Ibero-American Symposium on Computer Graphics - SIACG (2006)
P. Brunet, N. Correia, and G. Baranoski (Editors)

Realistic depth of field effects with OpenGL

Daniel Berjón[1] and Francisco Morán[2]

berjon@dit.upm.es, fmb@gti.ssr.upm.es
[1]Grupo de Sistemas de Tiempo Real, Depto. de Ingeniería de Sistemas Telemáticos
[2]Grupo de Tratamiento de Imágenes, Depto. de Señales, Sistemas y Radiocomunicaciones

E.T.S. de Ingenieros de Telecomunicación
Universidad Politécnica de Madrid
E-28040 Madrid, Spain

Abstract

This paper describes a general method for simulating realistic depth of field effects by first projecting a 3D scene with OpenGL and then performing a defocusing *step through a simple local averaging filter. OpenGL yields perfectly focused images which are not realistic, but is efficient (especially if hardware acceleration is available) and provides, through the depth-buffer, information on how distant the 3D point projected on each pixel of the frame-buffer was from the camera. Thanks to this information, we know how much we must* defocus *each pixel, whose colour we spread over a circle of confusion surrounding it.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

1. Introduction

OpenGL [SWND06] uses a simple projection model which transforms points in a 3D space into points located on the projection plane on a one-to-one basis. This model, although mathematically simple, is not adequate to obtain realistic images, for every object in the scene remains focused independently of its distance from the camera. No real device can produce completely sharp images since every photosensitive device, be it analog or digital, needs some luminous energy in order to work. The OpenGL projection model would be equivalent to a pin-hole camera with an infinitely small orifice, which would therefore only let in an infinitesimal amount of light, hence needing an infinite exposure period, all of which is far from realistic.

All practical camera-like devices, including eyes, adopt a different strategy: they let light in through a relatively large aperture in order to increase the amount of energy reaching the sensor, but to do so they must use a lens to focus the luminous rays onto the projection surface. This comes at a price: only objects which are an exact distance away from the camera are strictly focused, all other objects being more or less out of focus.

In this paper we will discuss the optical thin lens model as a more accurate yet relatively simple model of a real camera and will show how it can be implemented as a linear non-invariant post-processing filter for OpenGL 3D rendering programs.

2. Thin lens model

Throughout this paper we will assume that we are trying to simulate the optical model known as *thin lens model* [Tip95], which means that we are dealing with a single lens whose thickness is negligible compared to its focal lengh. This allows to make some minor mathematical simplifications which ease calculations without introducing too much error. We will assume also that there are no second-order effects such as spherical and chromatic aberrations or coma, since that would add unnecesary complexity to our analysis. In fact, every good photographic lens tries to banish these effects. Why would we want to simulate a bad photographic lens?

Since we are not trying to make a physical simulation but rather an emulation of the results, we do not need to know physical parameters of the lens such as its refraction index or the curvature of its surfaces. We only need to know its focal distance. Knowing the focal distance, it is easy to calculate where all the rays coming from a point in space will converge, thanks to the *thin lens equation*:

$$\frac{1}{S_1} + \frac{1}{S_2} = \frac{1}{f} \tag{1}$$

The most relevant meaning of this equation is that if we set an object at a distance S_1 in front of a lens with focal distance f, every point will converge in another point located at a distance S_2 behind the lens, provided that both S_1 and S_2 are greater than f.

Although this is all we need to know whether an object will be focused in any particular scene, we also want to know how other points at any other distances will project onto the projection plane.

