# THE PULL-PUSH ALGORITHM REVISITED

## *Improvements, Computation of Point Densities, and GPU Implementation*

Martin Kraus

*Computer Graphics & Visualization Group, Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany*

*krausma@in.tum.de*

Keywords: pyramid algorithm, image processing, real-time rendering, scattered data, meshless data, interpolation, GPU

Abstract: The pull-push algorithm is a well-known and very efficient pyramid algorithm for the interpolation of scattered data with many applications in computer graphics. However, the original algorithm is not very well suited for an implementation on GPUs (graphics processing units). In this work, several improvements of the algorithm are presented to overcome this limitation, and important details of the algorithm are clarified, in particular the importance of the correct normalization of the employed filters. Moreover, we present an extension for a very efficient estimate of the local density of sample points.

## 1 INTRODUCTION

Interpolation of scattered data points is a classical problem in computer graphics with many applications, e.g., inpainting, reconstruction of smooth functions from meshless point data, image post-processing effects, etc. (Gortler et al., 1996; Drori et al., 2003). As recent GPU implementations of pyramid algorithms for scattered data interpolation achieve real-time performances, they are also suited for real-time rendering and real-time video processing (Lefebvre et al., 2005; Strengert et al., 2006; Guennebaud et al., 2007; Kraus and Strengert, 2007a).

However, there are particularly challenging rendering algorithms, which require the interpolation of scattered pixels for multiple images per frame. For example, the rendering of depth-of-field effects by Kraus and Strengert (Kraus and Strengert, 2007a) requires two full-screen interpolation passes for each "subimage" with 1 to 20 "subimages" per frame (depending on the strength of the effect). Thus, even a high-performance GPU implementation of scattered data interpolation can constitute a serious performance bottleneck for some rendering algorithms. Therefore, this work is concerned with improvements of GPU implementations of the very efficient pull-push algorithm by Gortler et al. (Gortler et al., 1996).

The next section briefly summarizes related work while Section 3 presents our algorithmic improvements of the pull-push algorithm and compares them to the original algorithm. Section 4 presents a variant of the algorithm to compute the local density of sample points, and in Section 5, the GPU implementations of the algorithms and their performance is discussed.

## 2 RELATED WORK

The original pull-push algorithm by Gortler et al. (Gortler et al., 1996) is based on work by Burt (Burt, 1981; Burt, 1988) and Mitchell (Mitchell, 1987) and was used to interpolate between many sampling points in a four-dimensional domain in the Lumigraph system.

A variant of the pull-push algorithm was employed, for example, by Drori et al. (Drori et al., 2003) for an initial reconstruction of missing parts of images. Lefebvre et al. (Lefebvre et al., 2005) used a GPU-based implementation for inpainting gaps in texture maps. Strengert et al. (Strengert et al., 2006) proposed a more efficient GPU implementation, which is of linear time complexity in the number of pixels. The algorithm by Strengert et al. was employed by Kraus and Strengert (Kraus and Strengert, 2007a) to disocclude pixels in their depth-of-field al-

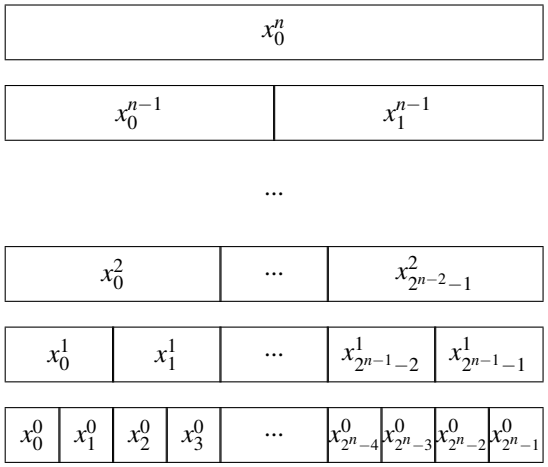



Figure 1: Illustration of the pixels $x_i^r$ of an image pyramid for a one-dimensional input image $x_i^0$ of size $2^n$.

gorithm. Guennebaud et al. (Guennebaud et al., 2007) employed a GPU implementation of the pull-push algorithm to reconstruct soft shadows from a reduced number of sample points; however, they did not describe any details of their implementation.

