Real-time, Accurate Depth of Field using Anisotropic Diffusion and Programmable Graphics Cards

Marcelo Bertalmío and Pere Fort *
Departament de Tecnologia
Universitat Pompeu Fabra

Daniel Sánchez-Crespo †
Universitat Pompeu Fabra - Novarama Technology

Abstract

Computer graphics cameras lack the finite Depth of Field (DOF) present in real world ones. This results in all objects being rendered sharp regardless of their depth, reducing the realism of the scene. On top of that, real-world DOF provides a depth cue, that helps the human visual system decode the elements of a scene. Several methods have been proposed to render images with finite DOF, but these have always implied an important trade-off between speed and accuracy. In this paper, we introduce a novel anisotropic diffusion Partial Differential Equation (PDE) that is applied to the 2D image of the scene rendered with a pin-hole camera. In this PDE, the amount of blurring on the 2D image depends on the depth information of the 3D scene, present in the Z-buffer. This equation is well posed, has existence and uniqueness results, and it is a good approximation of the optical phenomenon, without the visual artifacts and depth inconsistencies present in other approaches. Because both inputs to our algorithm are present at the graphics card at every moment, we can run the processing entirely in the GPU. This fact, coupled with the particular numerical scheme chosen for our PDE, allows for real-time rendering using a programmable graphics card.

1. Introduction

We can define the Depth of field (DOF) of a camera system as the range of distances, in front of and behind the point of focus, where the eye perceives the image to be sharp. This applies also to the human eye, which, having focused on a certain point, sees objects behind and in front of that point with a certain amount of blur. This blur increases as the objects are farther away from the focus distance. DOF is therefore an important aspect of visualization if we want the images to look *natural*, since the natural images that we see have always a finite DOF (that of our eyes). Also, a finite DOF provides the viewer with a useful *depth cue* which may be very helpful to interpret the geometry and relative position of the objects of the scene [16].

On the other hand synthetic images, 2D images rendered from a 3D scene, have *infinite* DOF, we see everything sharp, in focus. This stems from the fact that the camera model used for the rendering is the standard pin-hole camera, in which the diameter of the lens is zero (hence the name *pin-hole*.) Thus the rendering is very fast, at the expense of presenting everything in focus. Clearly, introducing the precise amount of (depth-dependent) blur to the image would slow things down, and this is the issue with algorithms for DOF rendering: they all have a very substantial trade-off between speed and accuracy. Applications such as video-games, interactive visualization or Virtual Reality navigation all would require real-time DOF rendering. But *fast* DOF algorithms (like those in [16] [13]) present severe visual artifacts, like *intensity leakage*, in which blurred objects leak intensity onto focused objects, or v.v. This is very distracting to the viewer, since it provides an inconsistent depth-cue. Conversely, *accurate* DOF algorithms (like those in [7] [12] [2] [5]) are computationally intensive, at most producing just a few frames per second, far from the requirements of the applications mentioned above.

Our contribution is the following. We present a novel anisotropic diffusion Partial Differential Equation (PDE). Given a 2D image rendered from the 3D scene with a standard pin-hole camera, and the depth buffer of the scene, this PDE performs accurate depth-dependent diffusion on the image using the depth information present in the Z-buffer. The numerical implementation of this equation (local computations, iterations) is especially well suited to the characteristics of the Cg programmable graphics hardware language, and all the operations are performed entirely in the Graphics Processing Unit (GPU), so we achieve DOF in real-time. We give details on the mathematical properties of the PDE (which is well-posed, with existence and uniqueness results), on the numerical scheme and on the Cg implementation.

* { marcelo.bertalmio, pere.fort }@upf.edu
† daniel.sanchez-crespo@upf.edu

2. A Depth of Field model

In any camera model the precise focus occurs only at the focusing distance, but the DOF, the *apparent* range of focus, can vary noticeably (depending on aperture size, focal length, etc.)