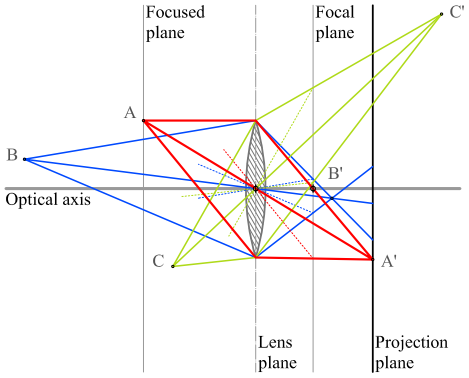

Figure 1: *Geometrical ray-tracing method for a thin lens*

Figure 1 shows how each ray passing through the lens behaves. For any arbitrary point at any distance of the lens, every ray will lay within the segment defined by the projections of the two rays passing through both ends of the lens. Since what is shown in the figure is just any longitudinal section, it is easily seen that every non-focused point in 3D space will turn into a circular spot –in fact, the same shape as the lens–, usually known as *circle of confusion*. This spot lies on the projection plane and its size depends on the distance of the corresponding 3D point from the lens.

Figure 2 shows trajectories of rays coming from an arbitrary point and how we can determine the size of any *circle of confusion*. The y coordinate for the main ray, which is the only ray that OpenGL computes, is:

$$y_{P'} = -s \cdot tan(\alpha) \tag{2}$$

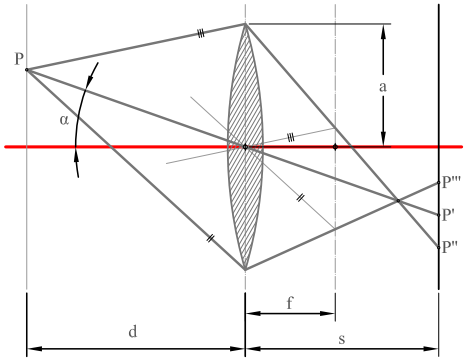

Figure 2: *Analysis of the size of the circle of confusion*

And y coordinates for rays passing through both ends of the lens are:

$$y_{P''} = \frac{\frac{a - d \cdot tan(\alpha)}{d} \cdot f - a}{f} \cdot s + a \tag{3}$$

$$y_{P'''} = \frac{-\frac{a - d \cdot tan(\alpha)}{d} \cdot f + a}{f} \cdot s - a \tag{4}$$

Operating and simplifying, we find that the radius of the circle of confusion does not depend on the angle α but only on the distance from the point to the lens:

$$|y_{P'} - y_{P''}| = |y_{P'} - y_{P'''}| = |a \cdot [s \cdot (\frac{1}{f} - \frac{1}{d}) - 1]| \tag{5}$$

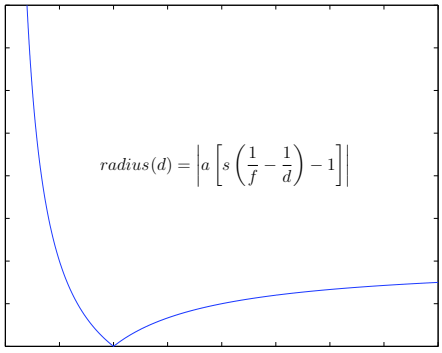

Figure 3: *Radius of the circle of confusion vs. distance*

Figure 3 shows how the radius of the circle of confusion varies as a function of distance to the lens. Strictly speaking, only the points at a specific distance are focused; however, points whose circle of confusion radius is below the resolution of the representation system are indistinguishable from

truly focused points. This phenomenon is usually known by photographers as *depth of field*.

3. Implementation as a linear filter

An implementation as a kind of Monte Carlo simulation has been proposed by the authors of the official OpenGL programming guide [SWND06] consisting in taking many images of the scene from points chosen over the cross-section of the lens with slightly different projection volumes, so that every point not in focus will have a different parallax depending on its distance to the viewpoint. The final image is calculated as an average of all of these images.

Although this approach is theoretically valid and it can be shown that it is optically equivalent to the thin lens model, we feel that it is inadequate to truly simulate the continuous nature of circles of confusion. As figure 3 shows, circles of confusion of points near the camera can be very large. Using this method, each image contributes with just one pixel for each point in space, so completely covering a big circle may require thousands of images, which is not very practical. In the vast majority of cases, this method yields a result more like a diffusion filter than like a truly unfocused image.