# 3 REVISION OF THE PULL-PUSH ALGORITHM

## 3.1 Pull

The original pull phase by Gortler et al. (Gortler et al., 1996) computes an image pyramid of data values $x$ and positive weights $w$ using a "pull filter" $\tilde{h}$. The input image determines the size of the finest pyramid level $r = 0$ while the size of pyramid level $r+1$ is half of the size of pyramid level $r$ in each dimension. For clarity we describe the algorithm for one-dimensional images and restrict their sizes to powers of two. Data values of pyramid level $r$ are denoted by $x_i^r$ and their weights by $w_i^r$ where $r$ equals 0 for the original (finest) resolution and pixels are indexed by $i$. This structure of a one-dimensional image pyramid is illustrated in Figure 1.

If the pull filter is specified by a discrete sequence $\tilde{h}$ of positive numbers, Gortler et al. suggest to compute the pixels of the next coarser image level by:

$$w_i^{r+1} = \sum_k \tilde{h}_{k-2i} \min\left(w_k^r, 1\right) \quad (1)$$

$$x_i^{r+1} = \frac{1}{w_i^{r+1}} \sum_k \tilde{h}_{k-2i} \min\left(w_k^r, 1\right) x_k^r \quad (2)$$

To reduce the number of clamp operations, i.e., min functions, we propose to introduce clamped weights $\hat{w}_i^r$ and weighted data values $\tilde{x}_i^r$, in particular for the input data on level $r = 0$:

$$\hat{w}_i^0 \stackrel{\text{def}}{=} \min\left(w_i^0, 1\right), \qquad \tilde{x}_i^0 \stackrel{\text{def}}{=} \hat{w}_i^0 x_i^0 \quad (3)$$

The computation of coarser image levels can now be performed with just one clamp operation and basic filter operations:

$$w_i^{r+1} = \sum_k \tilde{h}_{k-2i} \hat{w}_k^r \quad (4)$$

$$\hat{w}_i^{r+1} = \min\left(w_i^{r+1}, 1\right) \quad (5)$$

$$\tilde{x}_i^{r+1} = \frac{\hat{w}_i^{r+1}}{w_i^{r+1}} \sum_k \tilde{h}_{k-2i} \tilde{x}_k^r \quad (6)$$

This formulation is more suited for an efficient GPU implementation as discussed in Section 5.

It should be noted that the normalization of $\tilde{h}$ is crucial for the quality of the resulting interpolation. As shown in Section 3.3, a normalization of the maximum to 1 provides good results in most cases while a normalization of the integral of the filter $\tilde{h}$ to 1 leads to significantly worse results.

Our new formulation encapsulates the nonlinear part of the pull phase in a single clamp operation in Equation 5, which can be replaced easily by a continuous operation. For example, one could use a filter $\tilde{h}$ with a normalization such that its integral is 1 and achieve an effect similar to the nonlinear clamp operation by employing the following equation instead of Equation 5:

$$\hat{w}_i^{r+1} = 1 - \left(1 - w_i^{r+1}\right)^\gamma \quad (7)$$

Here, $\gamma$ is a parameter that controls the range of influence of data points: larger values of $\gamma$ tend to result in larger regions around isolated data points where the interpolated values are close to the values specified for the data points; examples are given in Section 3.3.

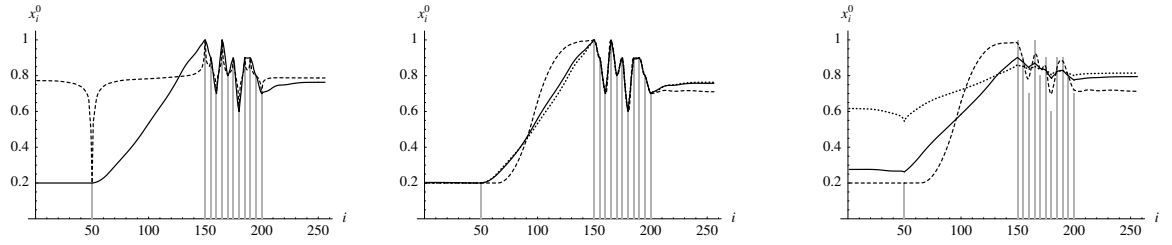
## 3.2 Push

After the image pyramid has been computed, a sequence of push operations recomputes all levels from coarse to fine with a "push filter" $h$. First, two temporary values are computed:

