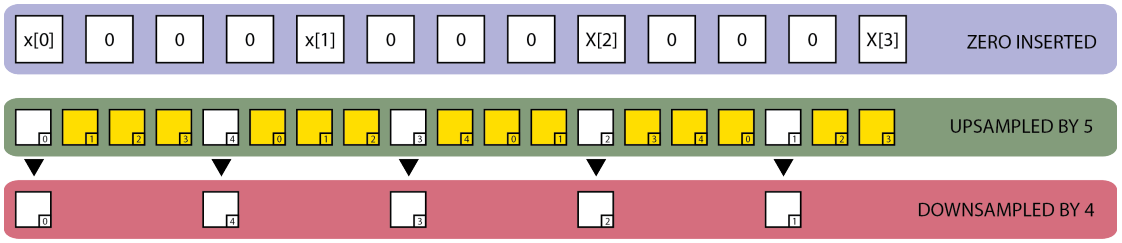


Figure 24: Upsampling of 5/4. Calculation of yellow pixels are redundant, as they are discarded.

4 phases will have a maximal deviation of $1/8$ from the ideal pixel position. [Ber03] claims that

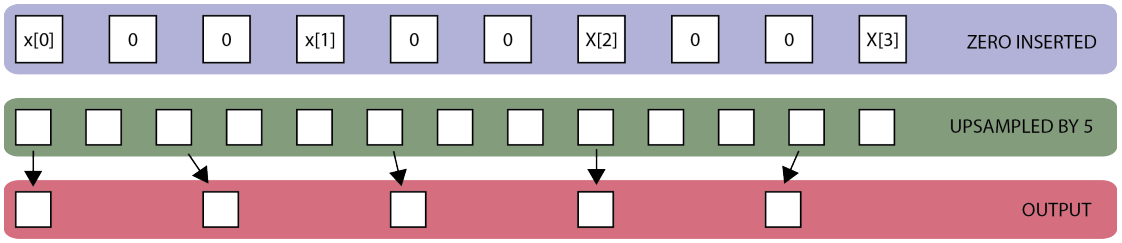


Figure 25: An approximation is needed as rational factors gets more precise

low-quality systems require 4 bits of precision when representing phase shift, while high-quality systems require 8 bits. Experiments show that filters with 7 bit precision and 64 phases is enough ([Ber03]). The maximal error would then be $1/128$. This can all be summed up in the following algorithm.

1. Insert 63 zeroes between all consecutive input samples.
2. Calculate the position of a desired output sample.
3. Take the closest neighbor of that sample among the intermediate layer upsampled by 64.
4. Calculate through digital filtering the closest neighbor.
5. Output the calculated sample

5.3. Linear Interpolation as FIR filter

In the case of linear interpolation (Section 3.4) Equation (18) can be written as Equation (33) where $\alpha = x - u_0$ and $k = x$. This equation is written in a form that is easy to transform into filter structures, shown in Figure 27. This implementation requires

- 2 multiplication
- 2 additions
- 1 linestore
- 2 registers

$$y[k] = \alpha(x[n+1] - x[n]) + n \tag{33}$$

We assume that the filter we use has 64 taps. In other words: new samples can be placed at 63 different positions between two already existing samples. With a coefficient precision

of 8 bits, phases are given as shown in Table 1. In other words, the digital filter is given by $h[] = \{0, 4, 8, ..., 252, 256, 252, ..., 4\}$. Each of the coefficients can be found by reading out the value of the kernel at tap0 and tap1 in Figure 26. For each phase-shift the curve is shifted right by $1/64$. The coefficients are in other words samples of the linear kernel at intervals $1/64$.

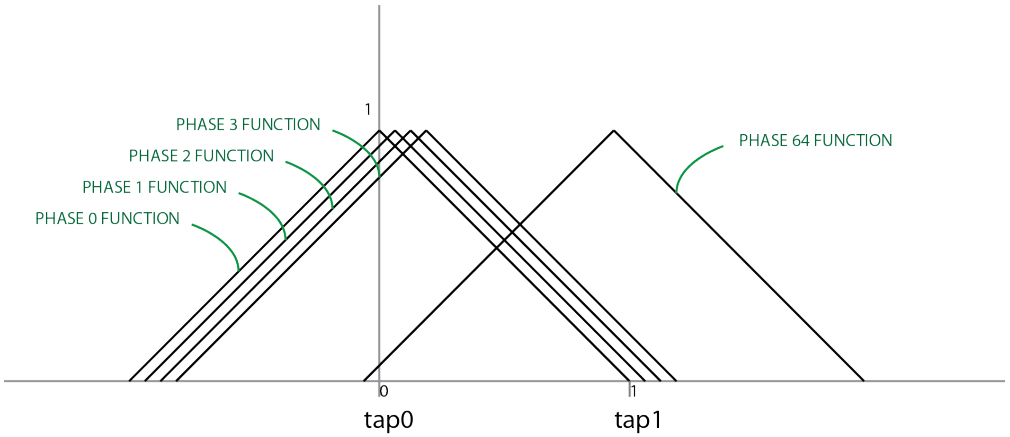


Figure 26: 64-phase 2tap sampling of linear kernel

Table 1: 64 tap Linear Interpolation with 8 bit precession

Phase	*Tap 1*	*Tap 2*
phase 0	256	0
phase 1	252	4
phase 2	248	8
⋮	⋮	⋮
phase 63	4	252

Phase	*Tap 1*	*Tap 2*
phase 0	h[64]	h[0]
phase 1	h[65]	h[1]
phase 2	h[66]	h[2]
⋮	⋮	⋮
phase 63	h[127]	h[63]

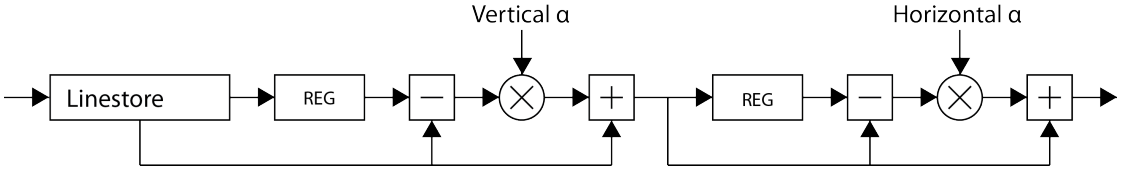


Figure 27: A FIR filter implementation of the 2D bilinear interpolation

Exploiting Phase Symmetry

In applications where coefficients are not generated at run-time but stored in memory, it is important to realize that the symmetry in the filter phases can lead to reduction of memory requirements. In our example of linear interpolation, phase 0 and 32 is unique, but all other phases appear in "flipped pairs" as an result for symmetry. As an example phase 1 is a flipped version of phase 63. Following this, we can reduce the required storage requirements by 33/64 with the help of some additional mapping logic. Table 1 could be shortened to Table 2.

Table 2: Simplified version of Table 1 based on kernel symmetry

Phase	*Tap 1*	*Tap 2*
phase 0	256	0
phase 1	252	4
phase 2	248	8
⋮	⋮	⋮
phase 32	128	128

Phase	*Tap 1*	*Tap 2*
phase 0	h[32]	h[0]
phase 1	h[33]	h[1]
phase 2	h[34]	h[2]
⋮	⋮	⋮
phase 32	h[65]	h[32]

Order of operations

In scaling-processes where each direction is individually processed, the order of which direction is scaled can be crucial to the quality of the result. The pixel values will be stored with limited precision after each step, which implies an introduction of error. This error could potentially be amplified through the next step in the process. [Ber03] places the stage which introduces the most error, as close to the end of the processing chain as possible. So if horizontal rescaling introduces larger error than vertical rescaling, then this stage should be placed near or at the end.

5.4. Higher order interpolation in FIR filter implementation

Section 3.1 showed us that the ideal interpolation FIR filter was a lowpass filter based on the sinc-function in the time-domain. [Ber03] gives the following algorithm to compute all taps in such a filter.

1. Assign P to the number of phases and T to the number of taps per phase.
2. Sample $P \cdot T$ samples from the $sinc(\frac{\pi t}{P})$ function by sampling it at integer values symmetrical to the origin. The last sample will have no symmetry.
3. Multiply the samples with a chosen window with the same amount of samples.

4. Split them into T phases
5. Assign the number of k bits of coefficient precision and multiply each phase by $P \cdot 2^k$
6. Assign each tap to the closest integer and check that the sum of each phase is 2^k
7. If the sum is not 2^k, then modify some of the taps such that the sum becomes 2^k

This algorithm can also be shown visually (as we did with the linear kernel) in Figure 28. In this case we calculate coefficients for a 5-tap 16-phase polyphase FIR filter based on the windowed sinc function.

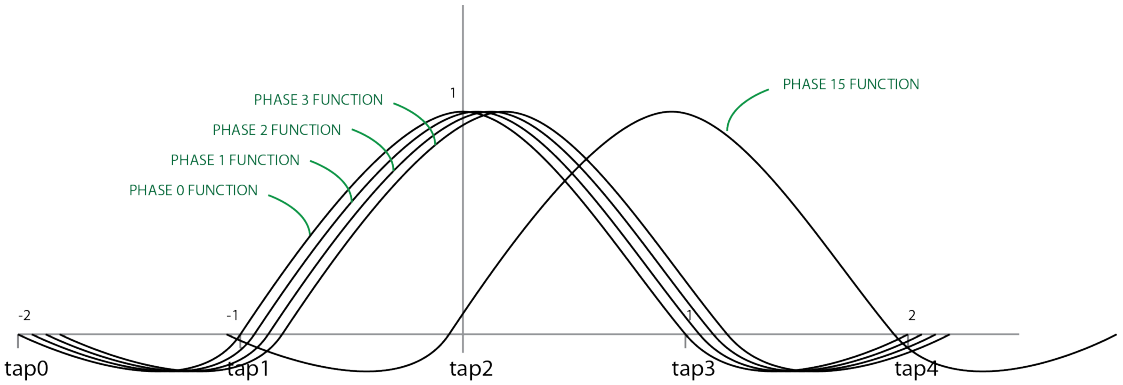
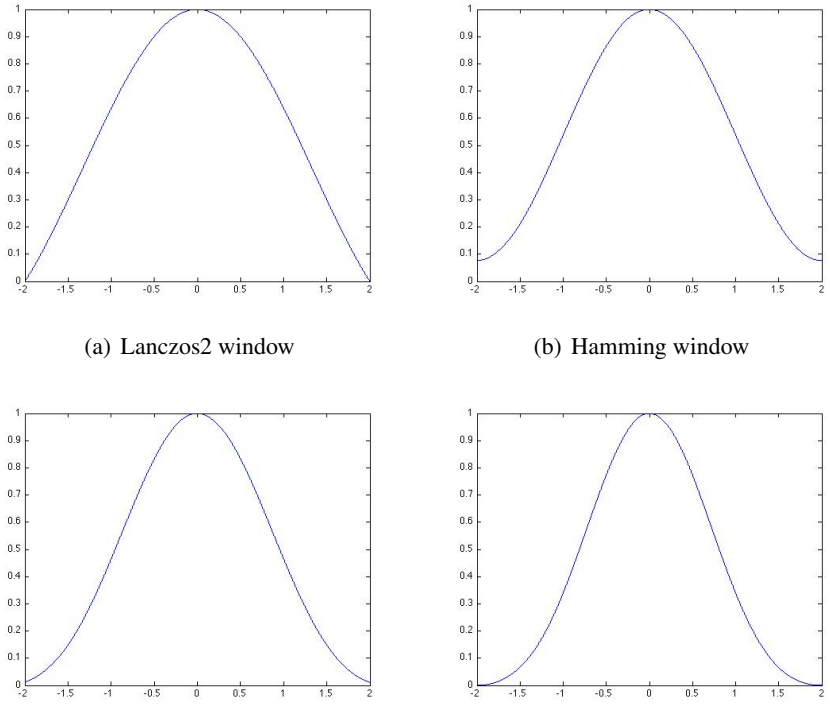


Figure 28: Coefficient generation for a 5-tap 15-phase sinc-based interpolation FIR filter

These computations can either be calculated at run-time by a processor or precalculated before hardware implementation and stored in a shared memory. MATLAB supports filter-generation with the command **Coeff = fir1(193, 1/64, ´Hamming´)**. It should be mentioned that if the number of taps per phase is even, phase 0 will be aligned with an existing input sample, and in the case of odd number of phases, phase 0 will fall exactly in the middle of two existing phases. [Ber03] recommends selecting odd number of taps, even if an even number is needed. This is done because of the symmetry of the sinc() function. The last tap is just ignored to obtain an even number of taps.

It is difficult to say that a windowing function is better than the next, because this varies between applications. [zip09] recommend designers to experiment with different windows for their application to decide which window gives best performance. The Lanczos2-windowed sinc (discussed in Section 3.7) is often used and known to give good performance. Other known windows are Hamming, Kaiser and Blackman, shown in Figure 29. [zip09] has developed a Matlab script for generating windowing functions and a script to calculate coefficients for the Lanczos2-windowed sinc function with 5 taps and 16 phases per tap. These scripts can be found in Appendix B.1 and B.2.



(a) Lanczos2 window

(b) Hamming window

(c) Kaiser window

(d) Blackman window

Figure 29: Commonly used windowing functions

6. Cisco Reference Scaler

Cisco has provided a reference scaler used today in their hardware solutions. The scaler was given as a functional runnable Matlab model. Although the algorithm of choice cannot be given, typical resource requirements and maximum performance are given in Table 3. This implementation does not include coefficient calculations, as these are performed in software by a processor and stored in a shared memory.

Table 3: Typical resource requirements and performance

FPGA	Altera CycloneIII EP3C120F780C7
Logic Cells	2019
Dedicated Logic Registers	1426
Memory Bits	81920
M9Ks	10
DSP Elements	8
DSP 9x9	8
LUT-Only LCs	559
Register-Only LCs	303
LUT/Register LCs	1157
Max Frequency	162.73MHz

7. Winscale

The simplest form of area-pixel based algorithm and architecture is presented in this section. A MATLAB model is coded and placed in Appendix B.3 - B.5

7.1. Algorithm

[CHK03] utilizes the area pixel model presented in Section 2.2 in designing the Winscale algorithm. The horizontal scale ratio (HSR) may be different from the vertical scale ratio (VSR). For upscaling the scale ratio (SR) is above 1.0 and below 1.0 for downscaling and the scaling is performed line by line in the Target-to-Source manner (Section 2.1).

