Pyramid Methods in GPU-Based Image Processing

Magnus Strengert, Martin Kraus, and Thomas Ertl

Institut für Visualisierung und Interaktive Systeme (VIS)
Universitätsstraße 38, 70569 Stuttgart, Germany
Email: {magnus.strengert,martin.kraus,thomas.ertl}@vis.uni-stuttgart.de

Abstract

There are numerous applications and variants of pyramid methods in digital image processing. Many of them feature a linear time complexity in the number of pixels; thus, they are particularly well suited for real-time image processing. In this work, we show that modern GPUs allow us to implement pyramid methods based on bilinear texture interpolation for high-performance image processing and present three examples: zooming with biquadratic B-spline filtering, efficient image blurring of arbitrary blur width, and smooth interpolation of scattered pixel data. In comparison with published techniques for GPU-based image processing, we achieve considerable performance improvements compared to published filtering techniques and improvements of image quality compared to bilinear interpolation.

1 Introduction

The pyramid algorithm [1, 9] has found many applications in image processing; in particular because the worst-case time complexity of many pyramid methods is linear in the number of pixels and does not depend on the potential range of influence of each pixel. Thus, for some applications pyramid methods are even more efficient than filtering techniques based on fast Fourier transforms [1]. Since the pyramid algorithm is related to discrete wavelet transforms and multiscale analysis, pyramid methods are applied to similar problems in image processing and vision; e.g., compression, enhancement, reconstruction, segmentation, feature measurement, etc.

The basic pyramid algorithm iteratively computes a pyramid of low-resolution approximations to an original image. The computation of each pixel of a coarser level depends only on the data of a small constant number of pixels of the previous, finer level. The computation of this image pyramid is often called analysis while the inverse process, i.e., the reconstruction of an image of the original size from the pyramid data, is called synthesis.

The basic problem of any implementation of pyramid methods on graphics hardware is the dependency of pixel computations on results of the previously computed level. If the computation of one level is implemented by rasterizing one rectangle with an appropriate fragment program, the data of the previous level has to be accessed by texture lookups. Thus, the pixels of each level have to be copied to a texture image before the next level can be rasterized. In the fixed-function OpenGL pipeline this would require to read back the rasterized pixels and to copy them to a texture image; thus, the data of each level is sent back and forth between graphics memory and main memory. Since this data transfer is a serious bottleneck, the performance of implementations of the pyramid algorithm on non-programmable graphics hardware was rather limited and, therefore, these implementations never achieved the popularity of CPU-based implementations of pyramid methods.

Modern GPUs offer the possibility to render directly to texture images; thus, the resulting pixel data may be used for the texturing of subsequent primitives without the performance costs of transferring image data to main memory and back to graphics memory. This feature has removed the most important bottleneck of GPU-based implementations of the pyramid algorithm. In fact, the main contribution of this work is to show with the help of three examples that pyramid methods can be implemented on modern GPUs without the performance limitations of previous implementations. Some specific publications about these implementations and about the pyramid algorithm in general are summarized in Section 2.

Since we focus on high-performance implementations, we do not consider advanced designs of analysis and synthesis filters for the presented pyramid methods. On the contrary, we employ very basic weighting functions that may be implemented by a single bilinear texture lookup as discussed in Section 3. These filters suffice to provide a satisfactory visual quality in several common applications of pyramid methods while resulting in particularly efficient implementations. More specifically, the presented examples are a pixel zoom with biquadratic B-spline filtering described in Section 4, image blurring with a worst-case time complexity that is independent of the blur width as discussed in Section 5, and the interpolation of scattered data including the filling of missing pixels presented in Section 6.

More potential applications for GPU-based pyramid methods are mentioned in Section 7, which concludes this paper.

2 Related Work

Burt did not only propose the pyramid algorithm in 1981 [1] but also described the Gaussian pyramid consisting of coarser approximations of an image and the Laplacian pyramid consisting of the detail information required for the reconstruction of each level of the Gaussian pyramid from the next coarser level. Specific pyramid methods are discussed by Ogden et al. [9] and Burt [2].

