Lens Distortion in 3DE4

Science-D-Visions

August 13, 2018

Contents

0 About this document **3**
0.1 How to avoid reading all this . . . 3
0.2 Versions of this document . . . 3

1 The Lens Distortion Plugin Kit (LDPK) **4**
1.0 Versions of the LDPK . . . 4
1.1 What you need . . . 4
1.2 Installing the LDPK . . . 4
1.3 Content of the LDPK . . . 5
1.4 Scripts . . . 6

2 Math **6**
2.1 Notation . . . 6
2.2 Definitions . . . 7
2.3 Inverting the distortion function . . . 8
2.4 A zoo of lens distortion models . . . 8
2.4.1 Coordinate systems . . . 9
2.4.1.1 Camera coordinates . . . 9
2.4.1.2 Unit coordinates . . . 10
2.4.1.3 Diagonally normalized coordinates . . . 11
2.4.2 Decentering . . . 11
2.4.3 3DE Classic LD Model . . . 11
2.4.4 Anamorphic, Degree 6 . . . 12
2.4.5 Anamorphic - Standard, Degree 4 . . . 12
2.4.6 Radial - Fisheye, Degree 8 . . . 13
2.4.7 Radial - Standard, Degree 4 . . . 13

3 Implementation **14**
3.1 General remarks . . . 14
3.2 The Lens Distortion Plugin Concept . . . 14
3.2.1 The API . . . 15
3.2.1.1 Model and parameter identifiers . . . 15
3.2.1.2 Parameter types and values . . . 15
3.2.1.3 Modifying parameter values . . . 16
3.2.1.4 Preparing the model . . . 16
3.2.1.5 JACOBI-Matrix . . . 17
3.2.1.6 Remove and apply lens distortion . . . 17
3.3 Building a compositing node . . . 17
3.3.1 Classes . . . 17
3.3.2 Unit coordinates and pixel coordinates . . . 18
3.3.3 Removing and applying lens distortion . . . 19

3.3.4 Resize and reapply 19
3.3.4.1 Resize 20
3.3.4.2 Reapply 20

A Table of symbols **21**

0 About this document

0.1 How to avoid reading all this

The document contains a lot of details which you do not need for developing at least a plugin with basic functionality, which means, 3DE4's core will be able to use it for calculations. In principle, all you need is the base class `tde4_ld_plugin` which is located in

`$LDPK/include/ldpk/tde4_ld_plugin.h`

The header file is quite self-explanatory. You derive your own 3DE4-plugin class from this base class. Your derived class then contains:

- Handling for the seven built-in parameters
- Handling for your model-dependent parameters
- A method `undistort()`
- A method `distort()`

Apart from that you need C-style create- and destroy-functions in the global namespace. Please copy them from

`$LDPK/source/ldpk/tde4_ldp_example_radial_deg_8.C`

and insert the name of your plugin class in the create-function. Also, please keep in mind:

- Your plugin must be thread-safe. In 3DE4 and some compositing systems, many instances of your plugin class will be created, one per thread. One instance is used by one thread only.
- Get your coordinate systems right and take into account lens center offset properly.
- 3DE4 will invoke `undistort()` pretty often, so this method should be fast.

After compiling, please move your shared object to

`$TDE/user_data/ld_plugins/`

and start 3DE4. Your plugin should appear in 3DE4's *Attribute Editor*. In order to develop a **compositing node**, please follow the instructions of the compositing system's development kit, create a node class and either implement the mathematics from scratch or use the plugin class you wrote a minute ago. Also, keep in mind:

- An image is undistorted using the method `distort()`!
- An image is distorted using the method `undistort()`!

0.2 Versions of this document

If you have questions about this document, the math, the implementation or the software licence, please contact uwe@sci-d-vis.com. Bug reports are welcome!

Version 1.5	2013——	In preparation
Version 1.4	2013-04-25	Removed obsolete section about derivatives
Version 1.3	2012-06-29	None
Version 1.2	2012-02-02	Added parameter `tde4_custom_focus_distance_cm`
Version 1.1	2011-09-05	Minor corrections
		JACOBI-matrix of the classic model
Version 1.0	2011-02-16	LDPK; Math; Implementation

1 The Lens Distortion Plugin Kit (LDPK)

The LDPK is meant to make it easier for you to develop a lens distortion plugin, either for 3DE4, or for your favourite compositing system. We tried to make it as little intrusive as possible. For 3DE4 there is only a single base class you have to use.