Upscaling

In the case of upscaling the target pixel may overlap one, two or maximum four source pixels according to the pixel area model. The pixel intensity of these pixels are defined by $C0$-$C3$ and the area of each source pixel is 1.0. All the boundary coordinates of the 2x2 source pixels can be calculated from the pixel width/height and the boundary coordinate (C_{0BX}, C_{0BY}). (C_{0BX}, C_{0BY}) is initially set to $(0, 0)$ as we scale according to the direction of the streaming video data: line by line, left to right and top to bottom.

The sum of the overlapping regions is called *the filter window* and is defined in size of area by $A0$-$A3$. $winW$ and $winH$ are the window height and width and is only dependent on scale ratio according to Equation 34). $(winX, winY)$ and (P_{BX}, P_{BY}) is the window's center coordinate and upper left corner coordinate.

$$\begin{aligned} winW &= \frac{1}{HSR} \\ winH &= \frac{1}{VSR} \\ SR = \frac{1}{winW \cdot winH} &= \frac{1}{\text{filter window area}} \end{aligned} \tag{34}$$

dL and dT are the width and height of the overlapping area of $C0$ and is defined by Equation 35 .

$$\begin{aligned} dL &= (C_{0BX} + 1) - P_{BX} \\ dT &= (C_{0BY} + 1) - P_{BY} \end{aligned} \tag{35}$$

These parameters are used in calculation of $A0$-$A3$ (Equation 36).

$$\begin{aligned} A0 &= dL \cdot dT \\ A1 &= dL \cdot (winH - dT) \\ A2 &= (winW - dL) \cdot dT \\ A3 &= (winW - dL) \cdot (winH - dT) \end{aligned} \tag{36}$$

The relationship between all these parameters is shown in Figure 30 and the pixel intensity of the target pixel can be calculated by Equation 37.

$$P = SR \cdot (A0 \cdot C0 + A1 \cdot C1 + A2 \cdot C2 + A3 \cdot C3) \tag{37}$$

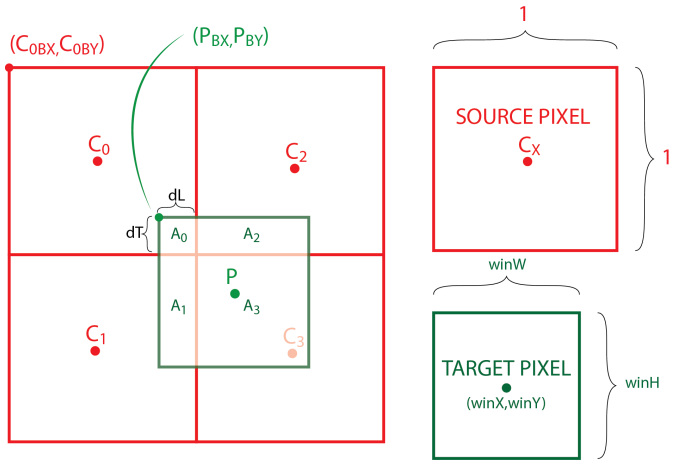


Figure 30: Parameter definitions when upscaling with Winscale

Downscaling

When downscaling with a factor bigger than two, the target pixel will overlap a bigger area than a 2x2 source pixel area. As the Winscale algorithm only operates on a maximum of 2x2 source pixels, it is therefor necessary to prescale the image with a simpler algorithm. [CHK03] purposes a prescaler which performs an *evenly weighted summation with two's power ratio* to prescale the image and reduce the downscaling factor between 1 and $1/2$.

7.2. Implementation Statistics

Both [CHK03] and [CcL07] have each done their own implementation of the Winscale algorithm; [CHK03] on an FPGA and [CcL07] on an ASIC. [CcL07] uses Table 4 to compare the two. It is not fair to compare the two, as the FPGA implementation includes the prescaler and the ASIC does not.

Table 4: Implementation details for Winscale

Author	[CHK03]	[CcL07]
Technology	Unknown FPGA	UMC 1P6M 0.18 μm (ASIC)
Line Buffer	1	1
Gate Count	29 000	17 414
Max Frequency	64 MHz	130.24 MHz

7.3. Hardware Architecture

[CcL07] purposes the following hardware architecture for implementation of the Winscale Algorithm. The prescaler is left out of the architecture depicted in Figure 31 as we assume that if needed, pre-scaling already have been perform in a former pipelined step. The system architecture presented here, is therefore only utilized for scaling factors between $+\infty$ and 0.5.

Coordinate Accumulator

The *coordinate accumulator* calculates the coordinates of the window, or in other words, the coordinates $(next\_i, next\_j)$ of the next target pixel to be calculated. Calculations are done

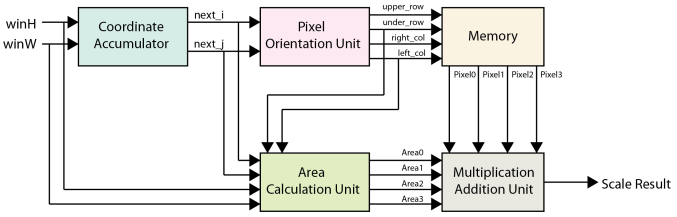


Figure 31: Winscale Hardware Architecture

with a simple add and accumulate process; by adding the window width ($winW$) or height ($winH$) with every horizontal or vertical movement. Equation 38 is realized by the structure shown in Figure 32.

$$\begin{aligned} next\_j(0) &= 0 \\ next\_j(m+1) &= next\_j(m) + winH \text{ for m = 0,1,2,...} \\ next\_i(0) &= 0 \\ next\_i(n+1) &= next\_i(n) + winW \text{ for n = 0,1,2,...} \end{aligned} \tag{38}$$

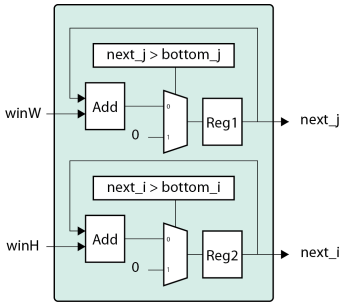


Figure 32: Coordinate Accumulator

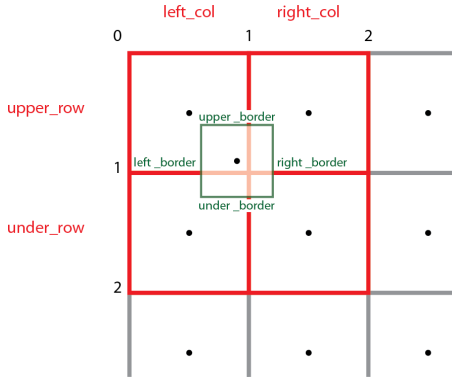


Figure 33: Coordinate column and row definitions

Pixel Orientation Unit

The *pixel orientation unit* calculates which source pixels the window overlaps. These are the four source pixels the new target pixel should be based on in its intensity calculations. Boundary

coordinates of the window are determined according by Equation 39 and source pixel rows and columns by ceiling and floor functions in Equation 40. These parameters are used either to address a memory containing all the pixel intensities or as control signals to a linestore memory containing parts of a streaming image.

$$\begin{aligned} upper\_border &= P_{BY} \\ under\_border &= P_{BY} + winH \\ left\_border &= P_{BX} \\ right\_border &= P_{BX} + winW \end{aligned} \tag{39}$$

$$\begin{aligned} upper\_row &= \lfloor upper\_border \rfloor + 1 \\ under\_row &= \lceil under\_border \rceil \\ left\_col &= \lfloor left\_border \rfloor + 1 \\ right\_col &= \lceil right\_border \rceil \end{aligned} \tag{40}$$

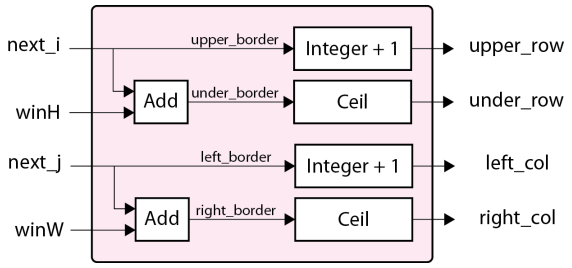


Figure 34: Pixel Orientation Unit

Area Calculation Unit

The *area calculation unit* calculates how much overlap the window has in each of the four source pixels. A challenge arises in upscaling with the target pixel being smaller than a source pixel and the target pixel only overlaps one source pixel. In some of these cases, dL and dT may be calculated larger than $winW$ and $winH$ resulting in larger area weights than the actual overlap. dL and dT should therefor be limited to a maximum of winW and winH (Equation 41). Overlap areas (Equation 36) are calculated by the structure in Figure 35.

$$\begin{aligned} dL &= min[upper\_row - next\_i, winW] \\ dT &= min[left\_row - next\_j, winH] \end{aligned} \tag{41}$$

Multiplication-Addition Unit

The *multiplication-addition unit* (Figure 36) is the actual scaling filter, calculating the new target pixel intensity (Equation 37) based on the window overlap and the four source pixel intensities. If we have a full overlap of a source pixel ($dT = winH$ and $dL = winW$) will the module assign the intensity of the $Pixel0$ to the new pixel.

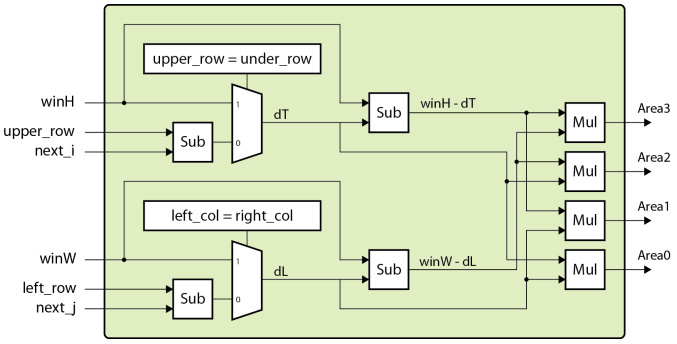


Figure 35: Area Calculation Unit

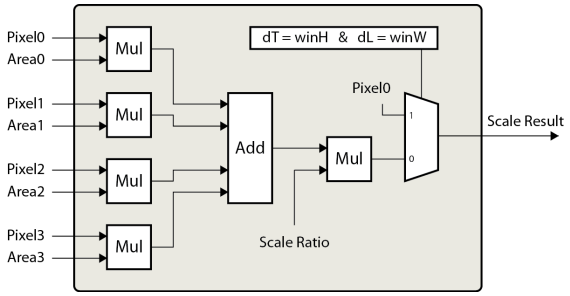


Figure 36: Multiplication-Addition Unit

7.4. Winscale Summary

[CcL07] claims that the Winscale method has a good high frequency response characteristics and better image quality than the bilinear method. It preserves edge characteristics of an image well, can handle streaming data directly and requires only a small amount of memory.

The Winscale algorithm is said to have *Nearest Neighbor properties* (good high frequency response and blur reduction) when the target pixel only overlaps one of the source pixel, and *linear properties* (blurring is added to avoid aliasing) when it overlaps multiple pixels.

8. Egde-Oriented Image Scaling Processor

A area-pixel based, edge-aware algorithm named Edgeprocessor is described in the following section. The mathematics of the algorithm is described in such detail that future coding of a matlab model or implementation on an FPGA would be done easily. An architecture is described and implementation statistics from the referred implementation is presented.

[PYC09] purposes an algorithm and a hardware architecture in the article "VLSI Implementation of an Edge-Oriented Image Scaling Processor". The algorithm is not mentioned by a specific name in the article, so we will call it *the edge processor algorithm*. It's in many ways very similar to the Winscale algortihm, in that both use a area pixel model rather than the point pixel model. The edge processor can be viewed as a further developed and more complex winscale algorithm and implementation. The main difference between the Winscale algorithm and the edgeprocessor algorithm is that the edgeprocessor performs a simple edge detection and utilize the results to preserve a good edge/high-frequency response.

8.1. Algorithm

It is first necessary to define some new variables and rename other variables already known form Winscale. The target image is a $SW \times SH$ pixel image fed to the scaler in a row by row order, with the upper left pixel as the first to arrive in each frame. Each source pixel is treated as a rectangle of height and width (s_h, s_w). The target image has the resolution $TW \times TH$ where pixel height and width still are defined as $(winH, winW)$. The coordinates of the current target pixel being calculated is (k, l). $mf\_w = \frac{TW}{SW}$ and $mf\_h = \frac{TH}{SH}$ defines the horizontal and vertical magnification factor. All these parameters are shown in Figure 37

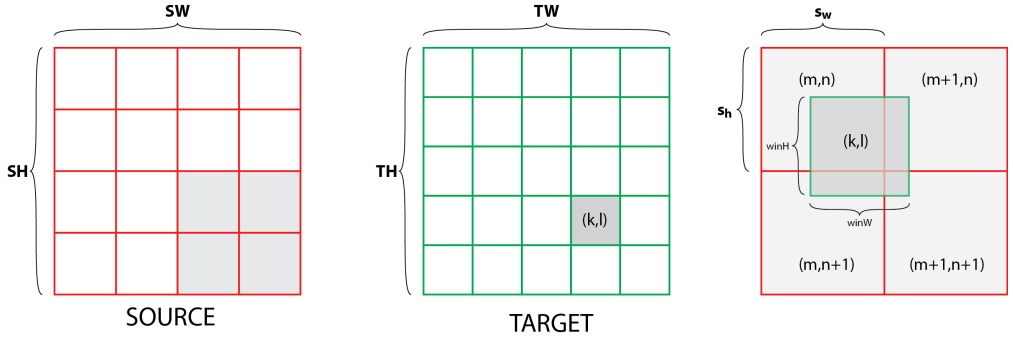


Figure 37: Source and target pixel definitions

The edge processor calculates the intensity $\hat{F}_T(k, l)$ of the new target pixel (k, l) similar to the Winscale algorithm, by a weighted average of the $2x2$ source pixel intensities. Winscale's Equation 37 is rewritten as Equation 42

$$\hat{F}_T(k,l) = \sum_{i=0}^{1}\sum_{j=0}^{1} F_S(m+i, n+j) \cdot W(m+i, n+j) \tag{42}$$

where $F_S(m, n)$ and $W(m, n)$ defines the pixel intensity and the weight factors of source pixel (m, n). The weighting factors corresponds to the portion of the total window area (A_{sum}) each of the overlapping areas accounts for. The areas are given by Equation 44 and shown in Figure