$$\omega_i^r = \sum_k h_{i-2k} \min\left(w_k^{r+1}, 1\right) \quad (8)$$

$$\chi_i^r = \frac{1}{\omega_i^r} \sum_k h_{i-2k} \min\left(w_k^{r+1}, 1\right) x_k^{r+1} \quad (9)$$

Note that Gortler et al. use the symbol $tw$ instead of $\omega$ and $tx$ instead of $\chi$.




(a) (b) (c)

Figure 2: (a) Interpolation of a set of scattered data *(gray bars)* by the "original" algorithm by Gortler et al. (Gortler et al., 1996) with $\tilde{h} = \left\{\frac{1}{3}, 1, 1, \frac{1}{3}\right\}$ *(solid line)* and $\tilde{h} = \left\{\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}\right\}$ *(dashed line)*. (b) Interpolation by the "w/o clamp" algorithm for $\gamma = 4$ *(solid line)* and $\gamma = 16$ *(dashed line)*. For comparison, the result of the original algorithm is also included *(dotted line)*. (c) Same as (b) for input data of weight $\frac{1}{32}$ instead of 1.

These temporary values are blended with the previously computed pyramid image. However, the equations provided by Gortler et al. lack clamping operations; the corrected equations are:

$$\begin{aligned} x_i^r &= \chi_i^r \left(1 - \min\left(w_k^r, 1\right)\right) + \min\left(w_k^r, 1\right) x_i^r \quad (10) \\ w_i^r &= \min\left(\omega_i^r, 1\right)\left(1 - \min\left(w_k^r, 1\right)\right) \\ &\quad + \min\left(w_k^r, 1\right) \quad (11) \end{aligned}$$

These equations can also be simplified by employing clamped weights $\hat{w}_i^r$ and weighted data values $\tilde{x}_i^r$ as introduced in the previous section. For the push phase we can additionally assume a normalization of the push filter $h$ such that no further clamp operations are necessary. Therefore, Equations 8–11 can be rewritten as:

$$\begin{aligned} \hat{\omega}_i^r &= \sum_k h_{i-2k} \hat{w}_k^{r+1} \quad (12) \\ \chi_i^r &= \frac{1}{\hat{\omega}_i^r} \sum_k h_{i-2k} \tilde{x}_k^{r+1} \quad (13) \\ x_i^r &= \chi_i^r \left(1 - \hat{w}_k^r\right) + \tilde{x}_i^r \quad (14) \\ \hat{w}_i^r &= \hat{\omega}_i^r \left(1 - \hat{w}_k^r\right) + \hat{w}_k^r \quad (15) \\ \tilde{x}_i^r &= \hat{w}_i^r x_i^r \quad (16) \end{aligned}$$

These filter operations can be implemented very efficiently (see Section 5); however, the linear interpolation with weights $\hat{w}_k^r$ and $1 - \hat{w}_k^r$ cannot be implemented by standard blending on GPUs due to the multiplication in Equation 16.