1.0 Versions of the LDPK

Version 1.7	not yet	Bounding Box methods in `$LDPKldp_builtin`
		New Nuke plugins (easier to install and to use)
Version 1.6	2013-11-12	Python bindings for the built-in models
		Bugfix for string-valued parameters
		Pre-compiled libraries, modules, plugins
Version 1.5	2013-09-16	Bugfixes, check for undefined built-in parameters
Version 1.4	2013-04-25	New model Anamorphic-Standard-Degree-4 and modified
		Radial-Standard-Degree-4 (beam-splitter compensation)
Version 1.3	2012-06-29	The Radial-Degree-8-Model compensates for equisolid-angle
		fisheye distortion
Version 1.2	2012-02-02	Added parameter `tde4_custom_focus_distance_cm`
Version 1.1	2011-04-16	Minor corrections in Doxygen doc
		Simplified `$LDPK/classic_3de_mixed_distortion`
		Base class for built-in plugin classes
		Thread-safety of the `distort`-method (lookup-tables)
		Helper classes now form a library `libldpk.a`
Version 1.0	2011-02-16	

1.1 What you need

In order to use the LDPK you need the following:

- One of the following operating systems:
  - Mac OSX 10.x 64bit
  - Linux 64bit
- `g++`, the gnu c++ compiler, version 4.x.x (must have)
- a browser for reading the class documentation (should have)
- `/bin/csh` called "C shell" (should have)
- `gnuplot` for visualizing (nice to have)

1.2 Installing the LDPK

The LDPK is delivered as a tar-archive. In order to install it please go to the directory, where you would like to place the LDPK, copy the tar-archive there and invoke

```
tar xfz ldpk-xxx.tgz
```

on the command line. LDPK will unpack into a directory named `ldpk-xxx`, where `xxx` stands for some version number. In this document, we will refer to the base directory of the unfolded tar-archive as `$LDPK`.

1.3 Content of the LDPK

The LDPK contains the following directories:

- `bin` - precompiled binaries for testing the plugin on various platforms. Compiled test programs will be placed here. Once you have implemented and compiled your plugin you can quickly check it by running one of the following programs:
  - `tde4_plugin_info.linux`
  - `tde4_plugin_info.osx`

  After running the compile script, you will find some more tools here:
  - `test_plugin_loader` - Another simple test program for loading a plugin. You simply run the program and pass the abolute or relative path to your plugin shared object. The program will write some information about the plugin to `stdout` and exit.
  - `test_model_visualizer` - A program which generates gnuplot data for plotting a vector field from a parameter specification file. We will have a look at this tool later, when we discuss how to implement a plugin for 3DE4.
- `doc` - Doxygen and pdf documents, including this one. The entry point for the Doxygen documents is `file:///$LDPK/doc/doxy/html/index.html`. The pdf-document you are currently reading is located in `$LDPK/doc/tex/`.
- `include` - header files. The best way to explore these is to read the Doxygen documentation, starting at `file:///$LDPK/doc/doxy/html/index.html` If you plan to implement a compositing node for 3DE4's built-in distortion models, please have a look at these classes. The distortion classes contain reference implementations for all built-in models of 3DE4.
  - `ldpk::radial_decentered_distortion`
  - `ldpk::generic_radial_distortion`
  - `ldpk::generic_anamorphic_distortion`
  - `ldpk::classic_3de_mixed_distortion`
  - `tde4_ldp_radial_decentered_deg_4`
  - `tde4_ldp_radial_deg_8`
  - `tde4_ldp_anamorphic_deg_6`
  - `tde4_ldp_classic_3de_mixed`
- `lib` - a place for libraries. When you run the compile script, any libraries to be generated are placed here.
- `script` - scripts for compiling and cleaning up. The scripts are:
  - `makeall.csh` - A script for compiling examples and test programs.
  - `cleanup.csh` - Remove all files created by `makeall.csh`
- `source` - source code for classes
- `test` - source code for test programs, see `bin/`.

1.4 Scripts

All test programs and example plugins are compiled by running `$LDPK/script/makeall.csh`. The script `$LDPK/script/cleanup.csh` is used to reset the directory content of `$LDPK` to its original state. The script `$LDPK/script/makedoc.csh` (which you probably don't need) is used for creating the Doxygen documents. When you have compiled everything you can do a first test by the following commands. This program generates table data for gnuplot by evaluating the plugin passed and using the model parameters given by some parameter file:

```
unix> cd $LDPK
unix> bin/test_model_visualizer lib/tde4_ldp_example_radial_deg_8.so\
        test/para_example_radial_deg_8.data /tmp/outgp.data
unix> gnuplot
gnuplot> plot '/tmp/outgp.data' with vector
```

The result is a vector field representation of a radial distortion model.