Catmull and Clark [3] described a subdivision scheme for cubic and quadratic B-splines. The scheme for biquadratic B-spline patches is wellkown as the Doo-Sabin subdivision scheme for regular quadrilaterals. It is of particular importance for our work because it can be implemented by a single bilinear texture lookup as described in Section 3.

The most important examples of Gaussian pyramids in real-time computer graphics are mipmap textures. Usually the generation of mipmaps on non-programmable graphics hardware requires the transfer of image data from main memory to graphics memory, which limits the performance of dynamic updates of mipmap textures. However, recent GPUs offer very efficient mipmap generation from texture images without this data transfer as discussed in Section 3.1.

Apart from mipmapping there are several publications about implementations of pyramid methods in graphics hardware; in particular, implementations of the discrete wavelet transform (DWT). Hopf and Ertl [6] implemented a DWT with the help of the OpenGL imaging pipeline. Considerably higher rendering performances were achieved by Wang et al. [13] and Tenllado et al. [12] who avoided data transfer between graphics memory and main memory by employing programmable graphics hardware. The pyramid methods we present in this paper differ from DWTs in at least two aspects: we focus on filters that are implementable with a single bilinear texture lookup and we do not require any data of the Laplacian pyramid, which corresponds to the detail coefficients of a DWT. Efficient GPU-based implementations of DWTs are most useful for image compression and non-interactive signal and image processing while there are very few applications in real-time computer graphics [5].

GPU-implementations of pyramid methods became also popular in the context of general GPU-based computations, especially for the efficient computation of the mean or the sum of pixel values of a set of pixels by a sequence of pixel gather operations, e.g., for linear algebra computations on GPUs as discussed by Krüger and Westermann [7]. A more recent example is the GPU-based computation of one-dimensional prefix-sums presented by Sengupta et al. [10].

An example of a GPU-based implementation of an algorithm for (non-interactive) image processing was published by Goodnight et al. [4]. While the authors compute the average luminance of an image with the help of a pyramid method, they employ a set of filtered images for the computation of "adaptation zones" to perform adaptive tone mapping. This "pyramid" is different from the Burt pyramid as each of its levels is of the dimensions of the original image. Nonetheless, the authors call it a "Gaussian pyramid" since each level represents the convolution of the original image with a Gaussian filter. Thus, the total number of pixels is considerably larger than in the Gaussian pyramid suggested by Burt. Moreover, the Gaussian filters employed by Goodnight et al. require access to many more pixels than the filters of constant width in the pyramid algorithm. Thus, the computation of adaptation zones presented by Goodnight et al. is not a pyramid method in the sense of the Burt pyramid.

Strictly speaking, the GPU implementation of the push-pull interpolation of scattered pixel data by