We have shown how every point in space turns into a circle on the projection plane whose centre is the point where the main ray intersects the projection plane. Since OpenGL calculates exactly that, just the main rays projections, we can understand the *defocusing* process as a post-processing filter to be applied onto the original image in which each point has its own associated *point spread function* (PSF) or impulse response.

Therefore the final image would be calculated as the sum of all the convolutions of each original point with its associated impulse response. For this to be physically accurate, the photosensitive sensor, be it chemical or electronic, should have a linear response, which is approximately true in the usual range of utilisation of such devices: every photographer knows that identical exposure can be obtained doubling the shutter time and closing the diaphragm one f-stop.

Since not every point in our original image will be convolved with the same impulse response (i.e. the filter is non space-invariant), we cannot resort to frequency filtering and we must perform true convolutions for every point.

3.1. Implementation

For the time being, we have implemented this as a sequential program run on the CPU, but we plan for the near future to implement it as a parallel fragment-shader to be run on the graphics adapter GPU, which will be much more efficient as it will eliminate the need to dump the frame- and depth-buffers onto main memory and back.

3.1.1. Obtaining distances to the lens

Fortunately, OpenGL uses and stores this very information to perform occlusion tests in the Z-buffer, so we can easily dump it onto main memory. OpenGL stores distances as floating-point normalized values. The near clipping plane maps to 0.0 and the far clipping plane maps to 1.0, but this mapping is not linear, so we need to denormalize it following the equation:

$$d = \frac{z_{norm} \cdot (z_{far} - z_{near}) + z_{near} \cdot z_{far}}{z_{far}} \tag{6}$$

3.1.2. Drawing circles of confusion

The simplest algorithm to draw a circle around a given pixel is to consider that every pixel whose distance from the centre is less than the radius belongs to it, and every pixel whose distance is greater does not. This is quite inefficient but can be improved at little cost. We need only consider pixels belonging to a bounding box, or *bounding square*, more precisely, which should be all those whose vertical and horizontal distances to the centre of the circle are both less or equal than its radius, and since those distances can only be integer, less or equal than the ceiling of the radius.

We are distributing the energy e contained in each original pixel over a circle or radius r, so each pixel belonging to the circle should have the value $e/\pi r^2$. However this does not yield really good results for two reasons.

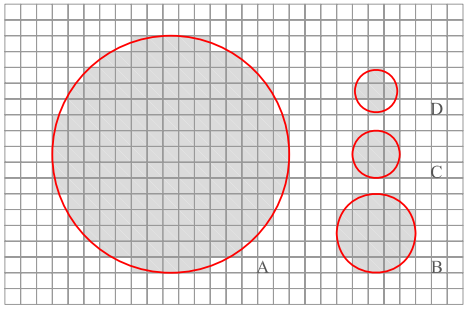

Figure 4: *Aliased circles*

As it can be clearly be seen in figure 4, small variations in the radius of the circle may result in much greater variations of the number of pixels involved, specially in areas close to the focusing distance which have very small circles of confusion (cf. cases C and D). These variations of the number of pixels mean not so subtle variations of luminance which are perceived as some sort of moiré patterns. A second reason for these moiré patterns is spectral folding. Considering that pixels must belong entirely or not at all to the confusion circle yields circles with hard, jagged edges, which are thus not band-limited in frequency, causing these undesirable effects.

Generally speaking, this should be solved by oversampling every pixel in the bounding square while drawing the point spread function, whose intensity could be non constant, then low-pass filtering and decimating the results. If a pixel is subdivided (supersampled) in, e.g. 10×10 subpixels, each of which are tested to belong or not in a binary way to the circle of confusion, then it is easy to calculate the percentage of the original pixel covered by the circle by just counting the number of covered subpixels.