38. The weighting factors are given by Equation 43.

$$\begin{aligned} A_{sum} &= A(m,n) + A(m+1,n) + A(m,n+1) + A(m+1,n+1) \\ A_{sum} &= winW \cdot winH \\ W(m+i,n+j) &= \frac{A(m+i,n+j)}{A_{sum}} \end{aligned} \tag{43}$$

$$\begin{aligned} A(m,n) &= left(k,l) \cdot top(k,l) \\ A(m+1,n) &= right(k,l) \cdot top(k,l) \\ A(m,n+1) &= left(k,l) \cdot bottom(k,l) \\ A(m+1,n+1) &= right(k,l) \cdot bottom(k,l) \end{aligned} \tag{44}$$

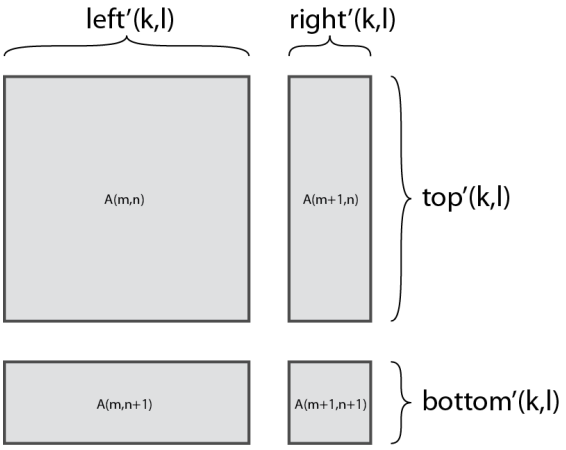


Figure 38: Edge Processor window overlap areas

8.1.1. Approximation - Appr[]

Implementation of the area-pixel model could potentially result in lots of floating point operations. Those operations often becomes the most complex and expensive modules to realize in these kind of designs. Instead [PYC09] proposes to use a approximation method $Appr[]$, to limit the bit-width or precision of these calculations. The approximated lengths (6-bit int) of the regions are thereby defined by Equation 45 and the approximated size of the overlapping areas defined by Equation 46.

$$\begin{aligned} left'(k,l) &= Appr[left(k,l)] \\ right'(k,l) &= Appr[right(k,l)] \\ top'(k,l) &= Appr[top(k,l)] \\ bottom'(k,l) &= Appr[bottom(k,l)] \end{aligned} \tag{45}$$

$$\begin{aligned} A'(m,n) &= left'(k,l) \cdot top'(k,l) \\ A'(m+1,n) &= right'(k,l) \cdot top'(k,l) \\ A'(m,n+1) &= left'(k,l) \cdot bottom'(k,l) \\ A'(m+1,n+1) &= right'(k,l) \cdot bottom'(k,l) \end{aligned} \tag{46}$$

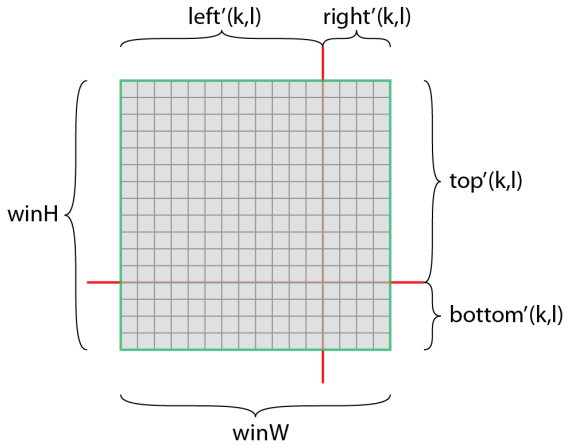


Figure 39: Window parameters definitions

Filter Window Parameters

The target pixel is defined as a grid with dimensions dependent on the magnification factors mf_w according to Equation 47. $winH$ can be calculated in the same way by changing mf_w to mf_h. This results in reduction of complexity in computing Equation 43, from a full division operator to a simple shift operator.

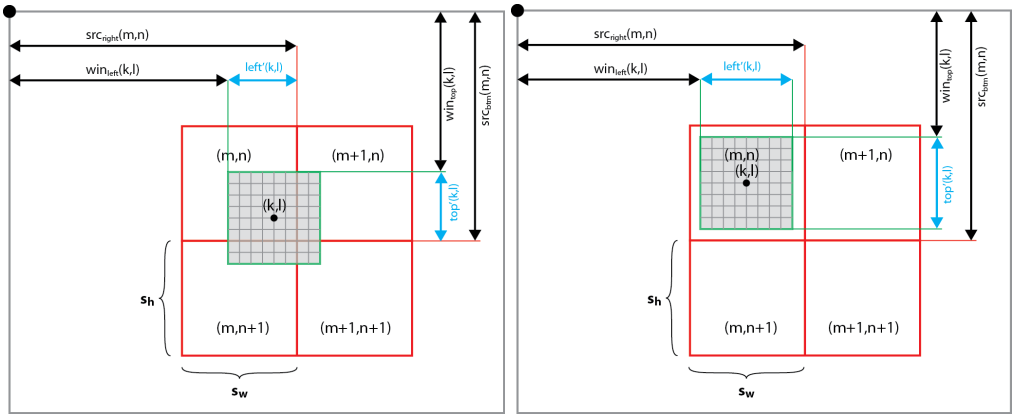
$$winW = \begin{cases} & \uparrow \\ 2^{n-2} & \text{for} \quad 25\% \leq mf\_w < 50\% \\ 2^{n-1} & \text{for} \quad 50\% \leq mf\_w < 100\% \\ 2^{n} & \text{for} \quad 100\% \leq mf\_w < 200\% \\ 2^{n+1} & \text{for} \quad 200\% \leq mf\_w < 400\% \\ & \downarrow \end{cases} \tag{47}$$

The variables $left'(k,l)$ and $top'(k,l)$ corresponds to Winscale's definition of dL and dT and is defined as Equation 48. As shown in Figure 40 are $left'(k,l)$ and $top'(k,l)$ dependent on whether the overlap is partial or full. The maximal values are $(winW, winH)$ at full overlap.

$$\begin{aligned} left'(k,l) &= min[src_{right}(m,n) - win_{left}, winW] \\ top'(k,l) &= min[src_{btm}(m,n) - win_{top}, winW] \end{aligned} \tag{48}$$

win_{left} and win_{right} corresponds to $next\_i$ and $next\_j$ definitions in Winscale, although initial values are different. Winscale aligned *upper left corner* of the first source and target pixel, while Edgeprocessor aligns the *centers* of the same pixels. Window corner coordinates (win_left, win_top) are therefor calculated by Equation 49.

$$\begin{aligned} win_{left}(0,0) &= \frac{s_w - winW}{2} \\ win_{left}(k,l) &= win_{left}(k-1,l) + winW \\ win_{top}(0,0) &= \frac{s_h - winH}{2} \\ win_{top}(k,l) &= win_{top}(k,l-1) + winH \end{aligned} \tag{49}$$



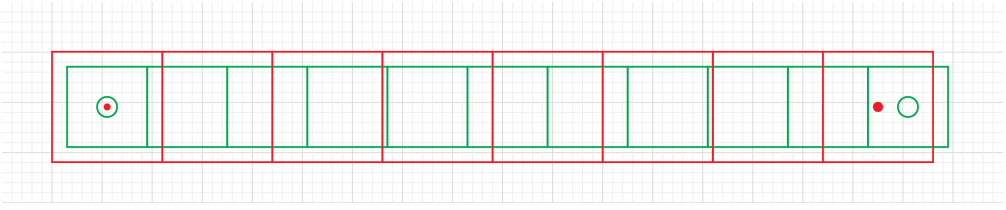
(a) Partial Overlap (b) Full Overlap

Figure 40: Window overlap

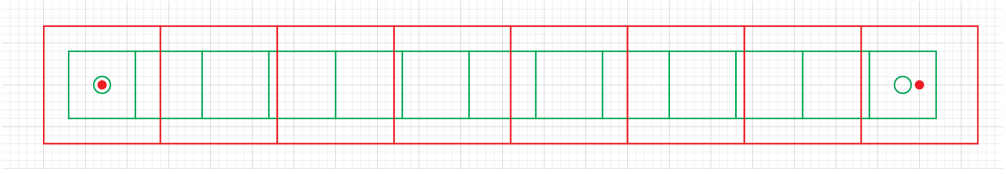
Compensation of rounding error in (s_w, s_h) calculations

The source pixel width and height (s_w, s_h) are dependent on source and target image resolutions (Equation 50). Since (s_w, s_h) are rounded to closest integer, a rounding error is introduced. To visualize the effect of the error, two examples will be shown. In the first example a source image (8x8) is upscaled to a target image (11x11). When upper left corner pixels are aligned at centers and s_w is *rounded down* to 11 according to 50, Figure 41(a) show that right corner pixels don't align at centers. Some of the target pixels need a negative offset (to the left). The opposite can be observed as an second example where a source image 8x8 is upscaled to the target image (13x13). Since s_w is *rounded up* to 14, some of the pixels need a positive offset (to the right). Which pixels and how much offset is not known at this point.

$$
\begin{aligned}
s_w &= round\left[\frac{TW-1}{SW-1}\cdot 2^n\right] \\
s_h &= round\left[\frac{TH-1}{SH-1}\cdot 2^n\right]
\end{aligned}
\tag{50}
$$



(a) Example 1: Upscaling $8x8 \rightarrow 11x11$



(b) Example 2: Upscaling $8x8 \rightarrow 13x13$

Figure 41: Rounding Error

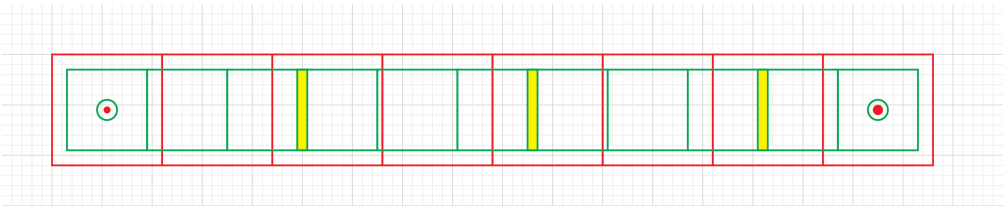
To compensate for the error a τ_w or τ_h is added when calculating the center coordinates of the 2x2 source pixel cluster (src_{right}, src_{btm}) (Equation 51).

$$\begin{aligned} src_{right}(0,0) &= s_w \\ src_{right}(m,n) &= src_{right}(m-1,n) + s_w + \tau_w \\ src_{btm}(0,0) &= s_h \\ src_{btm}(m,n) &= src_{btm}(m,n-1) + s_h + \tau_h \end{aligned} \tag{51}$$

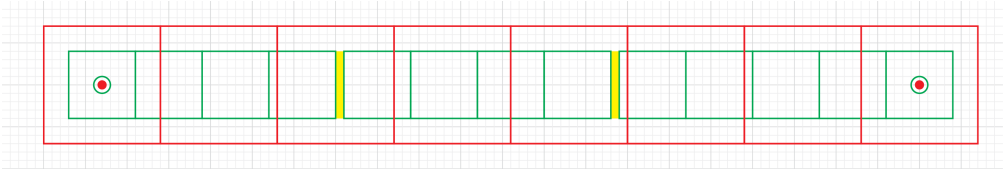
τ_w and τ_h are determined according to two different modes; *normal mode* and *regulating mode*. If the distance between the source and target center (far right) is less than 1, then the processor is in *normal mode*, and τ_w is set to zero. The processor enters *regulating mode* if the distance between the pixel centers are more than 1. In this mode will the processor shift the window position ($top'(k,l)$ and $top'(k,l)$) by setting τ_w to either -1 (compensating for rounding down error) or 1 (compensating for rounding up error) for some of the pixels along the row. The number of pixels to be regulated are r_w (Equation 52). $r_w = 3$ in the first example, and $r_w = 2$ in the second, meaning three pixels will get regulated with $\tau_w = -1$ in the first and two pixels with $\tau_w = 1$ in the second. The pixels are chosen at regular intervals through the row. The result is shown in Figure 42. The error of the rounding is not removed, it's just distributed throughout the image rather than experiencing scaling artifacts in the right side of the image.

$$r_w = \begin{cases} 2^n \cdot (TW-1) - s_w \cdot (SW-1) & \text{if } s_w \text{ is rounded down to an integer} \\ s_w \cdot (SW-1) - 2^n \cdot (TW-1) & \text{if } s_w \text{ is rounded up to an integer} \end{cases} \tag{52}$$

τ_h is set the same way as τ_w in the vertical direction.



(a) Example 1: Three pixels shifted left



(b) Example 2: Two pixels shifted right

Figure 42: Compensation for Rounding Error

8.1.2. Edge-Catching

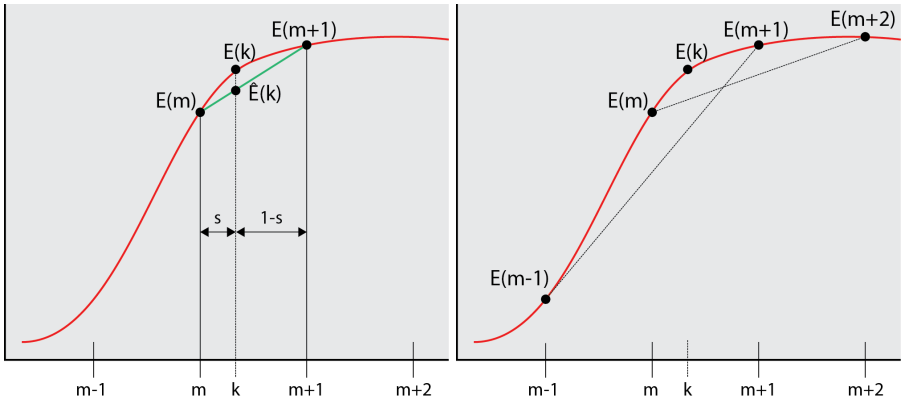
Figure 43(a) shows how linear interpolation may result in reduced edge quality. $\hat{E}(k)$ (from Equation 53) represents the estimated luminance value of the interpolated pixel (using linear interpolation), based on the neighboring pixels $E(m)$ and $E(m+1)$.

$$\hat{E}(k) = (1-s) \cdot E(m) + s \cdot E(m+1) \tag{53}$$

$\hat{E}(k)$ will clearly in many cases deviate from the ideal value $E(k)$. To reduce the error, a simple edge detection functionality is implemented in the Edgeprocessor. The edge is detected using an edge-describing parameter L calculated from Equation 54. Utilizing the luminance of the surrounding four pixels $E(m-1)$, $E(m)$, $E(m+1)$ and $E(m+2)$ (shown in Figure 43(b)), we are able to determine which side of the pixel is more homogenous, and therefore should affect the interpolated pixel more. The improvements are done by adjusting the s parameter either left or right.

$$L = |E(m+1) - E(m-1)| - |E(m+2) - E(m)| \tag{54}$$

The edge-parameter L is interpreted by three different cases:



(a) Reduced edge sharpness from linear interpolation

(b) Edge detection based on four surrounding pixels

Figure 43: Edge detection

- L = 0 indicates that the two sides of pixel k is symmetrical and s therefor remains unchanged.
- L > 0 indicates that there are a bigger change in pixel intensity between E(m+1) and E(m-1) compared to the change between E(m) and E(m+2). This indicates that the right side is more homogenous and more important. An increased s will in this case result in a more correct interpolated value.
- L < 0 indicates the opposite of the preceding case: the left side is more homogenous and more important. A reduces s will result in a better estimation.

