ISSN 2464-4617(print)
ISSN 2464-4625(CD)
Computer Science Research Notes
CSRN 2803
Poster's Proceedings
http://www.WSCG.eu

Realistic Lens Distortion Rendering

Martin Lambers
Computer Graphics Group
University of Siegen
Hoelderlinstrasse 3
57076 Siegen
martin.lambers@
uni-siegen.de

Hendrik Sommerhoff
Computer Graphics Group
University of Siegen
Hoelderlinstrasse 3
57076 Siegen
hendrik.sommerhoff@
student.uni-siegen.de

Andreas Kolb
Computer Graphics Group
University of Siegen
Hoelderlinstrasse 3
57076 Siegen
andreas.kolb@
uni-siegen.de

ABSTRACT

Rendering images with lens distortion that matches real cameras requires a camera model that allows calibration of relevant parameters based on real imagery. This requirement is not fulfilled for camera models typically used in the field of Computer Graphics.

In this paper, we present two approaches to integrate realistic lens distortions effects into any graphics pipeline. Both approaches are based on the most widely used camera model in Computer Vision, and thus can reproduce the behavior of real calibrated cameras.

The advantages and drawbacks of the two approaches are compared, and both are verified by recovering rendering parameters through a calibration performed on rendered images.

Keywords

Lens distortion, Camera calibration, Camera model, OpenCV

1 INTRODUCTION

In Computer Graphics, the prevalent camera model is the pinhole camera model, which is free of distortions and other detrimental effects. Real world cameras, on the other hand, use lens systems that lead to a variety of effects not covered by the pinhole model, including depth of field, chromatic aberration, and distortions. This paper focusses on the latter.

In Computer Vision, distortions must be taken into account during 3D scene analysis. A variety of camera models have been suggested to model the relevant effects; Sturm et al. [1] give an overview. The dominant model in practical use is a polynomial model based on the work of Heikkilä [2, 3] and Zhang [4] and is implemented in the most widely used Computer Vision software packages: OpenCV [5] and Matlab/Simulink [6]. In the following, we refer to this camera model as the standard model. Typical Computer Vision applications estimate the distortion parameters of the standard model for their camera system in a calibration step, and then undistort the input images accordingly before using them in further processing stages.

For a variety of applications, including analysis-by-synthesis techniques [7], sensor simulation [8], and special effects in films [9], it is useful to apply the reverse process, i.e. to synthesize images that exhibit realistic distortions by applying a camera model. Using the standard model for this purpose has the advantage that model parameters of existing calibrated cameras can be used directly, with immediate practical benefit to all application areas mentioned above.

In this paper, we present and compare two ways of integrating realistic distortions based on the standard camera model into graphics pipelines. One is based on preprocessing the geometry, and the other is based on postprocessing generated images. We show that both methods have unique advantages and limitations, and the choice of method therefore depends on the application. We verify both approaches by showing that standard model calibration applied to synthesized images recovers the distortion parameters with high accuracy.

2 RELATED WORK

In Computer Graphics, camera models that are more realistic than the pinhole model are typically based on a geometric description of the lens system that is then integrated into ray tracing pipelines [10, 11]. This approach is of limited use if the goal is to render images that match the characteristics of an existing camera, as suitable parameters cannot be derived automatically. Furthermore, this approach excludes rasterization pipelines, which is problematic for applications that benefit from fast image generation.

ISBN 978-80-86943-42-8
https://doi.org/10.24132/CSRN.2018.2803.4

In contrast, using a Computer Vision camera model allows to apply parameters obtained by calibrating a real camera and, as shown in Sec. 3, can be done in any graphics pipeline.

Sturm et al. [1] give an overview of camera models in Computer Vision. Most models account for radial distortion (e.g. barrel and pincushion distortion, caused by stronger bending of light rays near the edges of a lens than at its optical center) and tangential distortion (caused by imperfect parallelism between lens and image plane). Some also account for thin prism distortion (caused by a slightly decentered lens, modeled via an oriented thin prism in front of a perfectly centered lens), and tilted sensor distortion (caused by a rotation of the image plane around the optical axis).

The complete formulas for the standard model [5] compute distorted pixel coordinates from undistorted pixel coordinates and use parameters $k_1,\ldots,k_6$ for radial distortion, $p_1\,p_2$ for tangential distortion, $s_1,\ldots,s_4$ for thin prism distortion, and τ_1,τ_2 for tilted sensor distortion.

In practice, thin prism distortion and tilted sensor distortion are usually ignored, and radial distortion is limited to two or at maximum three parameters (the others are assumed to be zero). This is documented by the fact that the calibration functions of OpenCV[1] and Matlab/Simulink[2] estimate only the parameters k_1,k_2,p_1,p_2 and optionally k_3 by default.