Fortunately, these equations can be further simplified by a new approximation, which works very well for most input data and filters $\tilde{h}$ and $h$. As noted above, the pull filter $\tilde{h}$ should be scaled such that the weights $\hat{w}_k^r$ quickly approach 1 in coarser pyramid levels. During the push phase these weights are further increased, thus, they are usually equal to 1 or close to 1 in all pyramid levels. This motivates a variation of the blending such that all new weights $\hat{w}_k^r$ are (implicitly) set to 1. Thus, Equations 12–16 are significantly simplified to only two equations:

$$\begin{aligned} \chi_i^r &= \sum_k h_{i-2k} x_k^{r+1} \quad (17) \\ x_i^r &= \chi_i^r \left(1 - \hat{w}_k^r\right) + \tilde{x}_i^r \quad (18) \end{aligned}$$

Note that we start with $x_k^{r+1}$ instead of $\tilde{x}_k^{r+1}$ on the coarsest level, which can be easily computed in the last step of the pull phase by avoiding the multiplication with $\hat{w}_i^{r+1}$ in Equation 6. For all but the first iteration, $x_k^{r+1}$ in Equation 17 refers to the pyramid level that was most recently computed according to Equation 18 in the push phase, while $\hat{w}_k^r$ and $\tilde{x}_i^r$ in Equation 18 refer to values computed in the pull phase.

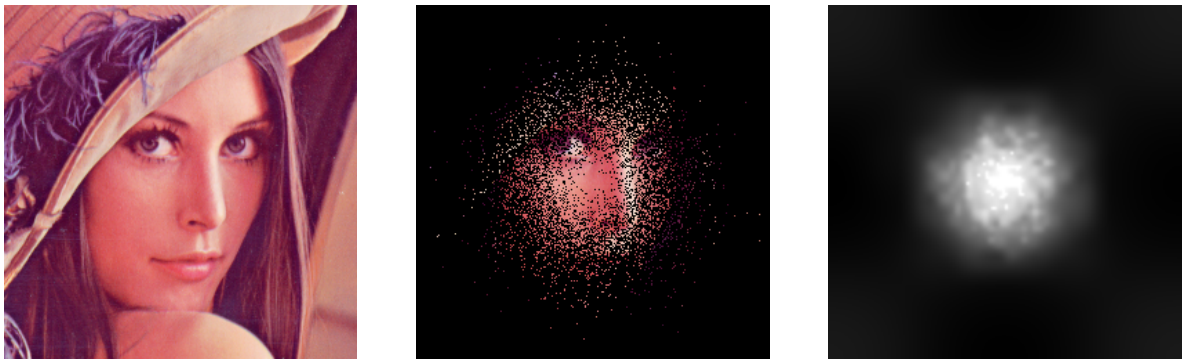
Our new formulation allows for both an efficient GPU implementation of the filtering operations by texture filtering as well as an efficient GPU implementation of the linear interpolation by frame buffer blending as discussed in Section 5.

## 3.3 Comparison of Variants

In this section, we compare the original algorithm by Gortler et al. (Gortler et al., 1996) with the two variants proposed in Sections 3.1 and 3.2.

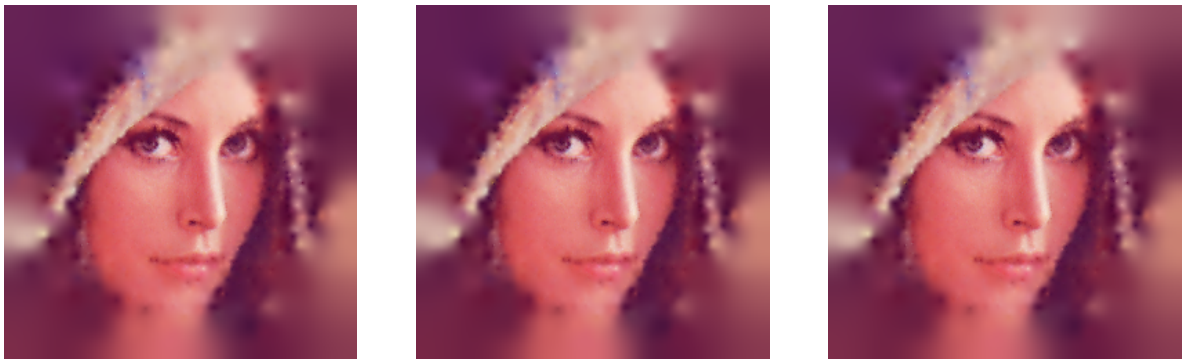
### 3.3.1 "Original" Algorithm

For the purpose of this comparison, the "original" algorithm is described by Equations 3–6 for the pull phase and Equations 8–11 for the push phase. A one-dimensional pull filter $\tilde{h}$ is used, which is defined by the sequence $\left\{\frac{1}{3}, 1, 1, \frac{1}{3}\right\}$, i.e., the maximum is normalized to 1. The push filter $h$ is defined by $\left\{\frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}\right\}$. These filters correspond to quadratic B-spline filters (Catmull and Clark, 1978) and are particularly well suited for GPU implementations (Kraus and Strengert, 2007b).