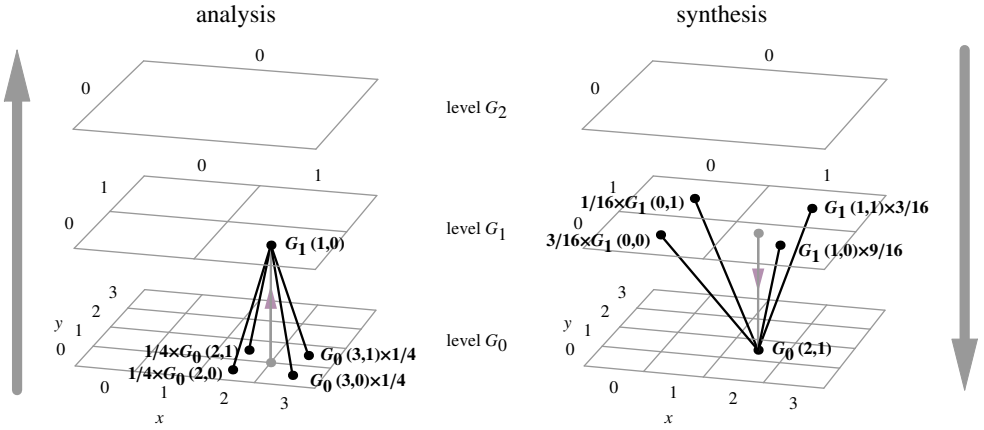


Figure 1: Basic structure of many pyramid methods: a bottom-up analysis *(left)* followed by a top-down synthesis *(right)* of image data. The analysis operation for $G_1(1,0)$ and the synthesis of $G_0(2,1)$ illustrate the specifc analysis and synthesis filters employed in this work. The gray dots represent the coordinates of the corresponding single bilinear texture lookups.

Lefebvre et al. [8] is also not a pyramid method since it accesses all levels of a mipmap texture for each pixel. This results in additional texture lookups of their implementation and therefore in a worse time complexity (by a logarithmic factor) and a worse performance compared to the pyramid method presented in Section 6.

An efficient GPU-based filtering of images with cubic B-splines for texture magnification was presented by Sigg and Hadwiger [11]. The implementation exploits bilinear texture interpolations to reduce the number of required texture lookups. While this technique allows for random access to a texture image, it requires considerably more texture lookups per pixel of a zoomed image than the pyramid method presented in Section 4.

Green [5] presented GPU-based implementations of infinite impulse response filters (IIR filters) for image processing. Although this technique is not related to pyramid methods, it features the same linear time complexity even for filters of infinite length. The performance of this approach is limited by the number of required passes (forward and backward for both dimensions) and the number of primitives (and render buffer switches) for each pass. The latter is linear in the dimensions of the image while it is logarithmic in these dimensions for the pyramid method presented in Section 5.

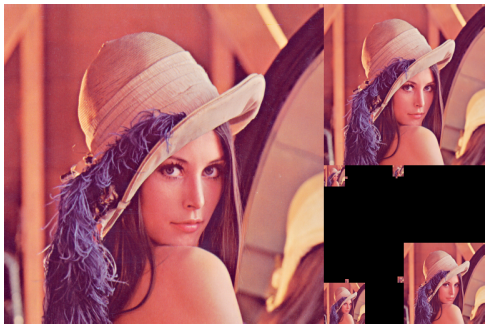
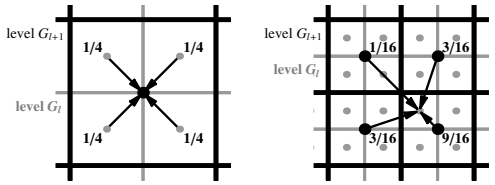


Figure 2: Layout of all 10 levels of the Gaussian pyramid in one 770×512 image buffer for the 512×512 Lena image.

3 Pyramid Schemes Based on Bilinear Texture Interpolation

This section introduces the basic concepts and operations employed in the more specific pyramid methods discussed in Sections 4, 5, and 6. We also discuss the implementation of these operations with the help of hardware-accelerated bilinear texture interpolation since this possibility motivated our choice for the particular analysis and synthesis operations.



(a) (b)

Figure 3: (a) Analysis operation for averaging. (b) Synthesis operation for biquadratic B-spline filtering.

3.1 Analysis for Mean Pyramids

The common structure of the pyramid methods presented in this work is illustrated in Figure 1. The original image is called G_0 since it is considered the 0th level of a Gaussian pyramid. Starting from G_0 the analysis computes a pyramid of coarser levels G_1, G_2, etc. The width and height of each level is reduced by a factor of two in comparison to the previous, finer level.

We store all even levels of the pyramid in one renderbuffer image and all odd levels in a second renderbuffer image for a ping-pong rendering between the two renderbuffers. Figure 2 depicts the layout of 10 levels of the Gaussian pyramid for an original image of size 512×512 pixels in our implementation with all levels in one image for the purpose of illustration only. Since new pixel data is computed from pixels of the previous level according to an analysis operation, the renderbuffer image of an odd level is used as a texture image to compute the pixels of an even level and vice versa. The computation of the pixels of one level is performed by rasterizing a rectangle covering these pixels. Texture coordinates of the vertices of this rectangle are set appropriately to access the pixel data of the previous level.