2 Math

In the following, we will specify in mathematical notation what we understand by a distortion model in the context of 3DE4. By intuition, you will already know most of what we present here, if you want to implement a lens distortion plugin or compositing node. Concerning plugins for 3DE4, it is important however to get a concept of *linear distortion models*, which we explain in this section.

2.1 Notation

Given two functions $f : X \to X'$ and $g : Y \to Y'$ where $f(X) \subset Y$ we compose them by writing

$$g \circ f \tag{1}$$

which maps x to $g(f(x))$. We denote the n-dimensional space by $\mathbb{R}^n$. The components of a tuple $p \in \mathbb{R}^n$ are p_i where $i \in \{0, \ldots, n-1\}$. Generally, we use latin indices in order to describe points in parameter spaces.
Images are defined on a two-dimensional subspace of $\mathbb{R}^2$. Points and vectors in images space are tuples (x, y). We describe them using an index notation as follows:

- For any point p in $\mathbb{R}^2$ we refer to the components of p by writing p_μ, where $\mu = 0$ and $\mu = 1$ address the x- and y-component of p, respectively.

- A function g mapping to $\mathbb{R}^2$ is decomposed in x- and y-component in the same way: $g_\mu(\ldots)$ or $g(\ldots)_\mu$.

- The derivative of a function g mapping from $\mathbb{R}^2$ is written as

$$\frac{\partial}{\partial p_\mu} g(p) \tag{2}$$

We use greek indices $\mu, \nu \ldots$ for the special case of components of tuples in an image space. The EUCLIDIAN norm of a tuple p in n dimensions is denoted by

$$|p| = \sqrt{\sum_{i=0}^{n-1} p_i^2} \tag{3}$$

2.2 Definitions

We do not specify explicitly the continuity class of functions in order to not overburden this section. Usually distortion functions are quite reasonable as far as continuity and differentiability are concerned. For completeness, let us assume that all functions are at least two times continuously differentiable.

Definition 2.2.1. Let P and Q be (reasonable) subsets of $\mathbb{R}^2$, i.e. they are connected submanifolds of $\mathbb{R}^2$. Let C be a connected submanifold of $\mathbb{R}^n$ for some number n. We call C the *parameter space* and the elements of C *parameter sets*. We consider a smooth mapping

$$g : P \times C \to Q \tag{4}$$

For any given parameter set $c \in C$ we define the function g_c by

$$g_c : P \to Q : p \mapsto g(p, c) \tag{5}$$

We call g a *distortion model* with n parameters, if it has the following properties:

1. **Fixed point** - There is a point $p_0 = (x_0, y_0) \in P$ so that for all $c \in C$

   $$g(p_0, c) = p_0 \tag{6}$$

   i.e. for any $c \in C$, p_0 is a fixed point of g_c. We call p_0 the *lens center*.

2. **Default parameters** - There is a parameter set $c_0 \in C$, so that for all $p \in P$

   $$g(p, c_0) = p \tag{7}$$

   i.e. g_{c_0} is the identity map on P.

3. **Invertibility** - g_c is invertible, i.e. there is a mapping

   $$g_c^{-1} : Q \to P \tag{8}$$

   so that

   $$g_c^{-1} \circ g_c = \mathrm{id}|_P \tag{9}$$

   This tells us more about P and Q than the distortion model itself. We simply demand, that the distortion model has an inverse, when we need it.

Remark. For any lens distortion model we can define a parameter set c_0 as *default values*, if g_c is the identity map, which is important for 3DE4's plugin concept.

Definition 2.2.2. Let $g : P \times C \to Q$ be a lens distortion model with fixed point p_{fix}. Let T_p be the translation operator $T_p : \mathbb{R}^2 \to \mathbb{R}^2$ which maps x to $x + p$. We define a shifted distortion model $\hat{g}$ by:

$$\hat{g}_c = T_{p_{\text{fix}}}^{-1} \circ g_c \circ T_{p_{\text{fix}}} \tag{10}$$

where $\hat{g}_c$ maps from $P - x$ to $Q - x$. The following proposition tells us, that we can simply consider lens distortion models around $(0, 0)$, as long as we define them as in this section. This simplifies our notation and allows us to separate mathematics from 3DE4-things when we implement a plugin.