The adjustment of s is calculated by Equation 55.

$$s_{new} = \begin{cases} s_{old} + L \cdot \frac{(1-s)}{2^8} & \text{if} \quad L \geq 0 \\ s_{old} + L \cdot \frac{s}{2^8} & \text{if} \quad L < 0 \end{cases} \tag{55}$$

In the case of our Edgeprocessor, we base our interpolation on a maximum of two rows of pixels. The size of the overlapping regions indicates whether we should base our edge detection on the upper or lower rows of pixels. If $top'(k,l)$ is larger than $\frac{winH}{2}$, the upper row dominates the overlap and is therefore used in edge detection. The adjustment parameters L_A and A_C is calculated from Equation 56 and 57.

$$L_A = |F_S(m+1,n) - F_S(m-1,n)| - |F_S(m+2,n) - F_S(m,n)| \tag{56}$$

$$A_C = \begin{cases} A'(m,n) & \text{if} \quad L_A \geq 0 \\ A'(m+1,n) & \text{if} \quad L_A < 0 \end{cases} \tag{57}$$

The size of the overlapping areas can finally be calculated from Equation 58.

$$\begin{aligned} A''(m,n) &= A'(m,n) - L_A \cdot \frac{A_C}{2^8} \\ A''(m+1,n) &= A'(m+1,n) + L_A \cdot \frac{A_C}{2^8} \\ A''(m,n+1) &= A'(m,n+1) \\ A''(m+1,n+1) &= A'(m+1,n+1) \end{aligned} \tag{58}$$

If the opposite is true, $top'(k,l)$ being smaller than $\frac{winH}{2}$, than the lower row is used. The adjustment parameters L_A and A_C is calculated from Equation 59 and 60, and the size of the overlapping areas changed to Equation 61.

$$L_A = |F_S(m+1,n+1) - F_S(m-1,n+1)| - |F_S(m+2,n+1) - F_S(m,n+1)| \tag{59}$$

$$A_C = \begin{cases} A'(m,n+1) & \text{if} \quad L_A \geq 0 \\ A'(m+1,n+1) & \text{if} \quad L_A < 0 \end{cases} \tag{60}$$

$$\begin{aligned} A''(m,n) &= A'(m,n) \\ A''(m+1,n) &= A'(m+1,n) \\ A''(m,n+1) &= A'(m,n+1) \ - L_A \cdot \frac{A_C}{2^8} \\ A''(m+1,n+1) &= A'(m+1,n+1) + L_A \cdot \frac{A_C}{2^8} \end{aligned} \tag{61}$$

8.2. Hardware Architecture

A pipeline structure is a natural way of implementing systems processing streaming data. The choice of architecture results in better utilization of operators and potentially increase of maximal clock frequency. The Edgeprocessor is implemented as a 7-stage pipeline architecture (shown in Figure 44) consisting of the blocks *Controller*, *Approximate Module* (AM), *Register Bank*, *Area Generator* (AG), *Edge Catcher* (EC), *Area Tuner* (AT) and *Target Generator* (TG). Pipeline registers marked P are scattered throughout the pipeline to store intermediate parameters. [PYC09] claims this VLSI implementation of the edgeprocessor supports any magnification factor mf between $\frac{1}{64}$ and 64 with n set to 3 in Equation 47.

Approximate Module

The approximate module calculates the overlapping parameters $left'(k,l)$, $right'(k,l)$, $top'(k,l)$ and $bottom'(k,l)$ based on the input frame dimensions ($SH \times SW$) and the output frame dimensions ($TH \times TW$). These calculations are done according to Equation 48 - 52. This module is not shown in any detailed figure because of it's complexity.

Register Bank

Since each new pixel potentially could be based on pixels from two successive rows, a need for temporary storage arises. From Equation 42: $F_S(m,n)$, $F_S(m+1,n)$, $F_S(m,n+1)$ and $F_S(m+1,n+1)$ is needed. In addition, from Equation 56 and 59: $F_S(m-1,n)$, $F_S(m-1,n+)$,

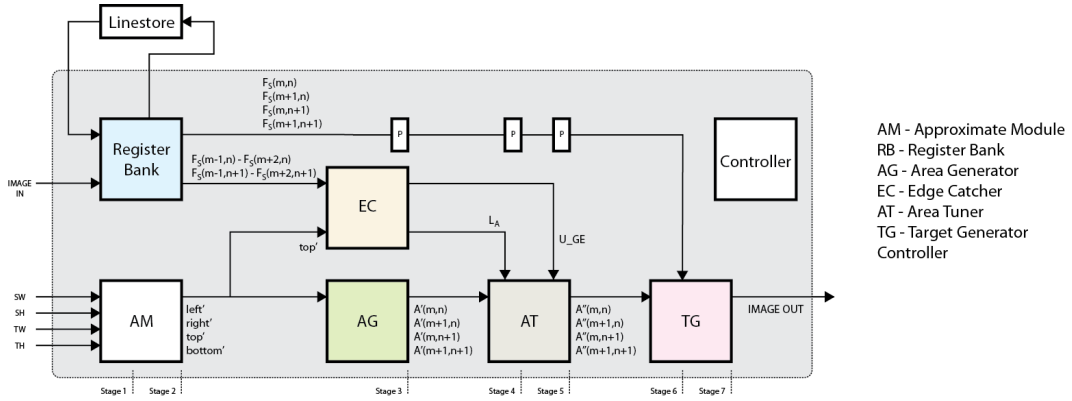
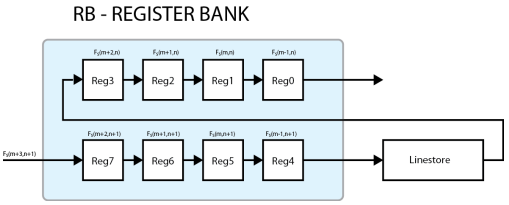
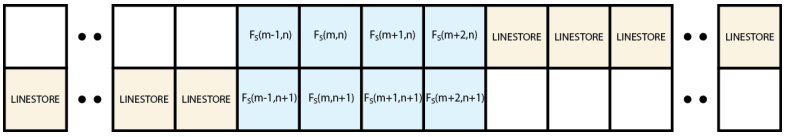


Figure 44: Edge Processor hardware architecture

$F_S(m+2,n)$ and $F_S(m+2,n)$ is needed. These luminance values are stored in the register bank module, implemented as a shiftregister. As shown in Figure 45(b) and 45(a), the pixel luminance values are stored in two pixel groups of four and shifted through a shift register (linestore), until they are needed again.



(a) Register Bank Architeture



(b) Where each pixel is stored

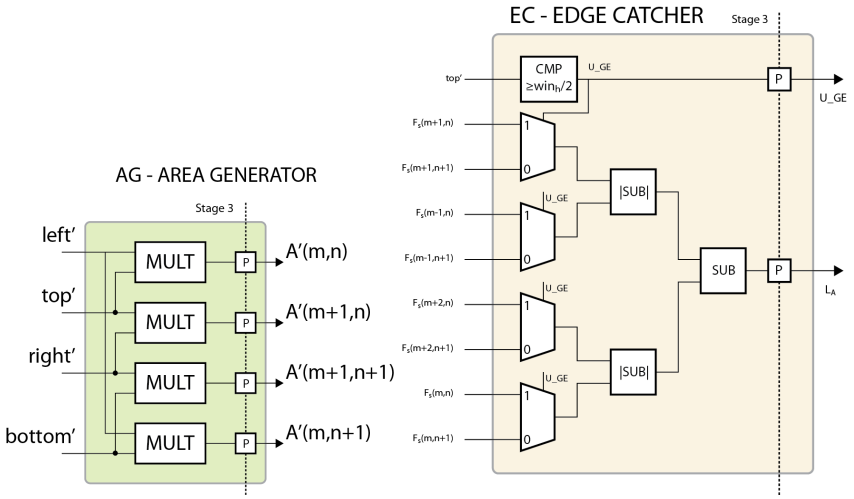
Figure 45: Register Bank and Linestore

Area Generator

The area generator calculates the overlapping areas according to Equation 46 using four (4x4) integer multiplicators.

Edge Catcher

The edge catcher performs the edge detection presented in Section 8.1.2 by calculating L_A from Equation 59 and 56. Through a series of multiplexers, the most dominant pixel row are chosen to produce L_A. The comparison result U_GE indicates whether upper ($U\_GE = 1$) or lower ($U\_GE = 0$) row of pixels are used in the edge detection.



(a) Area Generator (b) Edge Catcher

Figure 46: Area Tuner and Edge Catcher

Area Tuner

The area tuner utilizes the comparison results L_A and U_GE from the edge catcher to adjust overlapping areas according to Equation 58 and 61. This module is the final step in the process of calculating the filter coefficients used in the target generator.

Target Generator

The target generator is the actual scaling filter. This is the module where all the relevant parameters end up to calculate the estimated luminance value for the new target pixel. Through 4 MULT, 3 ADD units and a shifter are Equation 42 finally being calculated. It is this module which is simplified severely by replacing a divisor unit with a shifter.

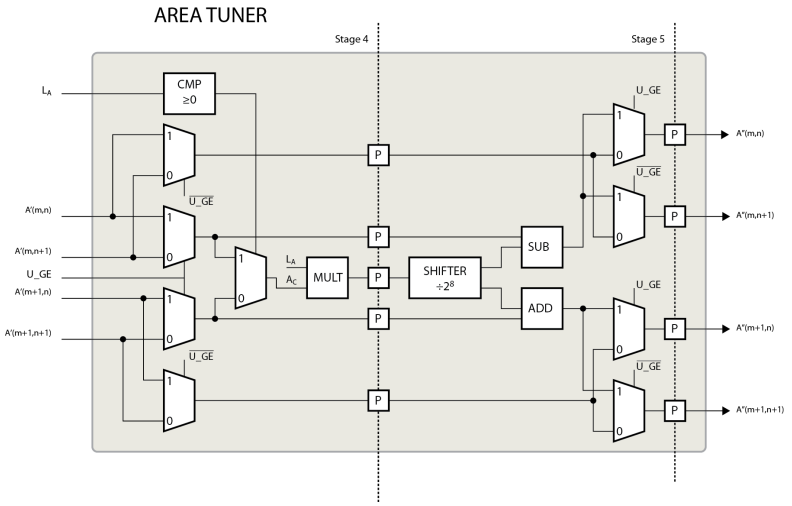
Controller

The controller sends all the required control signals to all the modules throughout operation. At any time, computation for 7 different target pixel be placed in the pipeline, all in different stages of computation. A calculation of a single target pixel requires seven clock cycles, but since the calculations is parallelized by a pipeline, it will complete *one new target pixel value every clock cycle*.

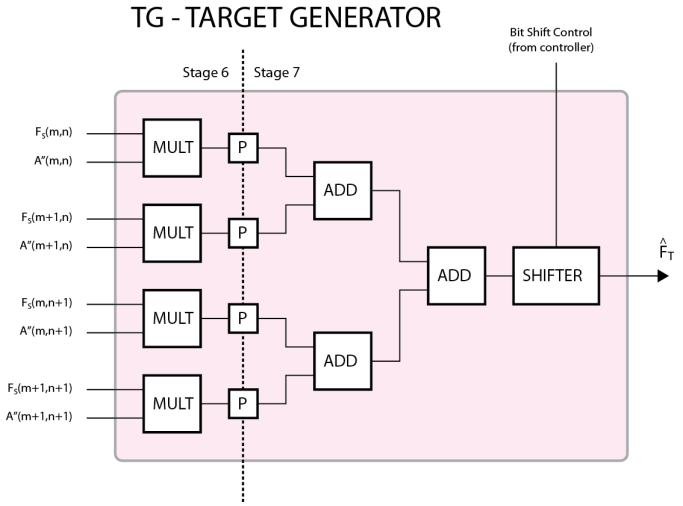
8.3. Simulation and implementation

[PYC09] has performed computational complexity evaluation, quantitative and visual quality comparisons of the Edgeprocessor, nearest neighbor, bilinear, bicubic and winscale. The computational complexity was compared by implementing the algorithms in C on two different PC architectures. By upscaling an image from $400x400$ to $512x512$ pixels, the results clearly showed that the edgeprocessor algorithm required less time than the other area-pixel based algorithms and bicubic.

To get a quantitative comparison of the algorithms, a series of test images were first downscaled/upscaled with a common algorithm then upscaled/downscaled to the original size with



(a) Area Tuner



(b) Target Generator

Figure 47: Area tuner and Target Generator

the different algorithms. The quantitative evaluation were then based on the peak signal-to-noise ratio (PSNR) between the original image and the regained image. Three test was done with the source image ($512x512$): downscaling to $400x400$, downscaling to $256x256$ and upscaling to $600x600$. It was concluded that the edgeprocessor performed better than the other low-complexity algorithms, but the degree of improvement were dependent on the image content.

The visual quality were evaluated by upscaling a picture of a text string with very sharp edges. As predicted would the nearest neighbor model produce a blocky image, while the bilinear model produce a blurry image. The important result from this test was that the edgeprocessor delivered better visual quality then winscale, with near-bicubic quality at lower computational complexity.

The edgeprocessor were implemented on three different technologies; two FPGAs and one ASIC. The performance and area consumption is summarized in Table 5.