The particular analysis operation employed in the presented pyramid methods is illustrated in Figures 1 and 3a. The components of a new pixel are determined by the average components of four pixels of a finer level; thus, the resulting pyramid is also known as "mean pyramid" or "average pyramid." The most important advantage of this scheme in a GPU-based implementation is the possibility to compute this average by a bilinear interpolated texture lookup as illustrated by the gray dot for the pixel $G_1(1, 0)$ in Figure 1. Higher-order analysis filters could be implemented with the help of additional averaging steps that do not reduce the size of the filtered image.

For non-power-of-two dimensions of the original image, we suggest to copy edge pixels to fill an image of the smallest power-of-two dimension that is greater than the original dimension on the 0th level. Any other approach appears to be significantly more complicated considering the particular bilinear texture lookup for averaging four pixels.

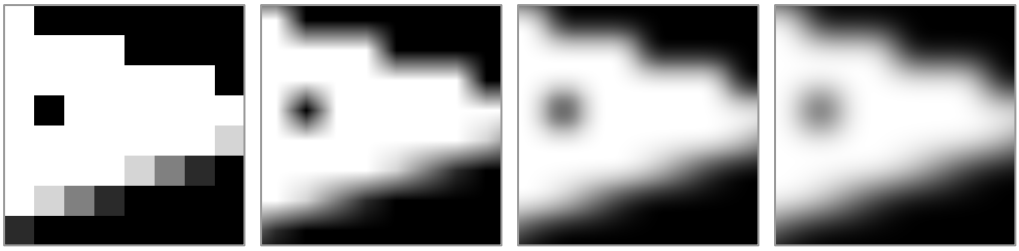
While our analysis operation does not access any pixels outside the actual pixels of each level, larger analysis filters (and also synthesis filters accessing this data) require additional pixel data outside of the computed image. Since standard clamping of texture coordinates cannot be employed for our layout scheme depicted in Figure 2, we simulate a clamp-to-edge approach by rasterizing additional pixels of a border around the image of each level with the same texture coordinates as for the adjacent edge or corner pixels. For our examples, this border is one pixel wide.

As an alternative to the proposed implementation of the analysis, the automatic mipmap generation defined in the "framebuffer object" OpenGL extension could be employed to build the initial image pyramid, provided that it does not result in additional data transfer between main memory and graphics memory. In fact, this alternative appears to perform slightly faster than the presented analysis on our target hardware. However, for an automatic mipmap generation the actual analysis filter is not specified and it is not customizable for higher-order analysis filters, alternative computations at edges, non-power-of-two images, etc.

3.2 Synthesis for B-Spline Filtering

The synthesis works top-down, computing new pixel data from pixels of the previous, coarser level. Again we rasterize a rectangle covering all pixels of one level and access pixels of the previous, coarser level by bilinear texture lookups with appropriate texture coordinates. A single bilinear texture lookup allows us to use the synthesis filter for biquadratic B-spline filters as illustrated in Figures 1 and 3b. The weights correspond to the subdivision of biquadratic B-spline patches and the regular Doo-Sabin subdivision [3].

Figure 4 demonstrates biquadratic B-spline filtering of pixel data in comparison to bilinear in-



(a) (b) (c) (d)

Figure 4: Filtering of image data: (a) piecewise constant, (b) piecewise bilinear interpolation (C^0 continuous), (c) biquadratic B-splines (C^1 continuous), (d) bicubic B-splines (C^2 continuous).

terpolation and bicubic B-spline filtering. The performance cost of bicubic instead of biquadratic B-spline filtering is rather modest in our implementation of the pyramid algorithm since an additional averaging step turns a biquadratic B-spline filtering into a bicubic B-spline filtering. However, a comparison between Figures 4c and 4d suggests that the C^2 continuity of bicubic B-splines is not a crucial advantage for image filtering while the stronger blurring and the worse approximation of the original data values are disadvantageous for many applications. Strong aliasing cannot be cured by neither type of B-spline filtering; however, biquadratic B-spline filtering of antialiased images already hides the regular sampling structure of the original image quite efficiently. Furthermore, the typical diamond-shaped artifacts of bilinear interpolation are completely avoided. Therefore, biquadratic B-spline filtering can provide a significantly improved visual quality compared to bilinear interpolation while the potential advantages offered by bicubic B-spline filtering strongly depend on the particular application.