In the following, we focus on the standard camera model of Computer Vision, and apply it to arbitrary rendering pipelines via either geometry preprocessing or image postprocessing.

3 METHOD

We first summarize the standard model in Sec. 3.1, focussing on the aspects relevant for this paper and incorporating its intrinsic camera parameters into the projection matrix of a pinhole camera model. On this basis, simulating lens distortion can be done in one of two ways:

- By *preprocessing geometry*. In this approach, each vertex of the input geometry is manipulated such that its position in image space after rendering corresponds to a distorted image.
- By *postprocessing images*. In this approach, an undistorted image is rendered based on the pinhole camera model, and distorted in a postprocessing step based on the standard model.

These approaches are described in detail in the Sec. 3.2 and Sec. 3.3.

[1] https://docs.opencv.org/3.4.0/dc/dbb/tutorial_py_calibration.html

[2] https://mathworks.com/help/vision/ug/camera-calibration.html

```
vec4 clipCoord = P * position;
vec2 ndcCoord = clipCoord.xy / clipCoord.w;
vec2 pixelCoord = vec2(
    (ndcCoord.x * 0.5 + 0.5) * w,
    (0.5 - ndcCoord.y * 0.5) * h);
// apply the standard model to pixelCoord
ndcCoord.x = (pixelCoord.x / w) * 2.0 - 1.0;
ndcCoord.y = 1.0 - (pixelCoord.y / h) * 2.0;
clipCoord.xy = ndcCoord * clipCoord.w;
```

Algorithm 1: GLSL code fragment for applying the standard model in the vertex shader.

3.1 The Standard Model

The standard model, reduced to the part that is relevant in this discussion, has the following parameters: the camera intrinsic parameters, consisting of the principal point c_x,c_y and the focal lengths f_x,f_y (both in pixel units), the radial distortion parameters k_1,k_2, and the tangential distortion parameters p_1,p_2. The model computes distorted pixel coordinates u,v from undistorted pixel coordinates x,y by first computing normalized image coordinates s,t with distance r to the principal point, applying the distortion, and then reverting the normalization [5]:

$$\begin{aligned} s &= \frac{x-c_x}{f_x} \\ t &= \frac{y-c_y}{f_y} \\ r^2 &= s^2+t^2 \\ d &= 1+k_1r^2+k_2r^4 \\ u &= (sd+(2p_1st+p_2(r^2+2s^2)))f_x+c_x \\ v &= (td+(p_1(r^2+2t^2)+2p_2st))f_y+c_y \end{aligned} \tag{1}$$

Here, the undistorted pixel coordinates x,y are equivalent to pixel coordinates generated with the pinhole camera model of a standard graphics pipeline when the camera intrinsic parameters c_x,c_y,f_x,f_y are accounted for in the projection matrix. This matrix is typically defined by a viewing frustum given by the clipping plane coordinates l,r,b,t for the left, right, bottom, and top plane. These values have to be multiplied by the near plane value n; here we assume $n=1$ for simplicity. Given the image size $w\times h$, suitable clipping plane coordinates can be computed from the camera intrinsic parameters as follows:

$$\begin{aligned} l &= -\frac{c_x+0.5}{f_x} \\ r &= \frac{w}{f_x}+l \\ b &= -\frac{c_y+0.5}{f_y} \\ t &= \frac{h}{f_y}+b \end{aligned}$$

ISBN 978-80-86943-42-8

Using this frustum to define the projection matrix in a standard graphics pipeline accounts for the camera intrinsic parameters of the standard model. The remaining problem is to integrate the lens distortion parameters k_1, k_2, p_1, p_2. This is discussed in the following sections.

3.2 Preprocessing Geometry

In this approach, each input vertex is manipulated such that its image space coordinates match the distorted coordinates of the standard model.

In a standard graphics pipeline, this manipulation is typically done in the vertex shader. Since the standard model operates on pixel coordinates, we first apply the projection matrix from Sec. 3.1 to each vertex, resulting in clip coordinates, and then divide by the homogeneous coordinate to get normalized device coordinates (NDC). By applying the viewport transformation, these are transformed to window coordinates, which are equivalent to pixel coordinates in the standard model. After modifying the x and y components of the window coordinates to account for lens distortion according to Eq. 1, we transform back to clip coordinates. See Alg. 1 for an OpenGL vertex shader code fragment.

This approach has two limitations.