Table 5: Implementation details for Edgeprocessor

FPGA	Xilinx Virtex-II Pro XC2VP50	Altera CycloneII EP2C8F256C6	TCMC's 018 μm process (ASIC)
Line Buffer	1	1	1
Area	581 CLB	1.06K logic elements	10.4K gate counts
Max Frequency	142 MHz	109 MHz	200MHz

9. Evaluating Winscale, Edgeprocessor and reference scaler

The computational complexity evaluation of edgeprocessor referred to in Section 8.3 was based on this single upscaling experiment and the quantitative quality is only evaluated in simulations with the magnification factors $\frac{400}{512}$, $\frac{600}{512}$ and $\frac{1}{2}$. None of these tests required a prescaler. As discussed in Section 7.1, winscale algorithm requires a prescaler when scaling with a factor lower than $\frac{1}{2}$. This would lengthen the computational time depending on the chosen prescaler. It is not clear from [PYC09] whether the edge processor requires an prescaler. As I see it, the edgeprocessor would require a prescaler for magnification factors lower than $\frac{1}{2}$, as it is a more advanced development of the winscale algorithm, but based on the same principles of $2x2$ pixel window-overlapping.

The visual quality was evaluated using only one image of a text string. These type of non-natural images consists of large homogenous areas with few edges. Natural images should have been included in the test, as the videoscaler should provide high quality scaling for both natural and non-natural images. The natural images would expose aliasing effects caused by scaling of areas with large amount of high frequency information.

The implementation resource requirements and maximal performanxe is summarized in Table 6. These are still difficult to compare, as they are implemented on different FPGA families. We may assume that the Edge Processor would require considerable less resources on a common FPGA chip, compared to the reference scaler.

Table 6: Implementation comparison

Algorithm	Reference Scaler	Winscale	Edge Processor	Edge Processor	Edge Processor
FPGA	Altera CycloneIII EP3C120 F780C7	NA	Xilinx Virtex-II Pro XC2VP50	Altera CycloneII EP2C8 F256C6	TCMC's 018 μm process (ASIC)
Line Buffer	N/A	1	1	1	1
Area	2019 logic cells	29K gates	581 CLB	1.06K logic elements	10.4K gate counts
Max Frequency	162.73MHz	65 MHz	142 MHz	109 MHz	200MHz

9.1. My Matlab Model Comparisons

The visual quality of 6 different algorithms were compared by scaling tests performed in Matlab. Scaled images from Matlab's own implementations of the Nearest Neighbor, Bilinear, Bicubic and Lanczos2 algorithms were compared to images scaled by the Cisco reference scaler and my Matlab model of Winscale (Appendix B.3 - B.5). Due to limited amount of time, the edgeprocessor is not included in these comparisons.

One of the most common images used in evaluating image processing algorithms, is the image of Lenna ([Wik11], Figure 48(a)). The image of the swedish playboy model is a good example of a natural image containing both low- and high-frequency regions. Another useful image is the synthetically generated test pattern in Figure 48(b), called a Zone Plate. It looks like ripples in a pond, when a stone is tossed into it. The pattern can be described mathematically as a image containing frequency components from zero and upwards, directed from the center and towards

the edges. Ideal scaling algorithms would keep the shape of the pattern without adding artifacts.

The visual quality of the image results of the following tests are highly dependent on the type of print method used in printing this thesis. For this reason, have all the image results been attached this report as a .zip file. See Appendix A.

(a) Lenna.png (b) Rings.png

Figure 48: Test images

Upscaling

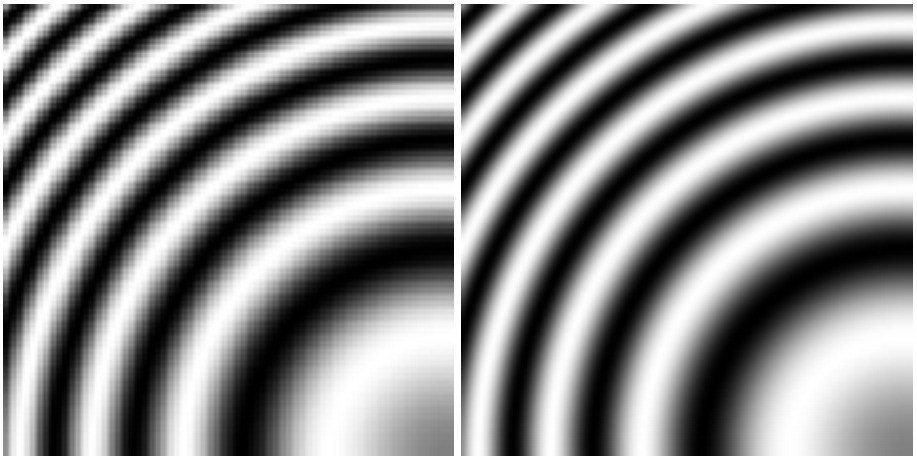
The first upscaling test was performed on a smaller region of the two source images, Lenna's eye (Figure 49(a)) and parts of the Zone Plate pattern (Figure 49(b)). These two regions were scaled

(a) Lenna.png Source (88x88) (b) Rings.png Source (100x100)

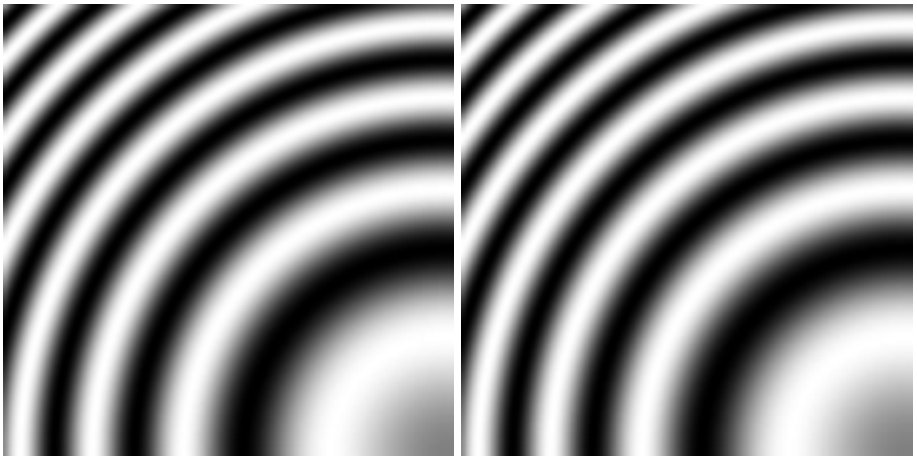
Figure 49: Source Regions Upscaling

up by the factors 1.681 and 2. The results are shown in Figure 50, 51,52 and 53. The reference scaler performs equally to Matlab's Lanczos2 and bicubic scaler with better edge response compared to bilinear. The nearest neighbor performs as expected by producing very aliased curves in both cases. The winscale algorithm (Figure 53(f), 52(f)) produces the same aliasing effects as the nearest neighbour at $SF = 2$. Various experiments confirms the statement from [CHK03], which states that the winscale algorithm is the same as nearest neighbor at integer scaling factors. The aliasing effects actually reduces when the scalingfactor is moved further away from an integer factor. This can be seen by observing less aliasing in Figure 51(f) than in Figure 53(f)

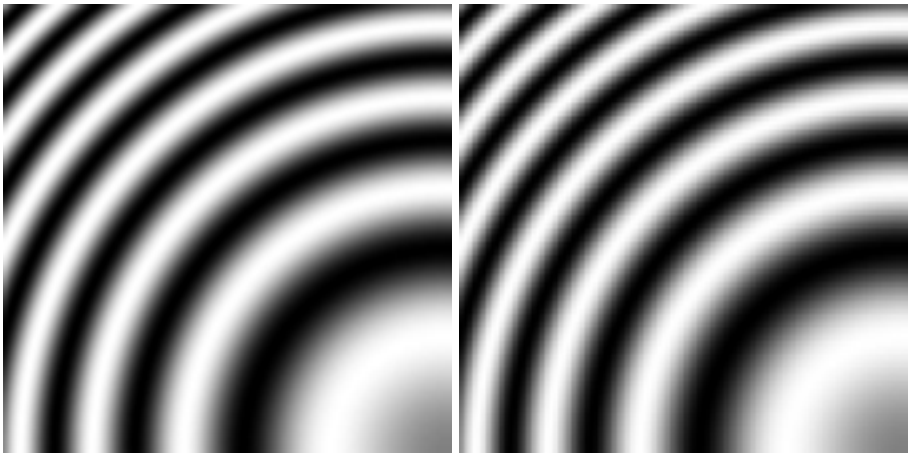
The aliasing at integer factors is an obvious disadvantage of the winscale algorithm. The ideal algorithm should perform equally for different factors.



(a) Nearest Neighbor (b) Bilinear

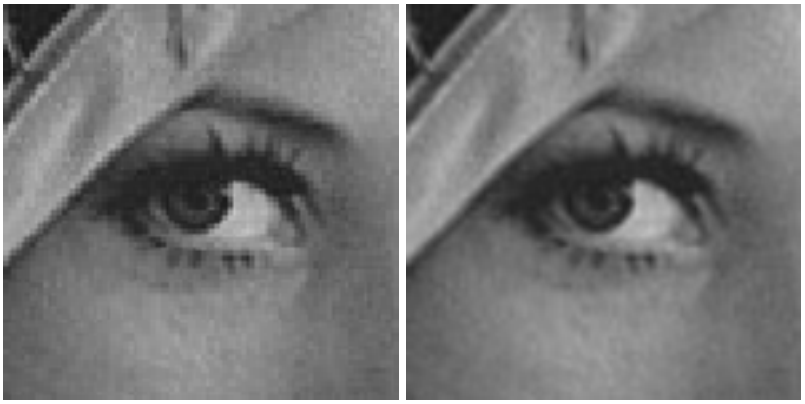


(c) Bicubic (d) Lanczos2



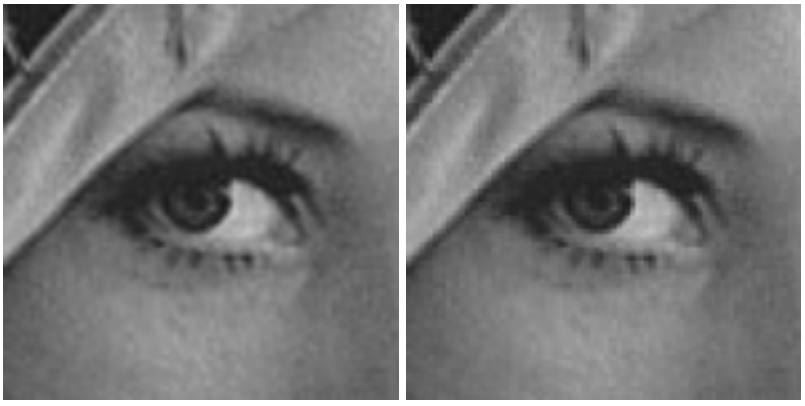
(e) Reference Algorithm (f) Winscale

Figure 50: Rings.png Upscaling factor = 1.681



(a) Nearest Neighbor (b) Bilinear

(c) Bicubic (d) Lanczos2



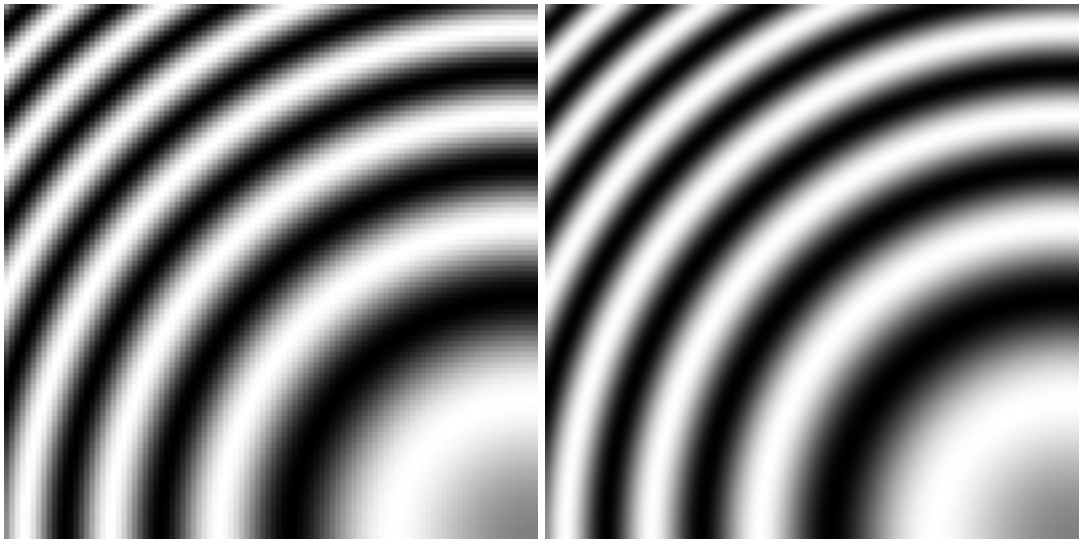
(e) Reference Algorithm (f) Winscale

Figure 51: Lenna.png Upscaling factor = 1.681

Downscaling

The Zone Plate pattern is especially important in downscaling test. The high frequency details will eventually reach pixel size when downscaling an image. At this point should the algorithm simulate our eye functionality by "blurring" the image, as our sight does with objects at long distances. As the winscale algorithm is modeled without a prescaler, visual comparisons may only be viewed with scalingfactors $\frac{1}{2} < SF \leq 1$. Figure 54, 55, 56 and 57 shows the resulting images of downscaling factors 0.5 and 0.78125. In evaluating of the Lena test (Figure 55 and

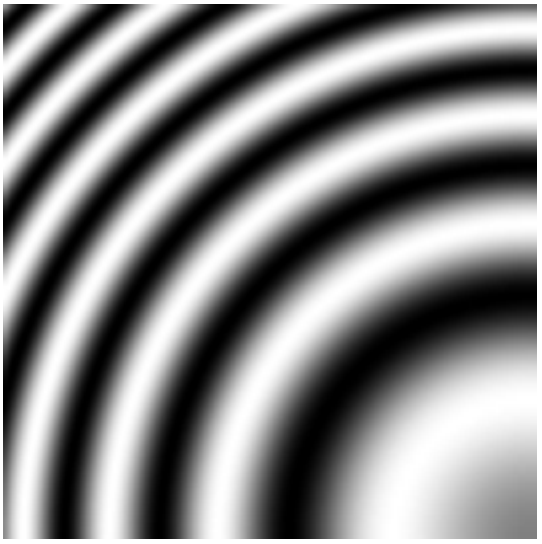
57) we clearly see that Nearest Neighbor and Bilinear underperforms on aliasing and blurring respectively. The reference model seemes to produce a slightly blurrier image at $SF = 0.78125$, compared to the Bicubic, Lanczos2 and the Winscale. At $SF = 0.05$ it's nearly impossible to separate bicubic, Cisco model, Lanczos2 and the Winscale. From these tests we can conclude that the winscale algorithm preforms very good at downscaling natural images in the range $\frac{1}{2} < SF \leq 1$. The question on how the winscale model performs under the 0.5 limit relies much on the chosen prescaler. This question remains unanswered, as potential prescalers not were investigated in this masters thesis. In evaluation of the Zone Plate test (Figure 54 and 56) we observe that the nearest neighbor, bilinear and winscale performs worse than bicubic, reference model and Lanczos2 in the high frequency regions. This proves that winscale is not the optimal algorithm for scaling synthetic images containing high frequency components, such as sharp and thin edges.