(a) (b) (c)

Figure 3: (a) A two-dimensional image of size $256 \times 256$. (b) Masked variant of the image in (a): the weight of black pixels is 0, the weight of all others is 1. (c) Density of the pixels with weight 1 computed by the algorithm presented in Section 4.



(a) (b) (c)

Figure 4: Images reconstructed from Figure 3b using (a) the "original" algorithm, (b) the "w/o clamp" variant, (c) the "w/o weight" variant.

Figure 2a depicts a set of scattered input pixels of weight 1 in a one-dimensional image of dimension 256 and the data interpolation computed with the original algorithm by Gortler et al. as described above. For comparison, the results with a differently normalized pull filter $\tilde{h}$ specified by $\left\{\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}\right\}$, is also included. With input weights less than or equal to 1, this normalization guarantees that all weights stay less than or are equal to 1 in the pull phase, therefore, the clamp operations are unnecessary.

However, the latter pull filter does not provide a smooth interpolation of the input points. In fact, it appears to be extremely difficult—if not impossible—to design appropriate pull filters for the original algorithm that do not require clamping operations but still result in smooth interpolations. Thus, in contrast to the discussion by Gortler et al., the clamping is not only necessary to limit the influence of clusters of many input points but it is also crucial for a smooth interpolation in general.

### 3.3.2 Variant "w/o clamp"

In order avoid the nondifferentiable clamping and to simplify the design of pull filters, we propose a first variant of Gortler et al.'s algorithm, named "w/o clamp," which replaces the clamping specified in Equation 5 by the differentiable Equation 7. For this variant of the algorithm, the integral of the pull filter $\tilde{h}$ has to be normalized to 1 in order to guarantee that weights do not exceed 1. We choose $\left\{\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}\right\}$ for the examples depicted in Figure 2b and 2c. The parameter $\gamma$ in Equation 7 was set to 4 and 16 in order to show its influence.

While the input pixels in Figure 2b are the same as in Figure 2a, the input weights in Figure 2c are only $\frac{1}{32}$ instead of 1 in Figure 2b. The dotted curve depicted in Figure 2c suggests that it is difficult to obtain a smooth curve approximating sample points with weights less than 1 using the original algorithm. The situation is quite similar to an unsuitable normaliza-

tion of $\tilde{h}$ as the clamping operation is ineffective for several levels in both cases. On the other hand, alternative nonlinear filters such as implemented in our variant "w/o clamp," can provide smoother curves for input points with weights less than 1 as illustrated by the dashed curve in Figure 2c.

#### 3.3.3 Variant "w/o weight"

The second variant is named "w/o weight" and implements Equations 3–6 for the pull phase; however, the multiplication with $\hat{w}_i^{r+1}$ in Equation 6 is skipped for the coarsest level in order to compute $x_i^{r+1}$ instead of $\tilde{x}_i^{r+1}$. For the push phase, Equations 17 and 18 are employed. In our experiments, this variant and the original algorithm computed very similar results although the "w/o weight" variant employs a considerably simplified push phase, which is better suited for an efficient implementation on GPUs.

#### 3.3.4 Two-Dimensional Examples

Figure 4 presents the two-dimensional results for the input depicted in Figure 3b. Visually there are no differences between the result of the original algorithm employed for Figure 4a and the results for the two new variants of the algorithm shown in Figure 4b and Figure 4c.

## 4 COMPUTATION OF POINT DENSITIES

It is often useful to compute a local density of scattered points, for example in order to decide where to place additional sampling points. Here we present a computation of the local density of input points by a variant of the presented pull-push algorithm.