First, modifying clip coordinates in this way means that a fundamental assumption of the graphics pipeline, namely that straight lines in model space map to straight lines in image space, is no longer fulfilled. This leads to errors. A similar problem occurs in graphics applications that project onto non-planar surfaces, e.g. shadow maps [12] and dynamic environment maps [13] that aim to reduce memory usage. There, the errors are considered acceptable if the tessellation of the input geometry is fine enough such that triangle edges in image space are short. Whether this condition is met in our case depends on the application.

Second, our vertex modification takes place before clipping, and therefore includes vertices that lie outside the domain of the standard model. Depending on the distortion parameters, transforming these vertices may place them into image space, resulting in invalid triangles that ruin the rendering result. To avoid this problem, we discard triangles that contain at least one vertex outside of the view frustum. A tolerance parameter δ can be applied during this test to avoid holes in the final image caused by triangles that are partly inside the frustum: a vertex is discarded if its unmodified NDC xy coordinates lie outside $[-1-\delta, 1+\delta]^2$. Since the preprocessing approach requires a finely detailed geometry anyway, simply using $\delta = 0.1$ should work fine. We used this value for all of our tests.

For certain types of distortion, mainly barrel distortion (see Fig. 1), we must additionally account for vertices that lie outside of the pinhole camera frustum but may be mapped into image space nonetheless. This is done by adding a distortion-dependent value D to the parameter δ. Given the inverse of the standard model (see Sec. 3.3 for details), we can determine a lower bound for D automatically by undistorting the distorted image space corner coordinates $(0,0), (w,0), (w,h), (0,h)$, transforming them to NDC coordinates, and setting D to the maximum of the absolute value of each coordinate, minus one.

3.3 Postprocessing Images

In this approach, the scene is first rendered into an undistorted image using an unmodified graphics pipeline based on a pinhole camera with the projection matrix from Sec. 3.1. The result is then transformed into a distorted image by applying the standard model in a postprocessing step, e.g. using a fragment shader.

This postprocessing step requires the computation of undistorted pixel coordinates (x,y) from distorted pixel coordinates (u,v), i.e. the inverse of Eq. 1. This inversion is not a trivial problem; several approaches exist, but none supports the full set of parameters of the original standard model. For example, Drap and Lefèvre propose an exact inversion, but for radial distortion only [14].

We apply ideas by Heikkilä [3] to invert Eq. 1 using an approximation based on Taylor series. Note that his camera model differs from the standard model; in particular, it computes undistorted pixel coordinates from distorted pixel coordinates. Nevertheless, his inversion process is still applicable. The resulting formulas support radial distortion parameters k_1, k_2 and tangential distortion parameters p_1, p_2, which is sufficient in practice:

$$\begin{aligned} s &= \frac{u - c_x}{f_x} \\ t &= \frac{v - c_y}{f_y} \\ r^2 &= s^2 + t^2 \\ d_1 &= k_1 r^2 + k_2 r^4 \\ d_2 &= \frac{1}{4k_1 r^2 + 6k_2 r^4 + 8p_1 t + 8p_2 s + 1} \\ x &= (s - d_2(d_1 s + 2p_1 st + p_2(r^2 + 2s^2)))f_x + c_x \\ y &= (t - d_2(d_1 t + p_1(r^2 + 2t^2) + 2p_2 st))f_y + c_y \end{aligned} \tag{2}$$

Note that the postprocessing step can only fill areas in the distorted image for which information exists in the undistorted image. For certain types of distortion, mainly barrel distortion (see Fig. 1), this means that some areas of the result remain unfilled. This can only be alleviated by using both an enlarged frustum and an increased resolution when rendering the undistorted

 ISBN 978-80-86943-42-8

	Preprocessing geometry	Postprocessing images
Distortion model completeness	full	limited to radial and tangential
Prerequisites	finely detailed geometry	none
Result completeness	full	may have unfilled areas
Rendered data types	all	limited to interpolatable, relocatable data
Complexity	geometry-dependent	resolution-dependent

Table 1: Comparison of the pre- and postprocessing approaches to lens distortion rendering based on the standard model. See Sec. 3.4 for details.

image. Note that while it is possible to derive suitable frustum and resolution parameters by computing undistorted coordinates for the distorted image space corner coordinates $(0,0),(w,0),(w,h),(0,h)$, similar to method described for the preprocessing approach, we did not do so in our tests for simplicity.

3.4 Discussion

In this section, we discuss several aspects of the preprocessing and postprocessing approaches, summarized in Tab. 1.

Distortion model completeness: In the preprocessing approach, we apply the forward standard model and thus can use the full formulas unchanged, i.e. with support for all parameters, including thin prism and tilted sensor distortion if relevant. The postprocessing approach requires the inverse model, and no inversion is known that accounts for all parameters. It is therefore limited to radial and tangential distortion with parameters k_1,k_2,p_1,p_2, but this should be sufficient for the majority of applications.