Every 3D point at the focusing distance will map (project) as a point onto the image plane. As the 3D point moves out of focus (away or closer), it will map as a circle, called the circle of confusion (CoC). See figure 1. The farther out of focus a 3D point is, the larger these circles become. But close to the focusing distance, this CoC's are small, the human eye can not resolve them, and the image appears to be sharp over a range of distances: this range is the DOF [15].

The following equation (an expansion of the one in [15]) gives us the diameter c of the CoC for a 3D point at a distance Z (the point's depth):

$$c = \alpha \frac{|Z - Z_f|}{Z} \tag{1}$$

where the constant α is computed as:

$$\alpha = \frac{F^2}{n \cdot (Z_f - F)} \tag{2}$$

and Z_f is the focal distance (the depth at which the image is in perfect focus), F is the focal length of the lens, and n is the aperture number. Notice that when $Z = Z_f$, the diameter of the CoC is zero, as expected. Also, in a first order approximation the CoC diameter varies linearly with the absolute value of $Z - Z_f$, which is the difference in depth between the 3D point and the focal plane.

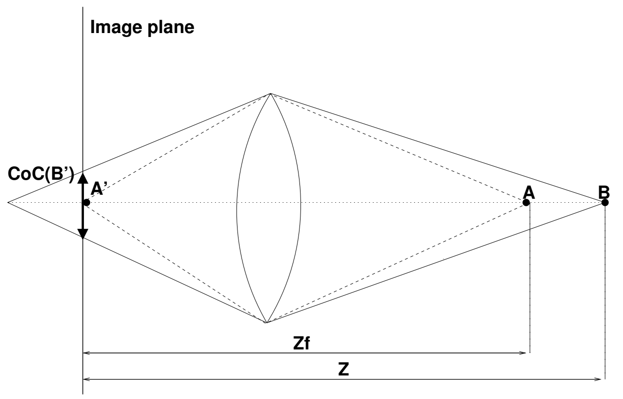

Figure 1. DOF model: *A* projects onto point *A'*, while *B* projects onto a circle *CoC(B')*.

In the following section, we will show how to approximate the CoC's with an anisotropic diffusion process, in which a depth-dependent blurring is performed on the 2D image rendered with a pin-hole camera.

3. PDE's and Anisotropic Diffusion

The use of PDE's to perform diffusion in image processing started with the seminal works of Koenderink [8] and Witkin [20], in which the classical heat equation, a second order PDE equivalent to Gaussian filtering, is used to obtain a multi-scale representation of images.

Later, Perona and Malik's [14] work on anisotropic diffusion became one of the most influential papers in the area. They proposed to replace Gaussian smoothing, equivalent to isotropic diffusion via the heat flow, by a selective diffusion that preserves edges. For a survey of PDE's and image processing see for instance [19].

3.1. The Heat Equation and Gaussian Kernels

The heat equation is a second order PDE:

$$\frac{\partial I}{\partial t}(x, y, t) = \Delta I(x, y, t) = \nabla \cdot (\nabla I(x, y, t)) = I_{xx} + I_{yy} \tag{3}$$

The n-th step of an explicit numerical implementation of this equation can be written as:

$$I^{n+1}(i, j) = I^n(i, j) + \frac{h^2}{8} \Delta I^n(i, j) \tag{4}$$

where $\frac{h^2}{8}$ is the time step.

It can be shown [4] that running one step of this equation on the image I is equivalent to convolving I with a Gaussian kernel of width h:

$$I^n = I^{n-1} * G_h \tag{5}$$

Also, running N successive iteration steps is equivalent (on limit) to convolving the initial I with a Gaussian kernel of width $\sqrt{2Nh}$:

$$I^N = I^0 * G_{\sqrt{2Nh}} \tag{6}$$

So we see that we can always approximate Gaussian blurring with a certain number of iteration steps of the heat equation. It can be shown that 0.25 is the greatest value for the time step that ensures stability of the equation [17], so we choose $\frac{h^2}{8} = 0.25$. This choice and equation (6) give us the following equivalence:

$$N = \frac{\sigma^2}{2\sqrt{2}} \tag{7}$$

where N is the total number of iterations of the heat equation that we must perform on the image I if we want to achieve the same result as if convolving I with a Gaussian kernel of width σ.