The presented synthesis cannot perfectly reconstruct the original image from any of the coarse levels of the Gaussian pyramid since this pyramid does not provide all the required information and the Laplacian pyramid, which contains the missing information, is not computed in our analysis for performance reasons. However, the pyramid methods presented in this work either do not require this information for their purposes or retrieve the original data from the Gaussian pyramid, which is not overwritten during the analysis in our implementation.

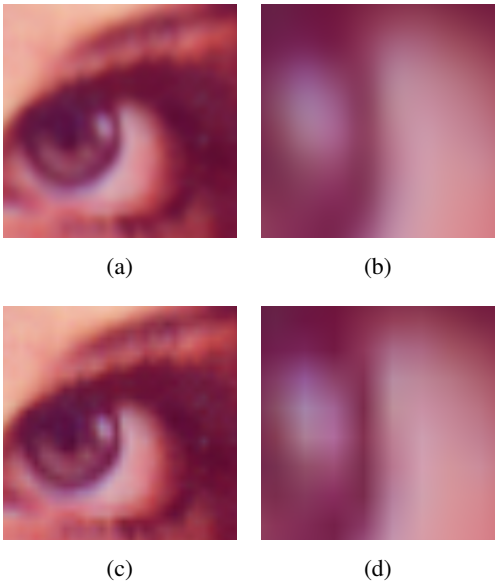
The bilinear interpolated texture lookup illustrated in Figure 3b requires valid pixel data of texels outside of the image of the coarser level if a boundary pixel of a finer level is rasterized. Thus, for the Gaussian level that is accessed for the first synthesis step, we have to ensure valid pixel data on a one-texel texture border; e.g., by copying boundary pixels after the computation of each level in the analysis. Note that the new one-texel border of the finer level can be easily computed by including these border pixels in the synthesis step and linearly extrapolating texture coordinates. This is possible since no additional texels of the coarser level are required for the synthesis of these pixels.

As demonstrated by the GPU-based pyramid methods described in Sections 4, 5, and 6 there are many useful variants of the presented basic pyramid method.

4 Pixel Zoom

We first discuss the application of our GPU-based implementation of the pyramid algorithm to pixel zooming. A particularity of a pyramid method for zooming is the lack of any analysis process. For a scale factor of 2^l our approach initializes level G_l by copying the pixels of the zoomed section of the original image into level G_l (including an additional one-pixel wide border). After this initialization all levels from G_{l-1} to G_0 are synthesized as discussed in Section 3.2. Since the analysis is not part of this method, the resulting image corresponds to a convolution of the pixel data with uniform biquadratic B-splines.

Figures 5a and 5b show zoomed sections of the 512×512 Lena image using scale factors 16 and 64, respectively; i.e., the original pixels are copied to level G_4 and G_6, respectively. The rendering time was 0.11 and 0.13 milliseconds for the two exam-



(a) (b) (c) (d)

Figure 5: Results of zooming the 512×512 Lena image with biquadratic B-spline filtering with scale factors 16 (a) and 64 (b) in comparison to bilinear texture magnification (c) and (d). The zoomed sections cover 32×32 *(left column)* and 8×8 *(right column)* pixels of the original image.

ples. More timings are given in Section 5, which discusses the synthesis step for image blurring, because the identical program sequence is used in both cases and therefore shows the same performance.