(a) Nearest Neighbor

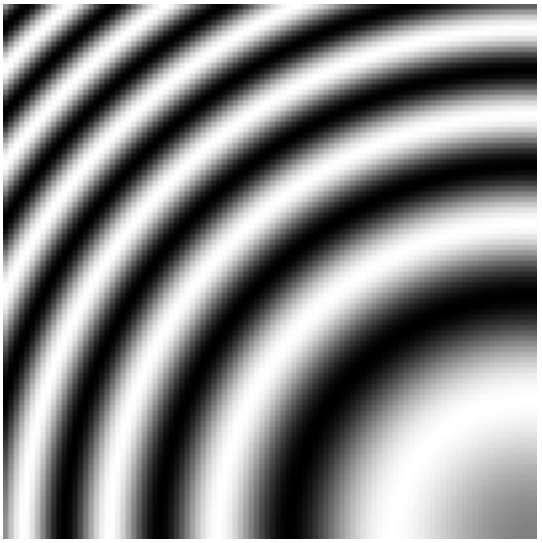
(b) Bilinear

(c) Bicubic



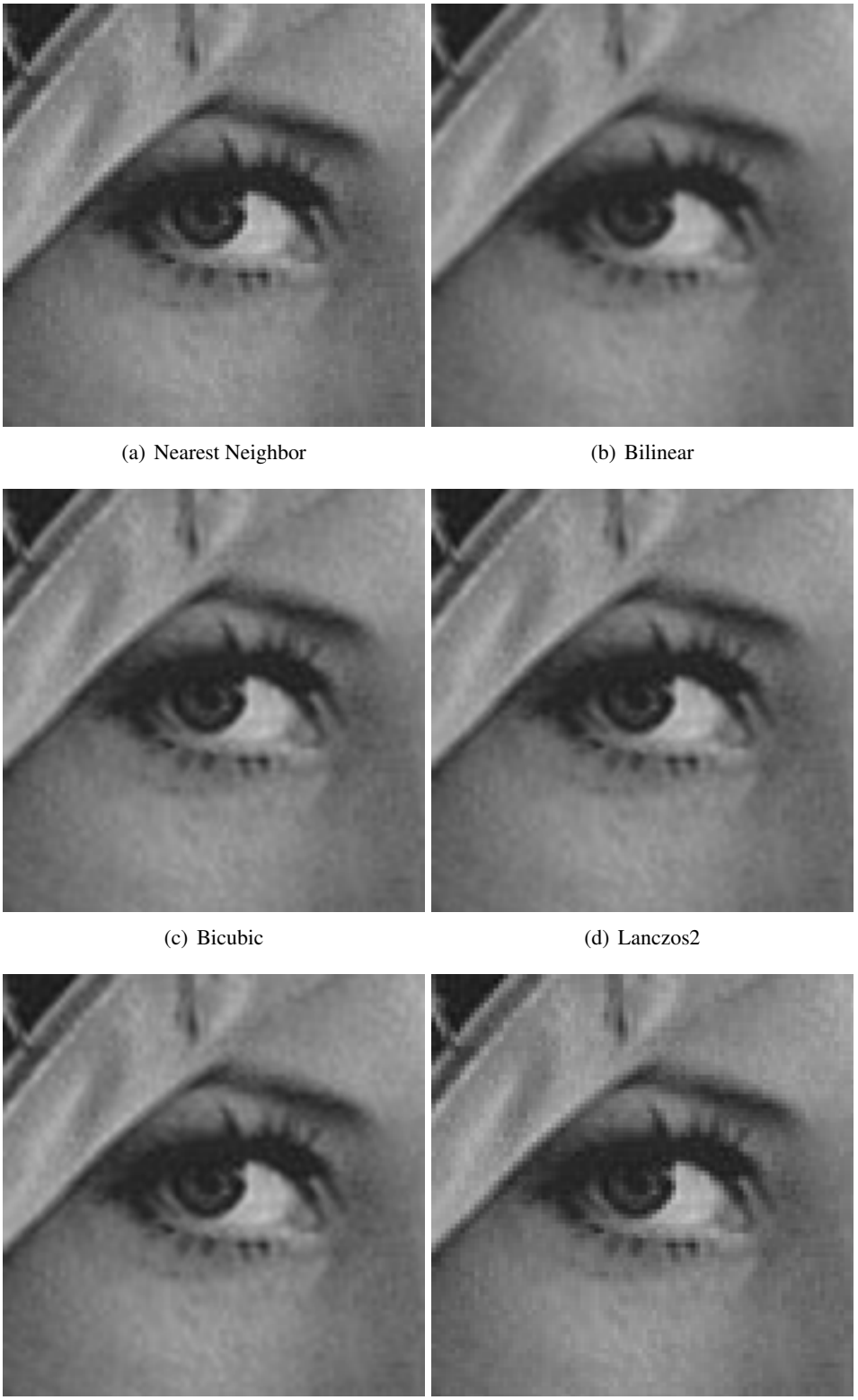
(d) Lanczos2

(e) Reference Algorithm



(f) Winscale

Figure 52: Rings.png Upscaling factor = 2.0

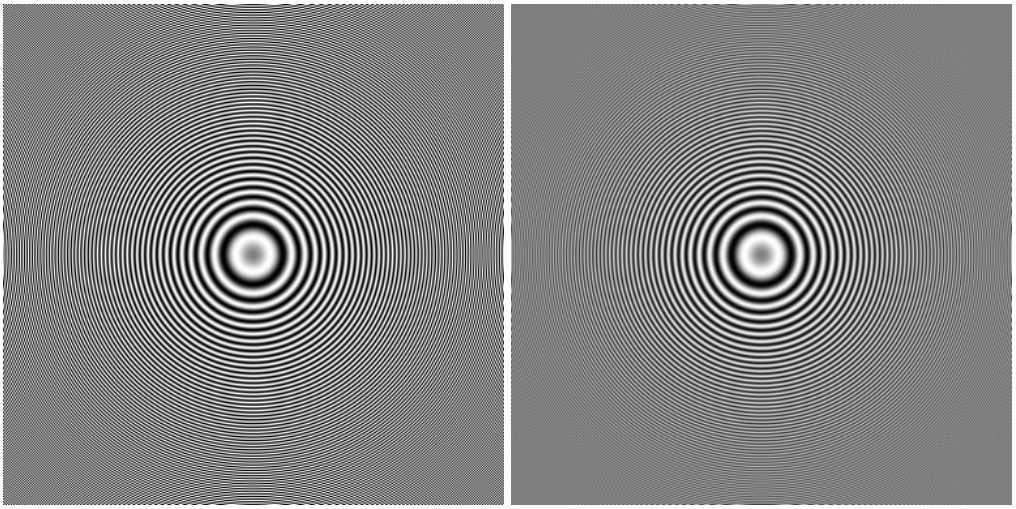


(a) Nearest Neighbor (b) Bilinear

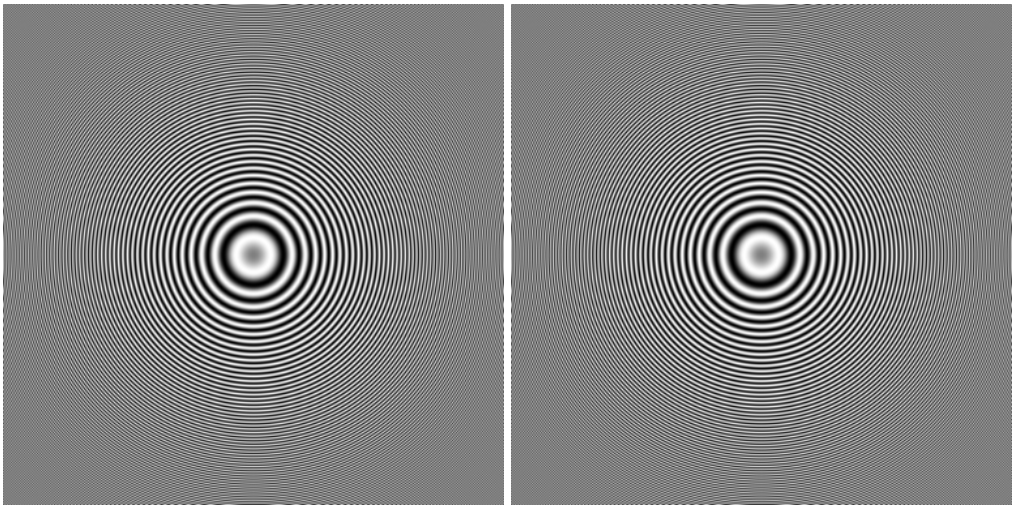
(c) Bicubic (d) Lanczos2

(e) Reference Algorithm (f) Winscale

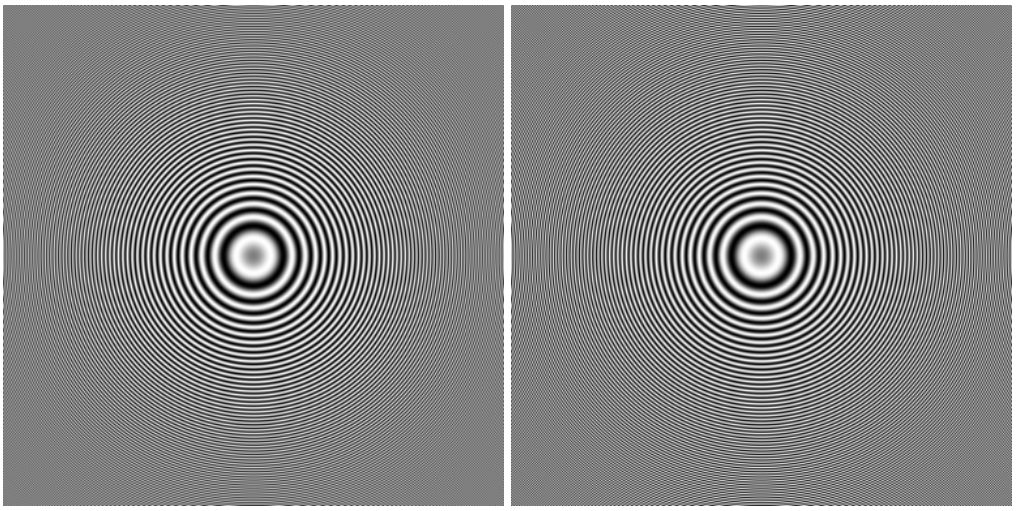
Figure 53: Lenna.png Upscaling factor = 2.0



(a) Nearest Neighbor (b) Bilinear



(c) Bicubic (d) Lanczos2



(e) Reference Algorithm (f) Winscale

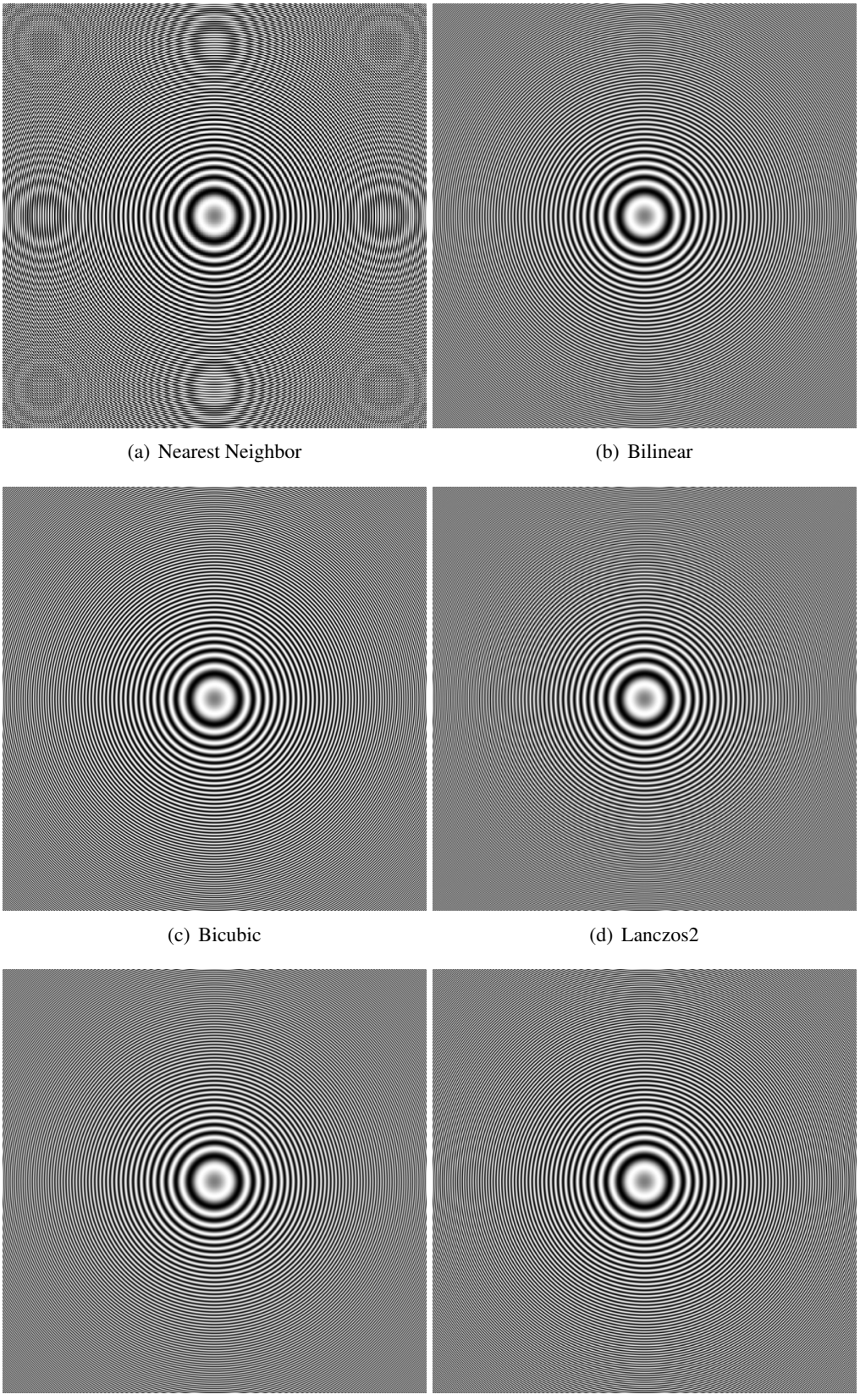
Figure 54: Rings.png Upscaling factor = $\frac{1}{2}$