3.2. Our equation

We can introduce anisotropy in the heat equation, by means of a weighting function $g(x, y, t)$:

$$\frac{\partial I}{\partial t}(x, y, t) = \nabla \cdot (g(x, y, t) \nabla I(x, y, t)) \tag{8}$$

where g is a real-valued function between 0 and 1. For instance, if g decreases with the norm of the gradient of I, we get Perona and Malik's edge-preserving anisotropic diffusion [14].

Our goal is to perform anisotropic diffusion on I so that at each point the amount of blurring corresponds to the width of the CoC at that point. Eq. (1) says that, at each point, the width of the CoC depends only on the depth of that point. This tells us that we must choose g to be constant in time, but how should it vary with depth?

Let us look at a simple example: a region of 3D space with constant depth Z_c, say a square parallel to the image plane. This square will of course project onto a square on the image plane. Inside the square, the depth buffer will have the constant value Z_c, so g must be constant as well, and eq. (8) simplifies to:

$$\frac{\partial I}{\partial t}(x, y, t) = g \nabla \cdot (\nabla I(x, y, t)) = g \Delta(I(x, y, t) \tag{9}$$

The n-th step of an explicit numerical implementation of this equation can be written as:

$$I^{n+1}(i, j) = I^n(i, j) + g \frac{h^2}{8} \Delta I^n(i, j) \tag{10}$$

Also, for every point inside the square, the CoC will have diameter $\alpha \frac{|Z_c - Z_f|}{Z_c}$, where Z_f is the depth of the focal plane. If, as before, we fix $\frac{h^2}{8} = 0.25$, and we compare eqs.(10) and (4), we get from equations (10), (1) and (6) the following relationship:

$$g \propto (\frac{|Z_c - Z_f|}{Z_c})^2 \tag{11}$$

That is, this choice of g ensures that iteration of eq. (8) performs on I the same effect as *local* convolution with a Gaussian kernel the width of the CoC (notice that since the CoC varies with (x, y), the kernel width does as well, so we don't have the *same* Gaussian kernel for the whole image I, but one kernel for each point (x, y).)

From this, we choose for the general case our function g to be

$$g(x, y) = \alpha \cdot (\frac{|Z(x, y) - Z_f|}{Z(x, y)})^2 \tag{12}$$

where now α is a constant dependent on the camera parameters (and scaled so we keep g in the interval $[0, 1]$), $Z(x, y)$ is the depth-buffer value at (x, y), and Z_f is the focal distance.

Therefore, we propose equation (8) with g defined in eq. (12) as our PDE that performs depth-dependent blurring. This blurring is the same that we would obtain by local convolution with a Gaussian kernel the width of the CoC at each point, so our equation approximates a finite DOF effect on I.

This equation is well posed, and there always exists a unique solution for it. See for instance [9].

3.3. *Leakage* prevention

Our equation does not cause *intensity leakage*, which is one of the major problems in other fast approaches to DOF such as [16] and [13]. As we mentioned above, the term *intensity leakage* refers to the very distracting visual artifact in which blurred objects leak intensity onto focused objects, or v.v. In figure 2, obtained with the approach in [13], we see that the foreground object *leaks* onto the background: the whole foreground prism should be in focus, but its upper boundary is not sharp, it appears blurred, while the lower boundary and the inside of the same prism are indeed sharp.

In our equation, the weighting function g multiplies directly the gradient ∇I, and this is *not* the same as g multiplying the Laplacian ΔI:

$$\frac{\partial I}{\partial t} = \nabla \cdot (g \nabla I) = g \Delta I + \nabla g \cdot \nabla I \tag{13}$$