For comparison, Figures 5c and 5d employ a bilinear texture lookup to perform the same task, which results in strong artifacts typical for bilinear interpolation; especially in Figure 5d. Measurements were taken using the same conditions for better comparison; in particular, they were also performed while rendering into and reading from a 16 bit float offscreen buffer. The bilinear zoomed results were rendered in 0.07 milliseconds. The performance overhead of the pyramid method is therefore less than a factor of 2 compared to bilinear interpolation.

5 Image Blur

Our pyramid method for image blurring for a blur width of 2^l copies the original image into level G_0

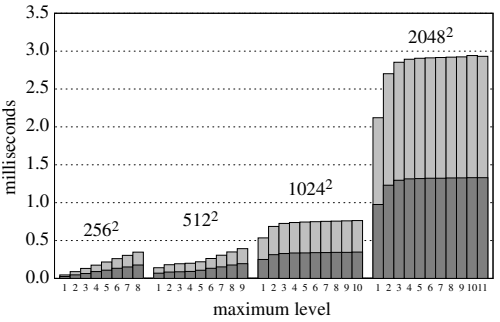


Figure 6: Timings for blurring scaled variants of the Lena image in milliseconds for the analysis steps *(dark gray)* and the synthesis steps *(light gray)*.

and computes l levels G_1 to G_l in the analysis process described in Section 3.1. For level G_l the one-pixel wide border is computed and the synthesis of levels G_{l-1} to G_0 is performed as discussed in Section 3.2.

The top row of Figure 8 (see color plate) shows the resulting images when applying this method to the 512×512 Lena image for various values of l. For comparison, the bottom row of Figure 8 shows the results of non-interactive applications of IIR Gaussian blur filters of comparable blur widths provided by the GNU Image Manipulation Program "Gimp."

As shown in Figure 4, filtering image data with B-splines of small filter widths cannot remove aliasing artifacts. Thus, if a level of the mean pyramid shows strong aliasing artifacts, they will become visible if the synthesis starts from this level; see, for example, Figure 8b. Figure 8f demonstrates that blurring by computing an actual convolution with a large filter avoids these aliasing artifacts.

Our analysis for average pyramids discussed in Section 3.1 does not avoid aliasing artifacts on any level; thus, it depends on the original image whether and which levels of the average pyramid feature aliasing. Analogously to the generation of mipmap textures, a wider analysis filter would improve the resulting visual quality.

Performance numbers of the pyramid image blurring for various image sizes are given in Figure 6. Although less fragments are generated during the analysis — level G_0 only acts as source texture — it requires about the same time as the synthesis due to the different access pattern of texture data. While

in the case of the analysis each texel of the source level is sampled by only a single fragment, the texel information can be reused up to sixteen times for the synthesis, which results in a more efficient usage of the texture cache.

The maximum total overhead required for buffer switches was about 0.4 milliseconds in our experiments. Apart from this overhead, the obtained timings clearly reflect the system's linear dependency on the number of input pixels. The blur width, which is determined by the maximum level used during the computation of the pyramid, has only little influence on the achieved overall performance; except for very small widths and small images where the overhead for buffer switches becomes the limiting factor.

In summary, our pyramid method requires less than 0.8 millisecond to blur 1 megapixel with arbitrary blur width, which is a suitable performance for real-time image postprocessing; e.g., for computer games and video processing. All reported numbers were taken on a 3.4 GHz Intel Pentium 4 and an NVIDIA GeForce 7800 GTX 512; `RGBA_FLOAT16_ATI` was used for all timings as format for the application-created framebuffer objects.

6 Interpolation of Scattered Data

Ogden et al. [9] and Burt [2] discuss pyramid methods for filling in missing pixels by extrapolating data in the analysis process and filling in synthesized data in the synthesis process. This kind of pyramid methods are also known as push-pull methods in the computer graphics community (see Lefebvre et al. [8] and references therein).