Prerequisites: Applying the preprocessing approach requires finely tessellated geometry to keep errors small. Not all applications may be able to make such guarantees. The postprocessing approach does not have this limitation.

Result completeness: While the preprocessing approach can map geometry outside of the pinhole camera view frustum into the distorted image, such information is not available to the postprocessing approach unless an enlarged frustum and increased resolution are used for the undistorted image. See Fig. 1.

Rendered data types: The postprocessing approach will usually map undistorted pixels with interpolation to the distorted image. This is fine e.g. for RGB images, but may break for other kinds of data that special applications may render into images, e.g. object IDs or 2D pixel flow. In these cases, only the preprocessing approach can be applied.

Computational complexity: The complexity of the postprocessing approach depends on the number of vertices in the input geometry, while the complexity of the postprocessing approach depends on the number of output pixels. While the preprocessing approach can be integrated directly into any pipeline, the postprocessing approach requires an additional render pass.

4 RESULTS

We implemented both the preprocessing and the postprocessing approach in a standard OpenGL rendering pipeline. To verify that our implementation produces results that match the OpenCV/Matlab implementation of the standard model, we varied the model parameters $c_x,c_y,f_x,f_y,k_1,k_2,p_1,p_2$, then rendered a set of 17 images of size 800×600 for each parameter set, containing the standard OpenCV checkerboard calibration pattern in various 3D positions and orientations, and then used the OpenCV `calibrate.py` script to estimate the model parameters from the rendered images. Note that OpenCV also supports a circle grid calibration pattern, but we chose to use the more widely used checkerboard pattern.

In most cases, the original parameters were recovered with high accuracy, even though the rendered set of images was of low quality for calibration purposes. The average recovery error was less than $1\,\%$ for $c_x,c_y,f_x,f_y,k_2,p_1,p_2$. Interestingly, for k_1 the error was significantly larger, however this did not cause noticeable errors in the undistorted images that were produced for verification purposes. For a few sets, calibration failed, mostly caused by parts of the checkerboard pattern not being visible in some images.

Fig. 2 shows a visual verification: first, an undistorted image is rendered, then a distorted one with a specific set of parameters, and this distorted image is finally undistorted using OpenCV with the same parameters. The first an last image show only minimal differences.

5 CONCLUSION

We presented two methods to accurately render images that match the characteristics of real cameras regarding implicit parameters and lens distortion. Both methods are based on the most widely used camera model in Computer Vision, and can be integrated into any rendering pipeline.

We highlighted the specific advantages and drawbacks of each approach to help implementers pick the right approach for a given application.