(we have omitted the variables x,y and t.) We see that there is a second term, $\nabla g \cdot \nabla I$. This is what prevents the intensities from *leaking* when we blur I, which will become apparent when we look at the numerical implementation, in the following subsection. If we dropped this term, we would have a modified heat equation,

$$\frac{\partial I}{\partial t}(x, y, t) = g(x, y) \Delta I(x, y, t) \tag{14}$$

in the spirit of both [16] and [13], and intensity leakage would be unavoidable. Why? Consider the following example: an image I where the left half (pixels (i, j) with $i < i_H$) corresponds to the foreground, in focus with $g = 0$, and the right half (pixels (i, j) with $i \geq i_H$) corresponds to the background, with $g = 1$, which should be blurred. With eq. (14), blurring is performed *isotropically* with the Laplacian ΔI, and *afterward* weighted by g. This means that any pixel in the background with $i = i_H$ will be blurred with contribution from its neighbors with $i = i_H - 1$, but this is precisely the definition of *intensity leakage*: the pixels with $i = i_H - 1$ are in focus, so they can not contribute to the blurring of any other pixel (their CoC's have zero diameter.)

See figure 3 for a comparison of the effects of eqs. (8) and (14). With our equation (bottom left image of the figure) the foreground object has sharp boundaries, as it should.

3.4. Numerical implementation

We have chosen an explicit numerical scheme for the implementation of equation (8). As in [18], we can not use a central differences approximation of the gradient nor the divergence, because it would be an unconditionally unstable scheme. We use instead backward differences for the divergence and forward differences for the gradient:

$$\begin{aligned}
&\nabla^+ u_{i,j} = (u_{i+1,j} - u_{i,j}, u_{i,j+1} - u_{i,j}) \\
&\nabla^- \cdot \overline{w}_{i,j} = u_{i,j} - u_{i-1,j} + v_{i,j} - v_{i,j-1} \\
&\overline{w}_{i,j} = (u_{i,j}, v_{i,j})
\end{aligned} \tag{15}$$

(these are just examples with generic scalar functions u and v and vector $\overline{w}$.)

The numerical scheme is then:

$$I^{n+1}_{i,j} = I^n_{i,j} + 0.25\nabla^- \cdot (g_{i,j}\nabla^+ I^n_{i,j}) \tag{16}$$

where the subscripts are the pixel coordinates and the superscript is the iteration step.

It is clear now why there is no *leakage* with this algorithm: if a neighbor of pixel (i, j) has $g = 0$, then it will *not* be averaged.

It must be noted that, while this behavior is the correct one when the foreground is in focus, it also implies that if the background is in focus instead and the foreground blurred then the foreground still has a sharp boundary. This would produce an awkward result, since a blurred object in the foreground should have blurred, not sharp, boundaries. This issue is easily solved by pre-processing g: we perform some isotropic diffusion on the pixels of g with depth lesser than the focal distance. This will smooth the sharp jumps in depth that happen along the boundaries of the foreground objects, while keeping sharp the boundaries at the focal plane. This pre-processing step is extremely fast, as we will see in the following section.

4. Programmable Graphics Hardware

In the last few years there have been a number of works that exploit the speed capabilities of graphics hardware to accelerate the solution of diffusion equations [6] [3] [1]. This is the logical conclusion of GPU speed growing faster than CPU speed: the graphics subsystem can today act as a geometry / image processing subsystem outperforming the CPU manyfold. GPUs are also cost-effective, present in all new PC's, and not tied to any specific operating system [6]. As a downside, GPUs are traditionally complex to code for, as several competing standards exist. This problem has been greatly reduced with the introduction of high-level programming languages for GPUs. Cg [10] is one of the best-known examples, which has been used in the implementation phase of our algorithm. We have decomposed our equation processing into a series of pixel shaders, which perform the Anisotropic Diffusion inside the GPU.

Our algorithm is very well suited for such an approach. First, the process is highly parallel: each pixel in the scene requires similar processing. Second, all data required for the actual processing can be stored in GPU memory, thus reducing drastically the bus traffic and increasing performance. Recalling from previous sections, all we need as inputs to our algorithm are the 2D image (or 3D scene saved as a texture) and the depth-buffer of the scene. Both inputs are present at every moment in the GPU, which allows for this algorithm to run *entirely* in the GPU. This is very adequate, since communication with the CPU through de data bus would hinder performance.