Our pyramid method for interpolation of scattered pixel data is based on averaging only known pixels in the analysis and filling in unknown pixels with synthesized pixel data in the synthesis as illustrated in Figure 7. The alpha component is employed to implement the "support image" described by Burt [2]: If a pixel is specified, alpha is set to 1. Otherwise, i.e., for unspecified pixels, alpha and all color components are set 0. When averaging four RGBA colors of this kind according to the analysis operation discussed in Section 3.1, the computed alpha component is set to $n/4$ with n being the number of known pixels among the four input pixels. Thus, the correctly averaged RGB components can

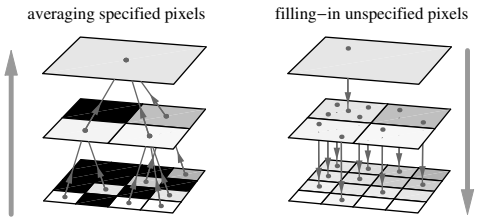


Figure 7: Interpolation of scattered pixel data: unspecified pixels *(black)* are ignored in the bottom-up analysis *(left)* and filled-in by bilinear texture interpolations in the top-down synthesis *(right)*.

be determined by a "homogeneous division" of the computed RGB components by the computed alpha component. However, it is not necessary to perform this division after each analysis operation since all computations of the analysis and synthesis can be performed in "homogeneous coordinates" if the division is applied to the resulting RGBA color in the last synthesis step.

Therefore, the analysis of our method is performed exactly as described in Section 3.1 while the synthesis has to test for each rasterized pixel whether the alpha component of this pixel in the Gaussian pyramid, which has been computed in the analysis, is greater than 0. In this case, the components of this pixel are *not* modified since it is considered a well-specified color (in homogeneous coordinates). Otherwise, i.e., if alpha is 0, the unspecified pixel is replaced by the synthesized pixel data. Moreover, when synthesizing level G_0, all resulting RGB components are divided by the alpha component.

Some results obtained with this GPU-based pyramid method are shown in Figure 9 (see color plate). Our approach smoothly interpolates between scattered pixels and also fills in unknown pixels. Since pixels are filled in on all levels of the Gaussian pyramid, the method also smoothly fills large areas of unspecified pixels as demonstrated in Figures 9g and 9h.

The fragment program implementing the synthesis operation of this pyramid method requires some additional arithmetic instructions and one additional texture lookup in comparison to the fragment programs for the methods presented in Sections 4 and 5; thus; the performance in our test en-

vironment (as described in Section 5) is about 0.41 milliseconds for the 512×512 images in Figure 9. Images of size 1024^2 and 2048^2 require about 1.4 and 5.3 milliseconds, respectively. For comparison, our OpenGL implementation of the push-pull interpolation by Lefebvre et al. [8] employed the automatic mipmap generation for the analysis, which offers a slight performance advantage. However, the total algorithm (including the synthesis) required 0.67, 2.5, and 10.0 milliseconds for the same images due to additional texture lookups.

7 Conclusion and Future Work

The three GPU-based pyramid methods presented in this work demonstrate that modern GPUs allow us to implement basic pyramid methods with their correct time complexity if bottlenecks such as data transfer between main memory and graphics memory are carefully avoided. Moreover, we showed that single bilinear interpolated texture lookups can be used to implement very efficient analysis and synthesis operations that avoid artifacts of bilinear texture interpolation and offer a sufficient visual quality for many applications.

All three of the presented pyramid methods can be extended in various ways: Our approach to zooming could be combined with higher-order filters and deferred filtering. The presented method for image blurring could be improved with the help of better analysis filters. Locally adaptive blurring would be an interesting feature with many applications in real-time computer graphics (depth-of-field, motion blur, soft shadows, etc.). An important enhancement of the presented interpolation of scattered pixel data would map the density of samples to intensity. Moreover, some applications require an approximation instead of an interpolation of samples. Further applications of GPU-based pyramid methods for image processing include the computation of summed area tables (generalizing the approach of Sengupta et al. [10] to images), tone mapping, motion estimation, feature enhancement, segmentation etc.

References

[1] Peter J. Burt. Fast filter transforms for image processing. *Computer Graphics and Image Processing*, 16:20–51, 1981.