 ISBN 978-80-86943-42-8

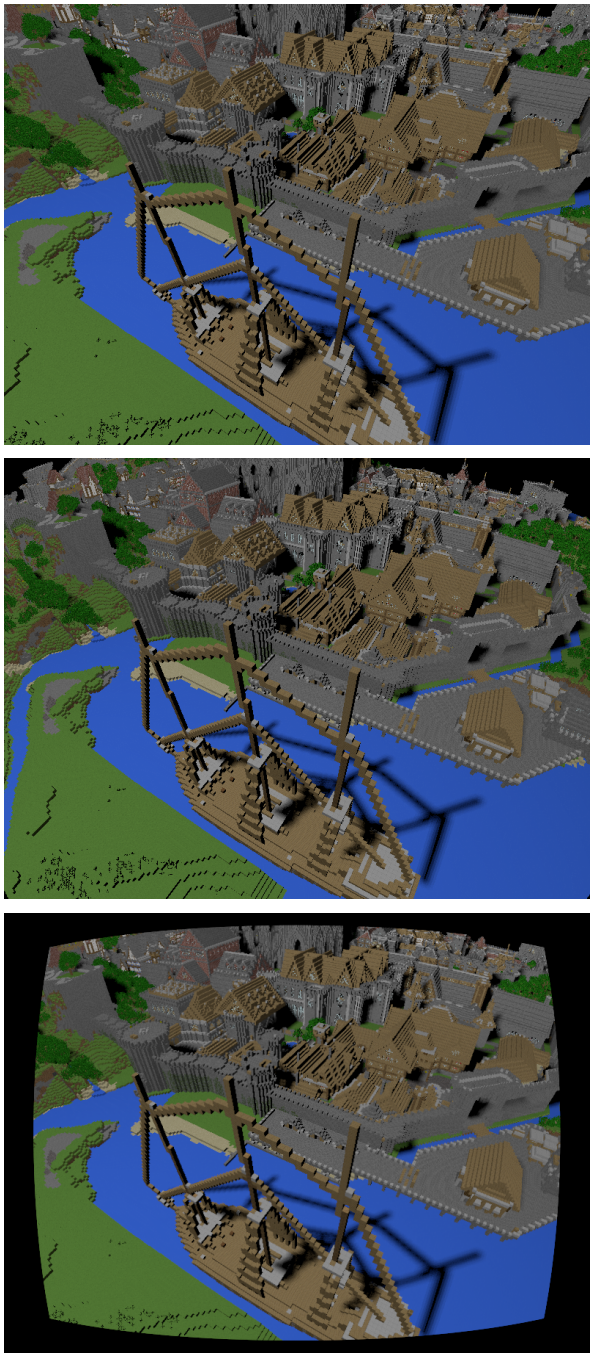

Figure 1: Effects of barrel distortion ($k_1 = -0.11, k_2 = 0, p_1 = 0, p_2 = 0$). From top to bottom: undistorted image, distorted image from preprocessing geometry, and distorted image from postprocessing the undistorted image.

Figure 2: From top to bottom: undistorted image of size $800, 600$ rendered with intrinsic parameters $c_x = 399.5$, $c_y = 299.5$, $f_x = f_y = 400$, distorted image rendered with parameters $k_1 = -0.05$, $k_2 = 0.01$, $p_1 = 0.03$, $p_2 = -0.01$, and undistorted image produced from the distorted image by OpenCV using the same parameters.

ISBN 978-80-86943-42-8

6 ACKNOWLEDGMENTS

The work is partially funded by the German Research Foundation (DFG), grants Ko-2960-12/1 and Ko-2960-13/1.

The scene displayed in Fig. 1 and Fig. 2 is the Rungholt scene from McGuire graphics data [15].

7 REFERENCES

[1] P. Sturm, S. Ramalingam, S. Gasparini, and J. Barreto. *Camera Models and Fundamental Concepts Used in Geometric Computer Vision*. Now Foundations and Trends, 2011.

[2] J. Heikkilä and O. Silvén. A four-step camera calibration procedure with implicit image correction. In *Proc. IEEE Comp. Soc. Conf. Computer Vision and Pattern Recognition*, pages 1106–1112, Jun 1997.

[3] J. Heikkilä. Geometric camera calibration using circular control points. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(10):1066–1077, Oct 2000.

[4] Z. Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, Nov 2000.

[5] OpenCV contributors. OpenCV camera model description. `https://docs.opencv.org/3.4.0/da/d54/group__imgproc__transform.html#ga7dfb72c9cf9780a347fbe3d1c47e5d5a`. Accessed 2018-03-14.

[6] Mathworks. Matlab / Simulink camera parameters. `https://www.mathworks.com/help/vision/ref/cameraparameters.html`. Accessed 2018-03-14.

[7] E. Brachmann, A. Krull, F. Michel, S. Gumhold, J. Shotton, and C. Rother. Learning 6D object pose estimation using 3D object coordinates. In *Proc. European Conf. on Computer Vision (ECCV)*, pages 536–551, 2014.

[8] M. Lambers, S. Hoberg, and A. Kolb. Simulation of time-of-flight sensors for evaluation of chip layout variants. *IEEE Sensors Journal*, 15(7):4019–4026, July 2015.

[9] D. Roble. Vision in film and special effects. *ACM SIGGRAPH Comput. Graph. Newsletter*, 33(4):58–60, November 1999.

[10] C. Kolb, D. Mitchell, and P. Hanrahan. A realistic camera model for computer graphics. In *Proc. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 317–324, 1995.

[11] J. Wu, C. Zheng, X. Hu, and C. Li. An accurate and practical camera lens model for rendering realistic lens effects. In *Int. Conf. on Computer-Aided Design and Computer Graphics*, pages 63–70, September 2011.

[12] D. Scherzer, M. Wimmer, and W. Purgathofer. A survey of real-time hard shadow mapping methods. *Computer Graphics Forum*, 30(1):169–186, 2011.

[13] T. Y. Ho, L. Wan, C. S. Leung, P. M. Lam, and T. T. Wong. Unicube for dynamic environment mapping. *IEEE Trans. Visualization and Computer Graphics (TVCG)*, 17(1):51–63, Jan 2011.

[14] P. Drap and J. Lefèvre. An exact formula for calculating inverse radial lens distortions. *MDPI Sensors*, 16(6), 2016.

[15] Morgan McGuire. Computer graphics archive. `https://casual-effects.com/data`, July 2017.

 ISBN 978-80-86943-42-8