(a) Nearest Neighbor (b) Bilinear

(c) Bicubic (d) Lanczos2

(e) Reference Algorithm (f) Winscale

Figure 55: Lenna.png Upscaling factor = $\frac{1}{2}$



(a) Nearest Neighbor (b) Bilinear

(c) Bicubic (d) Lanczos2

(e) Reference Algorithm (f) Winscale

Figure 56: Rings.png Upscaling factor = 0.78125

(a) Nearest Neighbor (b) Bilinear

(c) Bicubic (d) Lanczos2

(e) Reference Algorithm (f) Winscale

Figure 57: Lenna.png Upscaling factor = 0.78125

10. Video Scaling IP Cores

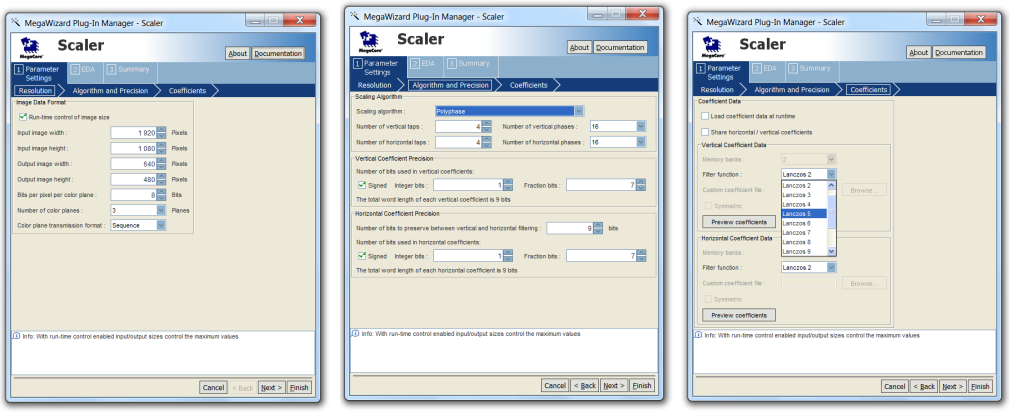
An *intellectual property core* (IP Core) is an of-the-shelf module, ready for ASIC or FPGA implementation. They are often optimized for a specific manufacturers device for maximum performance and low implementation cost. The cores are often configurable or parameterizable, so they can be adjusted for different applications. Models for functional simulation may be generated by the developer environment. The use of IP cores may reduce a product's time-to-market by avoiding designing standardized functions.

The disadvantage by using IP Cores is the lack of access to the IP Core source code. This makes more radical customization or debugging difficult. Although some IP providers provides the IP source code, the license often is very expensive. The designer gets bound to FPGA from a specific provider, if the source code is not available.

Altera provides the two video scaler IP cores *Scaler* and *Scaler II* in their *Viedeo and Image Processing Suite* ([alt]). Both of them are easily generated through Altera's *MegaWizard Plug-In Manager* (Figure 58,59).

Scaler

The Scaler MegaCore function resizes video streams. The Scaler supports nearest neighbor, bilinear, bicubic, and polyphase scaling algorithms. You can configure the Scaler to change resolutions or filter coefficients, or both, at run time using an AvalonMM slave interface.



(a) Resolution Settings (b) Algorithm Settings (c) Coefficient Settings

Figure 58: Mega Wizard Plug-In Manager - Scaler I

Scaler II

The Scaler II MegaCore function resizes video streams more efficiently than the Scaler. The Scaler II reduces the required resources with the support of 4:2:2 chroma data sampling rate. The Scaler II supports only bilinear and polyphase scaling algorithms.

IP-core Implementation

Table 10 presents some estimates to resource requirements for several specific scaling examples. This information is taken from Altera's own documentation ([Alt11],page 1-17:table 1-20 and 1-21) of the scalers. Statistics for the reference scaler is added for comparison purposes. These

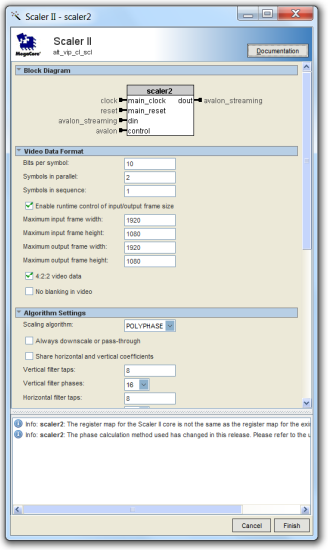


Figure 59: Mega Wizard Plug-In Manager - Scaler II

statistics are difficult to directly compare as it is not clear how flexible the example IP-core is at run-time according to non-fixed source and target resolutions. It would be very interesting to get detailed statistics on a more general purpose video scaler based on these two IP-cores. Such investigations have not been done due to time limitations.

Table 7: Scaler I and II statistics

IP core	**Device Family**	**Combinational LUTs/ALUTs**	**Logic Registers**	**Bits**	**M9K**	**(9x9)**	**(18x18)**	f_{MAX} **(MHz)**
Scaling up or down between NTSC standard definition and 1080 pixel high definition using 10 taps horizontally and 9 vertically. Resolution and coefficients are set by a run-time control interface								
Scaler I	Cyclone IV GX	4048	5243	417456	-	19	-	182.95
Scaler II	Cyclone IV GX	2839	4016	417936	76	29	-	156.37
Scaling NTSC standard definition (720x480) RGB to high definition 1080p using a bicubic algorithm								
Scaler I	Cyclone IV GX	1728	2078	69444	14	8	8	203.46
Scaler II	Cyclone IV GX	1397	1909	70512	13	12	-	167.34
Scaling up to, or down from maximum 1080p using a polyphase Lanczos2 algorithm using from 4 taps up to 16 taps. Resolution and coefficients are set by a run-time control interface.								
Reference Scaler	Cyclone III	1157	2019	81920	10	8	-	162.73

11. Dynamically Reconfigurable FPGA

As system designs grows larger and more complex, higher area requirements are set to the computing chip. FPGA has the advantage over ASIC, in that it may be configured at run-time. Such computing FPGAs are called run-time reconfigurable (RTR) FPGAs. This enables FPGAs to change functionality at runtime. A pipeline too large for the FPGA, could be implemented on a single FPGA by configuring different pipeline-stages at different times. Intermediate results have to be stored in registers while the FPGA is reconfigured. A time and resource overhead is connected to the use of RTR FPGAs. It is important that this overhead is negligible with respect to the time and resources used for computation. [RDH] shows an example of how a rather large video scaler may be implemented on a small Xilinx Reference Board. In this example, the reconfiguration overhead is less than one percent of the total computation time.

Another way of utilizing the RTR properties of the FPGA, is by making the actual filter structures in the different scalers more efficient. This could be done independently of the chosen scaling algorithm. [Bys08] describes how fixed constant coefficient FIR filter structures may be optimized by encoding coefficients with a CSD-encoding. Full multiplicatiors could thereby be replaced with lower complexity coefficient-specific multiplicators. It was proven to both reduce area consumption by up to 20% and increase maximum frequency up to 100% compared to the 2's complement representation. The coefficients are changed at every other interpolated pixel in the video scaling application. This would require reconfiguration of the scaler between every pixel. The reconfiguration total overhead would have to be significantly lower than the time in-operation, for this approach to be effective.

Yet another way of utilizing RTR properties would be by replacing the entire scaling structure. This could be useful if different algorithms were to be used for different scaling factors. Say that one Scaler A performs good upscaling quality at low areas, but requires large areas if used for downscaling. A second Scaler B, good at downscaling could replace Scaler A for downscaling factors. The total time and area required for this reconfigurable solution should not exceed an equal quality algorithm. It would be beneficial that Scaler A and Scaler B would be assembled by similar modules. This would reduce the required amount of reconfiguration, and reduce area and time overhead. Such potential architecture and algorithms have not been further researched or discussed beyond what have been presented in this section.

There is little established theory on video scalers based on dynamic reconfigurable FPGA architectures. This makes it difficult evaluate whether such solutions could be beneficial.

12. Conclusion and future work

It is clear that interpolation has a central role in designing a video or image scaler, although sinc-based interpolation algorithms suffer from some ringing-effects. The area-pixel model shows potential for low resource requirements and good quality in both upscaling and downscaling. The major drawback of the winscale algorithm is the fact that the visual quality varies, depending on whether the scale factor is near an integer. This makes it a not adequate replacement for such high complexity algorithms as Lanczos2 or the algorithm used in the reference scaler. The visual quality has to be consistent and somewhat predictable when changed. The visual quality of the edgeprocessor may be more persistent as a result of the added complexity of edge-detection and the compensation of the approximation, but further research, simulation and testing with this implementation is necessary.

The IP-core scalers examples could provide video scaling with faster operation, varying between 156 MHz and 203 MHz maximal frequency. IP-core could save both design cost and time-to-marked compared to own designs.

High complexity scalers based on the polyphasic FIR-filter structures and the Lanczos2 kernel will at this point in time be the best alternative to the reference scaler. The IP-core implementation of such scalers may provide better utilization of the FPGA resources.

Future Work

Future work might include:

- Matlab modeling and simulation of the edgeprocessor for visual quality evalutation.
- Implementation of a configured Scaler and Scaler II IP-core with the same configuration as the reference scaler.
- Complete implementation of the edgeprocessor on an FPGA in a videoconferencing system for in-field tests.
- Further research of effective frame prescalers, providing good visual quality together with area-pixel based algorithms, such as the edgeprocessor.
- Further research within the use of dynamical reconfigurable FPGA in video scaling applications. The prescalers in low complexity implementations may be implemented as a reconfigurable, and therefore optimized part of the scaler.

A. File attachments

A.1. List of file attachments

The following files where attached to this thesis as a .zip file:

Table 8: File Attachments

Path	Filename	Description
\	coefficients.m	MATLAB Lanczos2 Coefficient generator script
\	windowgeneration.m	MATLAB window function scipt
\	winscale_top.m	MATLAB MODEL: Winscale (1/3)
\	winscale_4pix.m	MATLAB MODEL: Winscale (2/3)
\	winscale_getDelta.m	MATLAB MODEL: Winscale (3/3)
\	lenna.png	Lenna Test Image
\	rings.png	Zone Plate Test Image
\lenna_2 \	[MULTIPLE .PNG]	Lenna image results factor = 2.0
\lenna_05 \	[MULTIPLE .PNG]	Lenna image results factor = 0.5
\lenna_168 \	[MULTIPLE .PNG]	Lenna image results factor = 1.681
\lenna_0781 \	[MULTIPLE .PNG]	Lenna image results factor = 0.78125
\rings_2 \	[MULTIPLE .PNG]	Rings image results factor = 2.0
\rings_05 \	[MULTIPLE .PNG]	Rings image results factor = 0.5
\rings_168 \	[MULTIPLE .PNG]	Rings image results factor = 1.618
\rings_0781 \	[MULTIPLE .PNG]	Rings image results factor = 0.78125

B. Matlab source code

B.1. windowgeneration.m

```
%SCRIPT FOR PLOTTING THE WINDOWING FUNCTIONS
%
% Author: ZIPcores.com (2009)
% "Estimating polyphase filter coefficients with a windowed function"
%
% 1. Lanczos2-windowed sinc
% 2. Blackman-windowed sinc
% 3. Hamming-windowed sinc
% 4. Kaizer-windowed sinc

% Lanczos2-windowed sinc over range -N/2,N/2
figure(1);
N = 5;
ks = -(N-1)/2:0.001:(N-1)/2;
kw = -(N-1)/2:0.001:(N-1)/2;

s = sinc(ks);
w = sinc(kw/2);

ws = s.*w;
%plot (ks,ws)
plot (ks,w)

% Blackman-windowed sinc over range -N/2,N/2
figure(2);
N = 5;
ks = -(N-1)/2:0.001:(N-1)/2;
kw = 0:0.001:(N-1);

s = sinc(ks);
w = 0.42 - 0.5*cos(2*pi*kw/(N-1)) + 0.08*cos(4*pi*kw/(N-1));

ws = s.*w;
%plot (ks,ws)
plot (ks,w)

% Hamming-windowed sinc over range -N/2,N/2
figure(3);
N = 5;
ks = -(N-1)/2:0.001:(N-1)/2;
kw = 0:0.001:(N-1);

s = sinc(ks);
w = 0.53836 - 0.46164*cos(2*pi*kw/(N-1));

ws = s.*w;
%plot (ks,ws)
plot (ks,w)

% Kaiser-windowed sinc over range -N/2,N/2
figure(4);
N = 5;
ks = -(N-1)/2:0.001:(N-1)/2;
kw = 0:0.001:(N-1);

s = sinc(ks);
```

```
alpha = 2*pi;
w = besseli(0,alpha*sqrt(1-(2*kw/(N-1) -1).^2))/besseli(0,alpha);

ws = s.*w;
%plot (ks,ws)
plot (ks,w)
```

B.2. coefficients.m

```
%SCRIPT FOR CALCULATING COEFFICIENTS FOR THE FOLLOWING FILTER
%
% Lanczos2-windowed sinc function
% 5 taps
% 16 phases per tap
%
% Author: ZIPcores.com (2009)
% "Estimating polyphase filter coefficients with a windowed function"
%

% Calculate coefficients for Phases 0 to 15,
% Taps 0,1,2,3,4
for p_index = 1:16
    for t_index = 1:5
        p = (p_index - 1)/16;
        t = (t_index - 1);
        x = t - 2 - p;
        coeff(p_index, t_index) = sinc(x) * sinc(x/2);
    end
end

% Quantize to 2.6-bit signed numbers
coeff_quant = round(coeff * 64);

% Check they sum to 1
sum_coeff = sum(coeff,2);
sum_coeff_quant = sum(coeff_quant,2);

% Write coefficients to a file I=Phase, J=Tap
fid = fopen('coeffs_5tap.txt', 'w');
fprintf(fid, ' TAP0 TAP1 TAP2 TAP3 TAP4 SUM\n\n');
for I = 1:16
    fprintf(fid, 'PHASE%2d : ',I-1 );
for J = 1:5
    fprintf(fid, '%5d ', coeff_quant(I,J));
end
    if sum_coeff_quant(I) == 64
        fprintf(fid, '%5d\n', sum_coeff_quant(I));
    else
      fprintf(fid, '%5d *\n', sum_coeff_quant(I));
    end
end

% The script stores the calculated coefficents to a text file.
% The values labelled with a asterisk (*) in the output text file must be
% adjusted in order to obtain sum to unity.
%
```