Fortunately, if we are working with a piecewise constant function as is the case we can use an adaptive supersampling scheme as shown in figure 5. Pixels which are completely into the circle will yield the same value no matter how hard we try to supersample them, and the same applies to pixels which are completely outside the circle, therefore it is only necessary to supersample it in those pixels whose centres are less than $\sqrt{2}/2$ pixel units apart from the edge of the ideal circle.

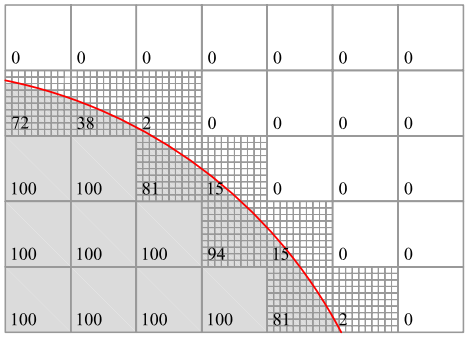

Figure 5: *Adaptive supersampling scheme for antialiasing. Numbers indicate the percentages of pixel covering*

In addition to the uniform scheme described above we have also tried *jittered sampling*, an irregular scheme which is a relatively low-cost approximation to the *Poisson-disk* distribution. This distribution is found among the sparse retinal cells outside the fovea and turns structured aliasing artifacts into random noise which is well accepted by the human visual system. Thus we should be able to use fewer samples to obtain similar subjective image quality [Wol90]. However, experimental results have shown that in our case the number of necessary samples is the same order of magnitude. Furthermore, we have found the cost of generating random numbers, be it using the standard C `rand()` function or using a higher quality generator such as the *Mersenne Twister* algorithm [MN98], to be quite high and add significant processing overhead unless using a precomputed set of numbers.

4. Conclusions and future work

We have shown that it is possible to implement a realistic defocusing algorithm with relative simplicity: we first let OpenGL efficiently project a 3D scene onto a perfectly (and hence, not at all realistically) focused image, which we then *defocus* in a post-filtering step by distributing the energy (i.e., 'spreading the colour') of each of its pixels over a circle of confusion. For each pixel (i, j) of the frame-buffer, we calculate the size of its circle of confusion thanks to the information stored in the same (i, j) position of the depth-buffer, which tells us how far apart from the camera the corresponding 3D point was and, therefore, how unfocused it should have been.

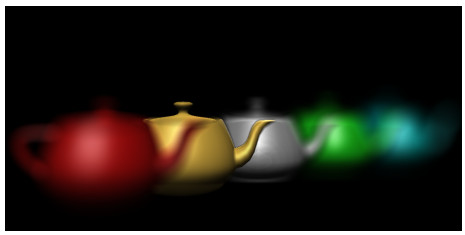

Figure 6: *Demo scene*

We have found our results to be quite accurate in photographic terms in spite of the very simple optical model we have used. However, we have also found that this algorithm is rather CPU intensive, so in order to make it really practical we intend to port it to the Shading Language included in OpenGL v2.0. By doing so we will spare the cost of moving data outside the graphics adapter and we will also benefit of the vastly superior floating-point capabilities nowadays present in such devices.

There is also an obvious flaw present in the specular highlights of objects due to the clamping of color values performed by OpenGL. Since the maximum permitted value for any channel under OpenGL is 1.0, when defocusing such a highlight the result is a much darker circle of confusion. Real-world highlights are often so bright that even when unfocused they remain so in relation with its surroundings. In order to solve these issue we are also experimenting with *high dynamic range* (HDR) rendering.

References

[MN98] MATSUMOTO M., NISHIMURA T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (Jan. 1998), 3–30.

[SWND06] SHREINER D., WOO M., NEIDER J., DAVIS T.: *OpenGL Programming Guide: The official guide to learning OpenGL, version 2*. Addison-Wesley, 2006.

[Tip95] TIPLER P. A.: *Física, 3^a edición*. Ed. Reverté, 1995.

[Wol90] WOLBERG G.: *Digital Image Warping*. IEEE Computer Society Press, 1990.