We use three internal textures, the same size of the image I: one stores g, which is fixed through all the iterations, while the other two textures are used to update I, and they change at every iteration. This makes the GPU memory requirements for our method reasonable. We found out that the bottleneck of the actual processing is not in the computations, but in the texture-lookup operations. The numerical scheme in (16) would require, in a direct implementation in Cg, eight texture-lookups: five for the current pixel $I(i, j)$ and its four neighbors, three for $g_{i,j}$, $g_{i-1,j}$, and $g_{i,j-1}$. When we coded the three values $g_{i,j}$, $g_{i-1,j}$, and $g_{i,j-1}$ in the same 32-bit number (by coding each value in 8 bits, leaving the alpha-channel unused) and stored this number in $g_{i,j}$, then we could access the three values with just one texture-lookup, and performance was increased almost 30%.

The fact that texture operations are the bottleneck explains why our algorithm can achieve real-time performance, while a direct computation of the Gaussian blurring is at present far from real-time applicability. Let us only count the number of texture-lookups required in both approaches in order to achieve the same result. In our algorithm, we require $6N$ lookups, where N is the number of iteration steps. A direct computation of Gaussian blurring would imply, for each pixel, adding the contributions of its neighbors (weighted by their respective g). If we choose 2σ as the radius of the neighborhood (so it will contain most of the kernel,) there will approximately be $4\Pi\sigma^2$ pixels in it. For each of those pixels, we will make two lookups, one for I and one for g, so the total number of lookups in this approach will be $8\Pi\sigma^2$. Recalling eq. (7), which relates N and σ, we get that with our algorithm we make just $\frac{1}{12}$ of the number of lookups that direct Gaussian blurring would require. Hence our method will approximately be one order of magnitude faster than direct Gaussian blurring.

As for the pre-processing of g which we mentioned at the end of the previous section, it is done in the following way. Given the original depth-buffer Z we compute g, call it $g1$, from equation (12). [1] Next we make a copy of $g1$, call it $g2$, in which we threshold all the values corresponding to depths greater than that of the focal plane. We isotropically blur $g2$ by averaging reduced scale versions of it (a technique similar to mip-mapping [11]). Then we combine $g1$ and $g2$ to obtain the g that we will use: the idea is to choose the value in $g2$ for pixels in the front of the focal plane, and $g1$ for pixels in the back. This combination is done non-linearly, to achieve a smoother blending. This pre-processing step is extremely fast, since the isotropic diffusion is actually performed by power of two scaling. Figure 4 shows $g1$ and the final g obtained."

5. Results

Table 1 shows the performance of our algorithm in frames per second (fps) for different values of screen resolution and number of iterations. We used a NVIDIA GEFORCE FX 5950 Ultra graphics card. We see for instance that, at TV-resolution ($640x480$), we get 21.7 fps with 10 iterations. This number of iteration steps is usually enough to achieve a realistic DOF effect. We can always increase the blurring, but it will be at the expense of the frame rate, as one would expect.

Table 1. Performance in frames per second as a function of screen resolution R and number of iterations N.

R/N	5	10	15	20
640x480	35.7	21.7	15.4	11.9
800x600	21.5	13.4	9.7	7.7
1024x768	13.5	9.0	6.0	4.9

Figure 5 shows details of several screen-shots of a real-time demo of our algorithm, where N is fixed at 20 iterations per frame and the screen resolution is $640x480$. Notice how there are no visual artifacts of any kind.

6. Conclusion and Future Work

In this article, we have introduced a novel PDE that approximates finite DOF on synthetic images by performing depth-dependent anisotropic blurring. This blurring is performed on a 2D image rendered with a pin-hole camera from a 3D scene. The inputs are the 2D image and the depth-buffer of the scene, which allows for this algorithm to run entirely in the GPU. The PDE is mathematically well behaved, with existence and uniqueness of solutions. Its numerical implementation is accurate, unlike previous real-time approaches to this same problem. Crucially, it is well suited to the parallel architecture of the GPU, allowing for real-time performance.