Proposition 2.2.1. *$\hat{g}$ is a distortion model.*

Proof. This is easy to see. We check the four items from our definition.

1. The fixed point of $\hat{g}_c$ is $(0, 0)$.
2. If c_0 is the default parameter set, then

$$\hat{g}_{c_0} = T_{p_{\text{fix}}}^{-1} \circ g_{c_0} \circ T_{p_{\text{fix}}} = T_{p_{\text{fix}}}^{-1} \circ id|_{\mathbb{R}^2} \circ T_{p_{\text{fix}}} = T_{p_{\text{fix}}}^{-1} \circ T_{p_{\text{fix}}} = id|_{\mathbb{R}^2} \tag{11}$$

3. Since $g_c : P \to Q$ and $T_{p_{\text{fix}}}$ are invertible, $\hat{g}_c : P - x \to Q - x$ is invertible as well.

$\square$

2.3 Inverting the distortion function

For a given distortion function $g_c : P \to Q$ we have to find a way of computing the inverse. Let us assume g_c is a distortion function for *removing* lens distortion, then we also need (e.g. for image processing) the inverse distortion function. For any given point $q \in Q$ we wish to find the point $p \in P$ so that $g_c(p) = q$. This can be done as follows using NEWTON's method. First, we define a function

$$F(p) = g_c(p) - q \tag{12}$$

Finding the inverse of q is equivalent to find the zero point of F. Starting at some initial value $p^{(0)}$ we iterate

$$p^{(k+1)} = p^{(k)} - J^{-1}(p)F(p^{(k)}) \tag{13}$$

where

$$J_{\mu\nu}(p) = \frac{\partial}{\partial p_\nu} F_\mu(p) = \frac{\partial}{\partial p_\nu} g_c(p)_\mu \tag{14}$$

is the JACOBI-matrix of F at p, until

$$\left| p^{(k+1)} - p^{(k)} \right| < \epsilon \tag{15}$$

for some[1] pre-defined ϵ. The algorithm will converge if $p^{(0)}$ is close to $g_c^{-1}(q)$. A good choice for the initial value, if no other information is available, is

$$p^{(0)} = q - (g_c(q) - q) = 2q - g_c(q) \tag{16}$$

The idea is the following: We assume that $g_c(p)$ depends smoothly on p and varies slowly. That means $g_c(q) - q$ is not so far away from $g_c^{-1}(g_c(q)) - g_c^{-1}(q)$ which is $q - p$. Then p hopefully is not so far from $2q - g_c(q)$. Results are not bad for this initial value. Yet, even with this choice of initial values, there are situations in practice, where NEWTON's method does not converge. For this reason it makes sense to generate lookup tables. This is explained in section 3.

2.4 A zoo of lens distortion models

In this section, we present the math of 3DE4's built-in distortion models. In the following subsections, we will call the undistorted point

$$q = (x', y'), \tag{17}$$

while the distorted point is

$$p = (x, y). \tag{18}$$

[1] We do not work out the details here. Just some sort of termination criterion. . .