[2] Peter J. Burt. Moment images, polynomial fit filters, and the problem of surface interpolation. In *Proceedings of Computer Vision and Pattern Recognition*, pages 144–152, 1988.

[3] Edwin Catmull and James Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978.

[4] Nolan Goodnight, Rui Wang, Cliff Woolley, and Greg Humphreys. Interactive time-dependent tone mapping using programmable graphics hardware. In *Rendering Techniques*, pages 26–37, 2003.

[5] Simon Green. Image processing tricks in opengl. Presentation at GDC 2005.

[6] Matthias Hopf and Thomas Ertl. Hardware accelerated wavelet transformations. In *Symposium on Visualization VisSym '00*, 2000.

[7] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[8] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *GPU Gems 2*, chapter Octree Textures on the GPU, pages 595–613. Addison Wesley, 2005.

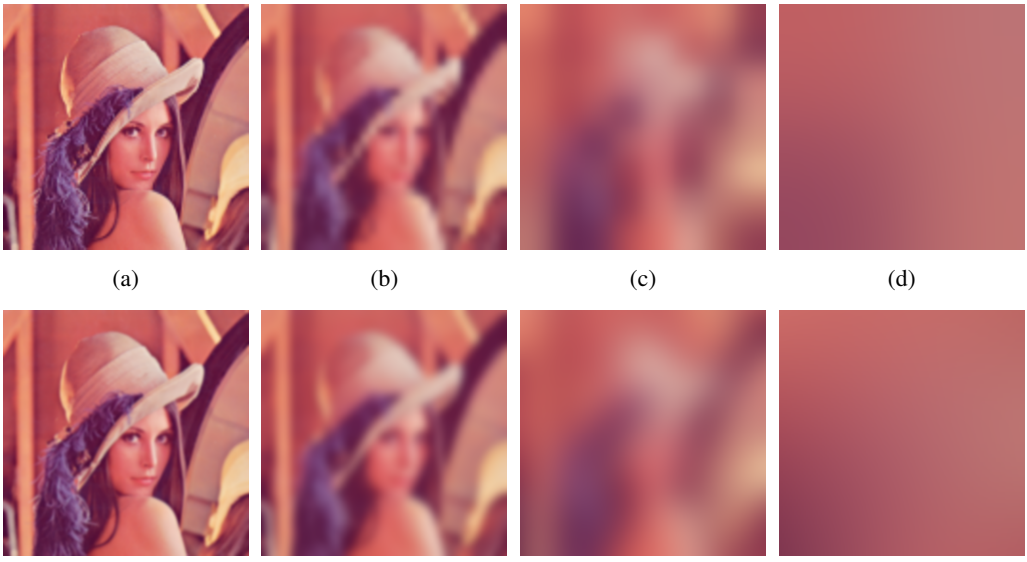
[9] Joan Ogden, Edward Adelson, James Bergen, and Peter Burt. Pyramid-based computer graphics. *RCA Engineer*, 30(5):4–15, 1985.

[10] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix-sum algorithm. In *Proceedings 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, 2006.

[11] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. In Matt Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 313–329, 2005.

[12] Christian Tenllado, Roberto Lario, Manuel Prieto, and Francisco Tirado. The 2d discrete wavelet transform on programmable graphics hardware. In *IASTED Visualization, Imaging and Image Processing Conference*, 2004.

[13] Jianging Wang, Tien-Tsin Wong, Pheng-Ann Heng, and Chi-Sing Leung. Discrete wavelet transform on gpu. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, 2004.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 8: *Top row:* Blurring of the 512×512 Lena image with our pyramid method. The images are synthesized from (a) level G_2, (b) level G_4, (c) level G_6 and (d) level G_8. *Bottom row:* For comparison we show the results of Gaussian IIR filters of comparable width in (e), (f), (g), and (h).

(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 9: *Top row:* variants of the 512×512 Lena image with unspecified pixels *(black)*. *Bottom row:* results of our pyramid method for interpolation of scattered pixel data.