We assume that all weights of input pixels $w_i^0$ are equal to 1 and all weights of unspecified pixels are 0. In this case, the pyramid images $w_i^r$ computed in the pull phase are in fact approximations of the density of input points if the integral of the pull filter $\tilde{h}$ is normalized to 1. However, it is crucial to choose an appropriate pyramid level $r$. In particular, $r$ should be chosen locally. To this end, we base the computation of the point density on those values $w_i^r$ where $r$ is appropriate for the specific position $i$. This can be achieved by introducing new weights $p_i^r$ for the interpolation of densities $w_i^r$: the weight $p_i^r$ determines the influence of $w_i^r$ analogously to the role $w_i^r$ played for $x_i^r$ in the original pull-push algorithm.

We determine actual values for $p_i^r$ in dependency of the average number $\bar{n}_i^r$ of input points at pixel $i$ in the pyramid image at level $r$:

$$\bar{n}_i^r \stackrel{\text{def}}{=} 2^r w_i^r \tag{19}$$

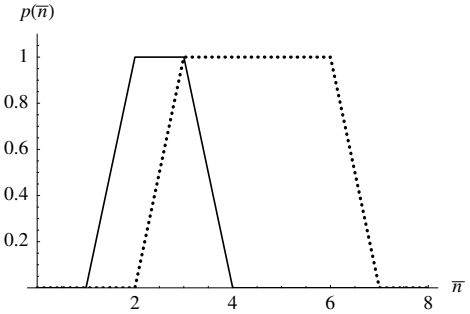



Figure 5: Two examples for the function $p(\bar{n})$ to determine weights depending on the average number $\bar{n}$ of input points per pixel. The solid curve is suitable for one-dimensional images while the dotted curve is suitable for two-dimensional images.

If the average number of input points per pixel $\bar{n}_i^r$ is less than 1, $p_i^r$ should be 0 because the corresponding $w_i^r$ is very likely to overestimate the local density in the neighborhood of a single, isolated input point. On the other hand, if $\bar{n}_i^r$ is very large, $p_i^r$ should also be 0 as local variations of the density of input points are not reflected by the corresponding $w_i^r$ if too many input points are included in the average. In between these extremes, $p_i^r$ is set according to a function $p(\bar{n})$, which is application-specific as even very fine local variations are of interest in some cases—for example when estimating the distance to the nearest input point—while other applications—in particular for high-dimensional images—require larger values of $\bar{n}_i^r$ to reliably estimate the density of input points. Figure 5 illustrates two examples for the function $p(\bar{n})$.

Having computed densities $w_i^r$ and their weights $p_i^r$ in the pull phase, we can then apply a push phase, which reconstructs a smooth function for $w_i^0$ on the finest pyramid level analogously to reconstructing a smooth function $x_i^0$ from data values $x_i^r$ and weights $w_i^r$. Next we describe this algorithm for the GPU-friendly variant of the pull-push algorithm presented in Section 3.2 (called "w/o weight" in Section 3.3).

The algorithm starts with an input image $w_i^0$, where the $w_i^0$'s are either 0 or 1. The pull phase implements the following equations:

$$w_i^{r+1} = \sum_k \tilde{h}_{k-2i} w_k^r \tag{20}$$

$$\bar{n}_i^{r+1} = 2^{r+1} w_i^{r+1} \tag{21}$$

$$p_i^{r+1} = p\left(\bar{n}_i^{r+1}\right) \tag{22}$$

As mentioned above, the integral of the pull filter $\tilde{h}$ has to be normalized to 1. The push filter $h$ has to be

normalized analogously and is used in the push phase, which recomputes the image pyramid from coarse to fine levels using these equations:

$$\chi_i^r = \sum_k h_{i-2k} w_k^{r+1} \quad (23)$$

$$w_i^r = \chi_i^r (1 - p_k^r) + p_k^r w_i^r \quad (24)$$

The finest pyramid image $w_i^r$ is the resulting estimate of the local density of points.