Currently we continue to optimize the performance of our method, mainly by looking for sources for speed-ups in the Cg code. In the future we will address the transparency of scene objects, which at present we are not taking into account.

GPU-based PDE solutions such as the one exposed in this paper show great potential for the future: effects like motion-blur, halos, and many others can be simulated using variants of our core algorithm. Still, these effects put a lot of stress on certain components of the GPU. We have seen how both the texture lookup units and, more broadly, the pixel shader processor, need to increase their computational power in order to allow for a broader spectrum of real-time cinematic effects.

[1] Actually, we use a linearized version of the equation, which introduces no visible error.

Acknowledgements

The authors would like to thank Vicent Caselles and Andrés Almansa for their suggestions and helpful comments, Josep Blat for his support, Serrana Cabrera for her assistance, and Lucas Bertalmío for being a permanent source of inspiration. We would also like to thank NVIDIA Corporation for supporting our research unconditionally both with the hardware and know-how needed to make this paper possible. This work was supported by European Commission Project IST-2001-34337, PNPGC Project reference BFM2003-02125, and CICYT BFM2000-0962-C02-01.

References

[1] P. Colantoni, N. Boukala and J. Da Rugna " Fast and accurate color image processing using 3D graphics cards" *Proceedings VMV*, Munich, Germany, November 19-21, 2003.

[2] R. Cook, "Stochastic sampling in Computer Graphics," *ACM Transactions on Graphics*, Vol. 5, No. 1, pp. 51-72, January 1986.

[3] U. Diewald, T. Preusser, M. Rumpf and R. Strzodka, " Diffusion models and their accelerated solution in computer vision applications" *Acta Mathematica Universitatis Comenianae (AMUC)*, 70(1):15–31, 2001

[4] F. Guichard and J.M. Morel, *Image iterative smoothing and P.D.E.'s*, Book in preparation, 2000.

[5] P. Haeberli and K. Akeley, "The Accumulation Buffer: hardware support for high-quality rendering," *Proceedings ACM SIGGRAPH*, 1990.

[6] M. Harris, G. Coombe, T. Scheuermann and A. Lastra, "Physically-based visual simulation on graphics hardware," *Proceedings Siggraph/Eurographics Workshop on Graphics Hardware*, 2002.

[7] J. Kȓivánek, J. Ẑára and K. Bouatouch, "Fast Depth of Field rendering with surface splatting," *Proceedings of Computer Graphics International 2003*, 2003.

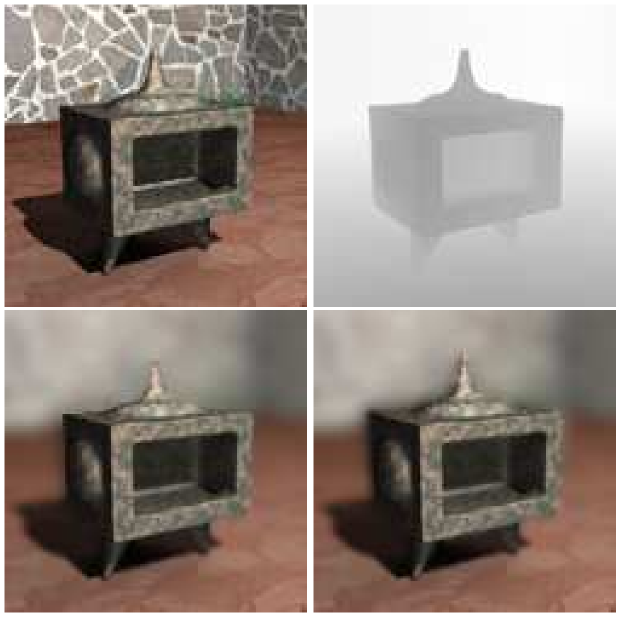

Figure 3. Top: original image (left) and Z-buffer (right.) Bottom: our method (left) produces no *leakage*.