B.3. winscale_top.m

```
function target_image = winscale_top( source_image, newHeight, newWidth, type
    )
```

```
%WINSCALE_TOP
%    SOURCE_IMAGE     - original image ( oldHeight x oldWidth) [IMG]
%    NEWHEIGHT        - scaled height [PIX]
%    NEWWIDTH         - scaled width [PIX]
%    TARGET_IMAGE     - scaled image (newHeight x newWidth) [IMG]
%    TYPE             - for testing 'int' or 'float'

%-------------------------------------------------------------
% SOURCE IMAGE
%
%    C0   C2
%
%    C1   C3
%
%
%    C0-C3 are pixelvalues
%    oldWidth -
%    oldHeight -
%

oldWidth = size(source_image,2);
oldHeight = size(source_image,1);

%-------------------------------------------------------------
% TARGET IMAGE
%
%    A0   A2
%
%    A1   A3
%
%
%    A0 - A3 are size of the overlapping areas

%output_image = zeros(newHeight,newWidth);
output_image = zeros(newHeight,newWidth);


%-------------------------------------------------------------
% DISPLAY SECTION
disp('------------------------------');
disp('- NEW SCALING STARTED ')
disp(sprintf('-- oldWidth : %f',oldWidth));
disp(sprintf('-- oldHeight : %f',oldHeight));
disp('--')
disp(sprintf('-- newWidth : %f',newWidth));
disp(sprintf('-- newHeight : %f',newHeight));
disp('--')



if (oldWidth > newWidth) && (oldHeight > newHeight)
    %---------------------------------------------------------
    % DOWNSCALING
    disp('-------  DOWNSCALING -------');

    scaletype = 'downscaling';

    SF = 1.0;

    winW = 1.0;
```

```
    winH = 1.0;

    sourW = newWidth/oldWidth;
    sourH = newHeight/oldHeight;

%-------------------------------------------------------------
% Calculating coordinates for old pixels. (TARGET INDEXED)
% Same as "Coordinate Accumulator" in article
next_x = zeros(1,newWidth);
next_x(1) = 0.5 + sourW/2;
for i=1:(oldWidth-1)
    next_x(i+1) = next_x(i) + sourW;
end

next_y = zeros(1,newHeight);
next_y(1) = 0.5 + sourH/2;
for i=1:(oldHeight-1)
    next_y(i+1) = next_y(i) + sourH;
end

%next_x;
%next_y;

          %Initial value for source pixels are allways source(1,1) for
          %downscaling
           source_x= 1;
          %source_y = 1;

    for target_x=1:newWidth
        source_y = 1;
        for target_y=1:newHeight

        %upper_border = next_y(source_y) - sourH/2;
        lower_border = next_y(source_y)+ sourH/2;
        %left_border = next_x(source_x) - sourW/2;
        right_border = next_x(source_x) + sourW/2;


        upper_window_border = target_y - winH/2;
        left_window_border = target_x - winW/2;


        %Check if window border has moved so much that source pixels has
        %changed in y-direction
        while upper_window_border >= lower_border
            source_y = source_y + 1;
            lower_border = next_y(source_y)+ sourH/2;
        end

        %Check if window border has moved so much that source pixels has
        %changed in x-direction
        while left_window_border >= right_border
            source_x = source_x + 1;
            right_border = next_x(source_x) + sourW/2;
        end



        %source_y
        %source_x
```

```
        source_4pix = zeros(2,2);
        source_4pix(1,1) = source_image(source_y      , source_x);      %C0
        source_4pix(2,1) = source_image(source_y + 1  , source_x); %C1
        source_4pix(1,2) = source_image(source_y      , source_x +1 );      %C2
        source_4pix(2,2) = source_image(source_y + 1  , source_x + 1); %C3



        if strcmp('int',type)
              output_image(target_y,target_x) = floor(winscale_4pix(
                  source_4pix,next_x(source_x),next_y(source_y),winW, winH,
                  sourW,sourH,SF,target_x,target_y,scaletype));
           else
              if strcmp('float',type)
                 output_image(target_y,target_x) = winscale_4pix(source_4pix,
                     next_x(source_x),next_y(source_y),winW, winH,sourW,sourH,
                     SF,target_x,target_y,scaletype);
              else
                 disp('No TYPE (int or float) assigned');
              end

           end

        end
     end
```

```
else
     if (oldWidth < newWidth) && (oldHeight < newHeight)
     %-------------------------------------------------------------
     % UPSCALING
     disp('------  UPSCALING  ------');

     scaletype = 'upscaling';

     sourW = 1.0;
     sourH = 1.0;

%-------------------------------------------------------------
% FILTER WINDOW
%
%    winX,winY - center coordinates
%
%
%    winW = (1/HSR) - incremental horizontal ([W]idth) value
%    winH = (1/VSR) - incremental vertical ([H]eight) value
%
%    HSR = newWidth/oldWidth - [H]orizontal [S]caling [R]atio
%    VSR = newHeight/oldHeight - [V]ertical [S]caling [R]atio
%
```

```
%    SF = 1.0/(winW*winH)
%

HSR = newWidth/oldWidth;
VSR = newHeight/oldHeight;

winW = (1/HSR);
winH = (1/VSR);

SF = 1.0/(winW*winH);


%-------------------------------------------------------------
% DISPLAY SECTION
disp(sprintf('-- horizontal scaling factor HSR : %f',HSR));
disp(sprintf('-- vertical scaling factor VSR : %f',VSR));
disp('--')
disp(sprintf('-- SF : %f',SF));



%-------------------------------------------------------------
% Calculating coordinates for new pixels.
% Same as "Coordinate Accumulator" in article
next_x = zeros(1,newWidth);
next_x(1) = 0.5 + winW/2;
for i=1:(newWidth-1)
    next_x(i+1) = next_x(i) + winW;
end

next_y = zeros(1,newHeight);
next_y(1) = 0.5 + winH/2;
for i=1:(newHeight-1)
    next_y(i+1) = next_y(i) + winH;
end

%next_x
%next_y



%target_x = 1;
%target_y = 4;

%source_image

for target_x=1:newWidth
    for target_y=1:newHeight
        upper_border = next_y(target_y) - winH/2;
        lower_border = next_y(target_y)+ winH/2;
        left_border = next_x(target_x) - winW/2;
        right_border = next_x(target_x) + winW/2;

        upper_source_row = round(upper_border);


        left_source_row = round(left_border);
        %disp(sprintf('uppser_source_row = %f',upper_source_row));
        %disp(sprintf('left_source_row = %f',left_source_row));
        % Border conditions
```

```
%lower_border >= oldHeight + 0.5

%((oldHeight + 0.5) - lower_border)

%Check if current frame is a border frame on lower boundry
if (((oldHeight - 0.5) - upper_border)<(10^-10))
    upper_source_row = upper_source_row -1;
 %   disp(sprintf('Lower boundry reached. upper_source_row changed =
    %f',upper_source_row))
end

%Check if current target frame is a border frame right boundry
if (((oldWidth - 0.5) - left_border)<(10^-10))
    left_source_row = left_source_row - 1;
end


%disp(sprintf('left_source_row = %f',left_source_row));
%disp(sprintf('upper_source_row = %f',upper_source_row));


%Extracting the 4 pixels overlapped by the window
source_4pix = zeros(2,2);
source_4pix(1,1) = source_image(upper_source_row      ,
    left_source_row);     %C0
source_4pix(2,1) = source_image(upper_source_row + 1  ,
    left_source_row); %C1
source_4pix(1,2) = source_image(upper_source_row      ,
    left_source_row +1 );     %C2
source_4pix(2,2) = source_image(upper_source_row + 1  ,
    left_source_row + 1); %C3


%source_4pix;

%Calculate interpolated value based on:
% - 4 original pixels (source_4pix)
% - window dimmensions (SF,winW,winH,)
% - window position (left_source_row, upper_source_row, next_x(
    target_x), next_y(target_y))

if strcmp('int',type)

output_image(target_y,target_x) = floor(winscale_4pix(source_4pix,
    left_source_row,upper_source_row,winW, winH,sourW,sourH,SF,next_x
    (target_x),next_y(target_y),scaletype));
else
    if strcmp('float',type)
output_image(target_y,target_x) = winscale_4pix(source_4pix,
    left_source_row,upper_source_row,winW, winH,sourW,sourH,SF,next_x
    (target_x),next_y(target_y),scaletype);
    else
        disp('No TYPE (int or float) assigned');
   end

end
```

```
    end
    %

end

end
end

if strcmp('int',type)
    target_image = uint8(output_image);
else
    if strcmp('float',type)
        target_image = output_image;
    else
        disp('No TYPE (int or float) assigned');
    end

end




end
```

B.4. winscale_getDelta.m

```
function delta = winscale_getDelta( windowLenght,windowCoordinate,
    sourceCoordinate )
%WINSCALE_GETDELTA
%   windowLenght        - winW or winH
%   windowCoordinate    - winX or winY
%   sourceCoordinate    - cX or cY
%

if(((1-windowLenght)/2)<(windowCoordinate-sourceCoordinate) && (1 - (1-
    windowLenght)/2)>(windowCoordinate-sourceCoordinate) )
    %disp('DELVIS OVERLAPP');
    delta = (0.5 -(windowCoordinate-sourceCoordinate))+windowLenght/2;
else
    if((1 - (1-windowLenght)/2)<(windowCoordinate-sourceCoordinate))
        %disp('INGEN OVERLAPP');
        delta = 0;
    else
        %disp('FULL OVERLAP');
        delta = windowLenght;
    end
end


end
```

B.5. winscale_4pix.m

```
function pixel = winscale_4pix( pix, cX, cY, winW, winH,sourW,sourH, SF, winX
    , winY, scaletype)
%WINSCALE_4PIX
% 4PIX  -   a 2x2 pixel frame from original image
% WINW  - window incremental horizontal ([W]idth) value
```

```
% WINH  - window incremental vertical ([H]eight) value
% SOURW - source incremental horizontal ([W]idth) value
% SOURH - source incremental vertical ([H]eight) value
% SF    - scaling factor
% WINX  - filter center x-coordinate
% WINY  - filter center y-coordinate
% CX    - C0 x-coordinate
% CY    - C0 y-coordinate
% SCALETYPE - 'upscaling' or 'downscaling'

%-------------------------------------------------------------
% COEFFICIENT AND AREA CALCULATIONS
%
%   dL = winX - winLX
%   dR = 0.0
%   dT = winY - winTY
%   dB = 0.0
%
%   A0 = dL * dT
%   A1 = dL * (winH - dT)
%   A2 = dT * (winw - dL)
%   A3 = (winH - dT) * (winW - dL)
%
%   W0 = A0 * SF
%   W1 = A1 * SF
%   W2 = A2 * SF
%   W3 = 1.0 - W0 - W1 - W2

C0 = pix(1,1);
C1 = pix(2,1);
C2 = pix(1,2);
C3 = pix(2,2);


C0BX = cX - sourW/2;
C0BY = cY - sourH/2;
C3BX = C0BX + sourW;
C3BY = C0BY + sourH;
PBX = winX - winW/2;
PBY = winY - winH/2;

% Calculate delta overlapping distances

if strcmp('upscaling',scaletype)

    dL = winscale_getDelta(winW,winX,cX);
    dT = winscale_getDelta(winH,winY,cY);

else

    dL = (C0BX + sourW) - PBX;
    dT = (C0BY + sourH) - PBY;


end

    A0 = dL * dT;
    A1 = dL * (winH - dT);
    A2 = dT * (winW - dL);
    A3 = (winH - dT) * (winW - dL);
```

```
W0 = A0 * SF;
W1 = A1 * SF;
W2 = A2 * SF;
W3 = 1.0 - W0 - W1 - W2;


pixel = (W0 * C0) + (W1 * C1) + (W2 * C2) + (W3 * C3);

end
```

References

[alt] http://www.altera.com/products/ip/dsp/image_video_processing/m-alt-vipsuite.html.

[Alt11] Altera. http://www.altera.com/literature/ug/ug_vip.pdf, May 2011.

[Ben10] BenVista. http://www.benvista.com/photozoompro, 2010.

[Ber03] Gheroghe Berbecel. *Digital Image Display - Algorithms and Implementation*. Whiley, 2003.

[Bys08] Vebjørn Bystrøm. Low power/high performance dynamic reconfigurable filter-design. 2008.

[Cam10] Cambridge. http://www.cambridgeincolour.com/tutorials/image-interpolation.htm, 2010.

[CcL07] Wen-kai Tsai Ming-hwa Sheu Huann-keng Chiang Chung-chi Lin, Zeng-chuan Wu. *The VLSI Design of Winscale for Digital Image Scaling*. 2007.

[CHK03] Jin-Aeon Lee Lee-Sup Kim Chun-Ho Kim, Si-Mun Seong. *Winscale: An Image-Scaling Algorithm Using an Area Pixel Model*. 2003.

[Cis11] Cisco. http://www.cisco.no, 05 2011.

[MB10] Mark J. Burge Manson Burger. *Principles of Digital Image Processing: Core Algoritms*. Springer, 2010.

[oS10] onOne Software. http://www.ononesoftware.com/products/perfect-resize/, 2010.

[PYC09] Chi-Pin Lu Pei-Yin Chen, Chih-Yuan Lien. *VLSI Implementation of an Edge-Oriented Image Scaling Processor*. 2009.

[Qim10] Qimage. http://www.ddisoftware.com/qimage/, 2010.

[RDH] Peter M. Athanas Rhett D. Hudson, David I. Lehn. A run-time reconfigurable engine for image interpolation.

[Wik11] Wikipedia. http://en.wikipedia.org/wiki/lenna, May 2011.

[zip09] *Estimating polyphase filter coefficients with at windowed-sinc function*. ZIPcores, 2009.