Figure 3c depicts the resulting density for the input shown in Figure 3b. The two-dimensional generalizations of the same pull and push filters were employed and the dotted curve in Figure 5 was used for $p(\bar{n})$.

# 5 GPU IMPLEMENTATION

While there are very efficient GPU implementations of linear filtering (Kraus and Strengert, 2007b), any additional functions in the summation (as in Equation 1) or additional nonconstant factors (as in Equation 2) defeat an efficient GPU implementation. This problem is solved by our improvements presented in Section 3, in particular Equations 4, 6, and 17.

For multidimensional data (e.g., RGB images) the filtering operations in Equations 6 and 17 can be performed in parallel for up to 3 components on GPUs. Moreover, the filtering of weights in Equation 4 can be performed in parallel in the A component of an RGBA texture image in order to implement Equation 18 by alpha-blending. Note that only $\hat{w}_i^r$ and $\tilde{x}_i^r$ have to be stored in the pull phase, while the push phase only stores new values for $x_i^r$ in place of the values $\tilde{x}_i^r$, which were computed in the pull phase.

While Gortler et al. (Gortler et al., 1996) claim that Equations 8–11 can be implemented by standard blending, this is unfortunately no longer true if the computation is based on weighted variables $\tilde{x}_i^r$ for more efficient filtering. This problem is solved by the approximation presented in Section 3.2, in particular Equations 17 and 18.

The two-dimensional filters employed in Section 3.3 require only four texture lookups per pixel of the coarser level in the pull phase and one texture lookup per pixel of the finer level in the push phase if the filtering is implemented efficiently (Kraus and Strengert, 2007b). As GPU implementations of the proposed algorithms are bandwidth-limited, the arithmetic operations do not significantly affect the performance. Our implementation requires 0.60 ms for a $1024 \times 1024$ 16-bit floating-point RGBA image on a NVIDIA Quadro FX 5800 graphics board.

# 6 CONCLUSIONS

As shown in this work, the proposed algorithms for the interpolation of scattered data points and the computation of local densities of scattered points are well suited for very efficient GPU implementations, which could be used for real-time rendering or real-time image and video processing.

The quality of the results computed by the proposed algorithms depends on the choice of the pull and push filters, the nonlinear function in Equation 7, and the function $p(\bar{n})$ in Equation 22. Determining suitable filters and functions for particular applications is part of future work.

# REFERENCES


Burt, P. J. (1981). Fast Filter Transforms for Image Processing. *Computer Graphics and Image Processing*, 16:20–51.

Burt, P. J. (1988). Moment images, polynomial fit filters, and the problem of surface interpolation. In *Proceedings of Computer Vision and Pattern Recognition*, pages 144–152.

Catmull, E. and Clark, J. (1978). Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer Aided Design*, 10(6):350–355.

Drori, I., Cohen-Or, D., and Yeshurun, H. (2003). Fragment-Based Image Completion. *ACM Transactions on Graphics*, 22(3):303–312.

Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F. (1996). The Lumigraph. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 43–54.

Guennebaud, G., Barthe, L., and Paulin, M. (2007). High-Quality Adaptive Soft Shadow Mapping. *Computer Graphics forum (Proceedings Eurographics 2007)*, 26(3):525–533.

Kraus, M. and Strengert, M. (2007a). Depth-of-Field Rendering by Pyramidal Image Processing. *Computer Graphics Forum (Conference Issue)*, 26(3):645–654.

Kraus, M. and Strengert, M. (2007b). Pyramid Filters Based on Bilinear Interpolation. In *Proceedings GRAPP 2007 (Volume GM/R)*, pages 21–28.

Lefebvre, S., Hornus, S., and Neyret, F. (2005). Octree Textures on the GPU. In Pharr, M., editor, *GPU Gems 2*, pages 595–613.

Mitchell, D. P. (1987). Generating Antialiased Images at Low Sampling Densities. In *Proceedings of SIGGRAPH '87*, pages 65–72.

Strengert, M., Kraus, M., and Ertl, T. (2006). Pyramid Methods in GPU-Based Image Processing. In *Proceedings Vision, Modeling, and Visualization 2006*, pages 169–176.