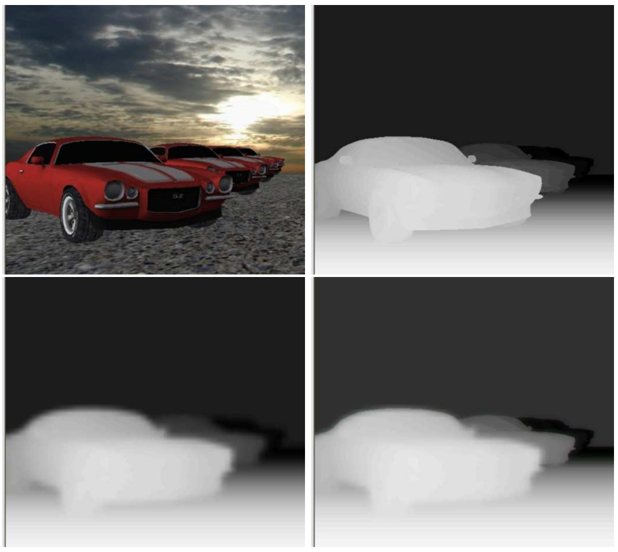

Figure 4. Pre-processing of g: original image(top left), $g1$ (top right), $g2$ (bottom left) and final g (bottom right) when focusing on 2nd row of cars (see text for details.)

[8] J. J. Koenderink, "The structure of images," *Biological Cybernetics* **50**, pp. 363-370, 1984.

[9] O. A. Ladyzhenskaya "The boundary value problems of mathematical physics," *Springer Verlag*, Theorem 4.1, 1985.

[10] W. R. Mark, R. S. Glanville, K. Akeley and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *Proceedings ACM SIGGRAPH*, 2003.

[11] L. Williams, "Pyramidal Parametrics," *Proceedings of ACM SIGGRAPH 83*, Vol. 17, No. 3. 1-11, 1983.

[12] J. D. Mulder and R. van Liere, "Fast perception-based Depth of Field rendering," *K. Y. Wohn, editor, VRST 2000*, pp. 129-133, 2003.

[13] NVIDIA Real-time DOF demo with Cg: `developer.nvidia.com/object/depth_field.html`

[14] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE-PAMI* **12**, pp. 629-639, 1990.

[15] M. Potmesil and I. Chakravarty, "A lens and aperture camera model for synthetic image generation," *Proceedings ACM SIGGRAPH*, 1981.

[16] P. Rokita, "Generating depth of-field effects in virtual reality applications" *Computer Graphics and Applications, IEEE*, Volume: 16 , Issue: 2 , Pages:18 - 21, March 1996.

[17] B. Romeny, Editor, *Geometry Driven Diffusion in Computer Vision*, Kluwer, The Netherlands, 1994.

[18] L. Rudin, S. Osher and E. Fatemi. *Nonlinear total variation based noise removal algorithms*. Physica D, 60, pp. 259-268, 1992.

[19] G. Sapiro, *Geometric Partial Differential Equations and Image Analysis*, Cambridge University Press, Cambridge-UK, 2001.

[20] A. P. Witkin, "Scale-space filtering," *Int. Joint. Conf. Artificial Intelligence* **2**, pp. 1019-1021, 1983.

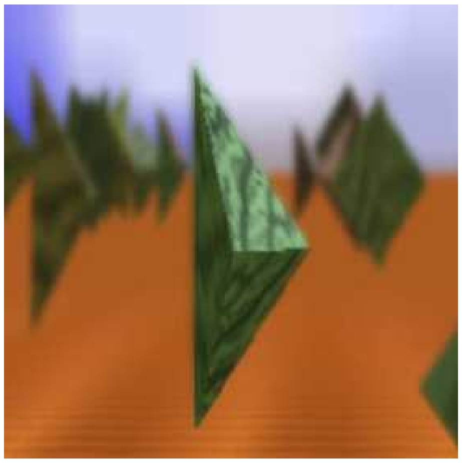

Figure 2. NVIDIA DOF demo [13].Notice *leaking* at the upper boundaries of the central prism.

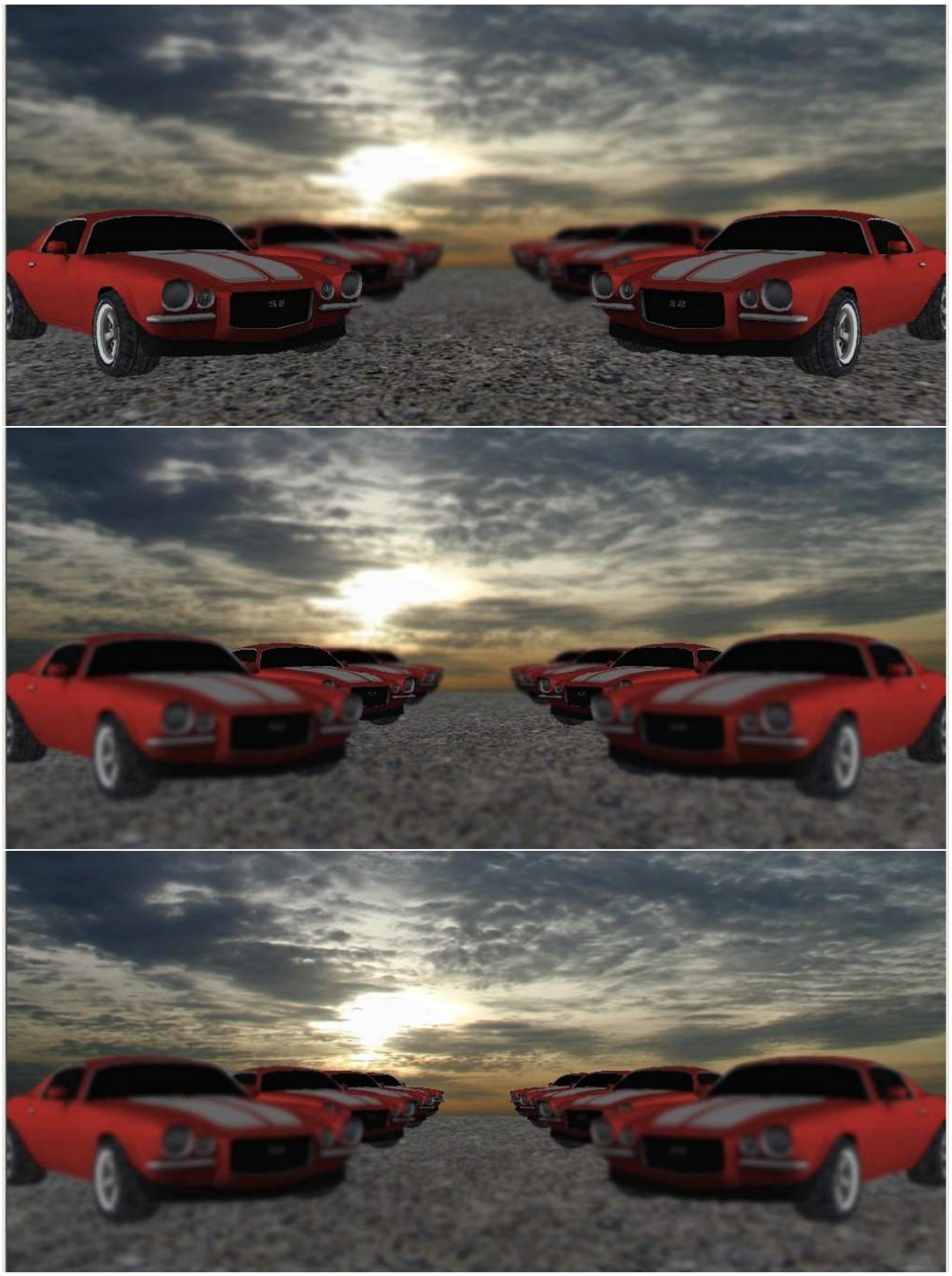

Figure 5. Examples of our DOF algorithm: focus on front (top), focus on 2^{nd} row of cars (middle), focus on rear (bottom.)