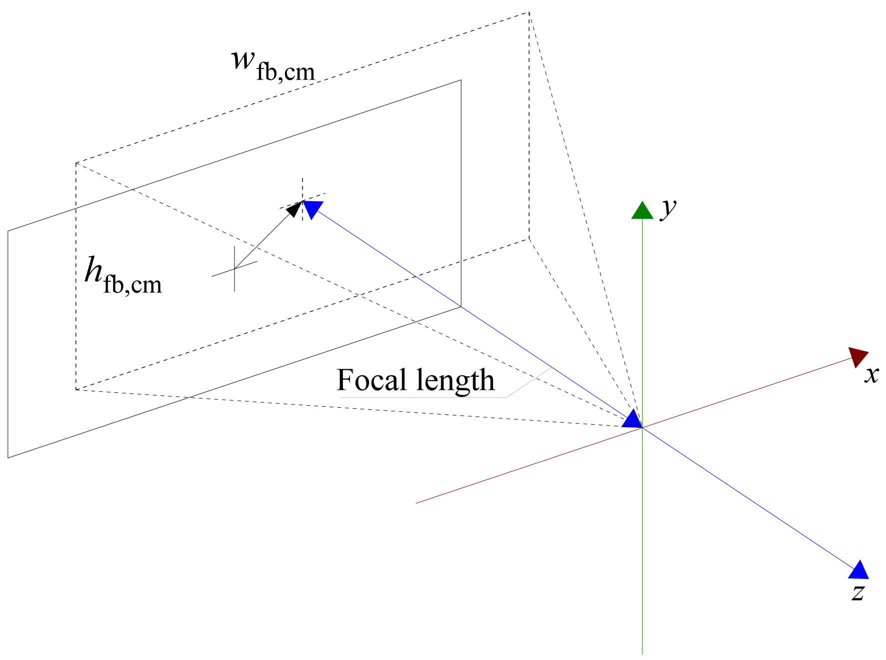

Figure 1: Camera pyramid and lens center

All models express by which prescription the undistorted point is computed from the distorted point. In all models we have chosen the coordinates in a way, so that the lens center is $(0, 0)$ and that the diameter of the image is 2 (i.e. diagonally normalized coordinates). Some of the models are represented in polar coordinates:

$$\begin{aligned} r &= |p| = \sqrt{x^2 + y^2} \\ \phi &= \arctan(y, x) \end{aligned} \tag{19}$$

where we use $\arctan(\cdot, \cdot)$ as math symbol for `double atan2(double y,double x)` (see manpage `man atan2`). For most of the built-in distortion models we show the Jacobi-matrix. The built-in models use this in order to compute the inverse of the distortion function (see section 2.3 for details). The advantage of implementing the Jacobi-matrix instead of relying on difference quotients is a higher performance by a factor 2 to maybe 3.

2.4.1 Coordinate systems

We will have to deal with several coordinate systems, which we describe in the following. Real measure coordinates are helpful because in 3DE4 the camera is specified by means of real measure quantities, including filmback width, filmback height, lens center offset and focal length. Unit coordinates are helpful, because they represent a resolution independent, camera-scaling invariant way to specify the lens distortion plugin API. Tracking data in 3DE4 are stored in unit coordinates. For our lens distortion models we need isometric, unit-free coordinates, which are called diagonally normalized coordinates. In this section we describe, how these systems are related to each other.

2.4.1.1 Camera coordinates Camera coordinates are real measure coordinates on the projection plane of a camera placed in the origin of three-dimensional space. The projection plane is located at $z = -f_{\text{cm}}$, where f_{cm} is the focal length in cm (see fig. 1). The lens center in these coordinates is $(0, 0)$, lower left and upper right corners are

$$(-\frac{w_{\mathrm{fb,cm}}}{2}, -\frac{h_{\mathrm{fb,cm}}}{2}) \tag{20}$$

and

$$(+\frac{w_{\mathrm{fb,cm}}}{2}, +\frac{h_{\mathrm{fb,cm}}}{2}), \tag{21}$$

respectively.

2.4.1.2 Unit coordinates Unit coordinates are important because we use them for defining the API of our lens distortion plugins. They are defined as shown in fig. 2. We define the domain of the image as $I \times I$ with $I = [0, 1]$, where $(0, 0)$ is the lower left corner and $(1, 1)$ is the upper right corner. The center of the image is $(\frac{1}{2}, \frac{1}{2})$. Unit coordinates are related to camera coordinates by the mapping $\psi_{\mathrm{unit}\leftarrow\mathrm{cm}}$:

$$\begin{aligned} x_{\mathrm{unit}} &= \frac{x_{\mathrm{cm}}}{w_{\mathrm{fb,cm}}} + \frac{x_{\mathrm{lco,cm}}}{w_{\mathrm{fb,cm}}} + \frac{1}{2} \\ y_{\mathrm{unit}} &= \frac{y_{\mathrm{cm}}}{h_{\mathrm{fb,cm}}} + \frac{y_{\mathrm{lco,cm}}}{h_{\mathrm{fb,cm}}} + \frac{1}{2}. \end{aligned} \tag{22}$$

The inverse mapping $\psi_{\mathrm{cm}\leftarrow\mathrm{unit}}$ is

$$\begin{aligned} x_{\mathrm{cm}} &= \left(x_{\mathrm{unit}} - \frac{1}{2}\right) w_{\mathrm{fb,cm}} - x_{\mathrm{lco,cm}} \\ y_{\mathrm{cm}} &= \left(y_{\mathrm{unit}} - \frac{1}{2}\right) h_{\mathrm{fb,cm}} - y_{\mathrm{lco,cm}}. \end{aligned} \tag{23}$$

Fig. 2 illustrates the relationship between the two coordinate systems.

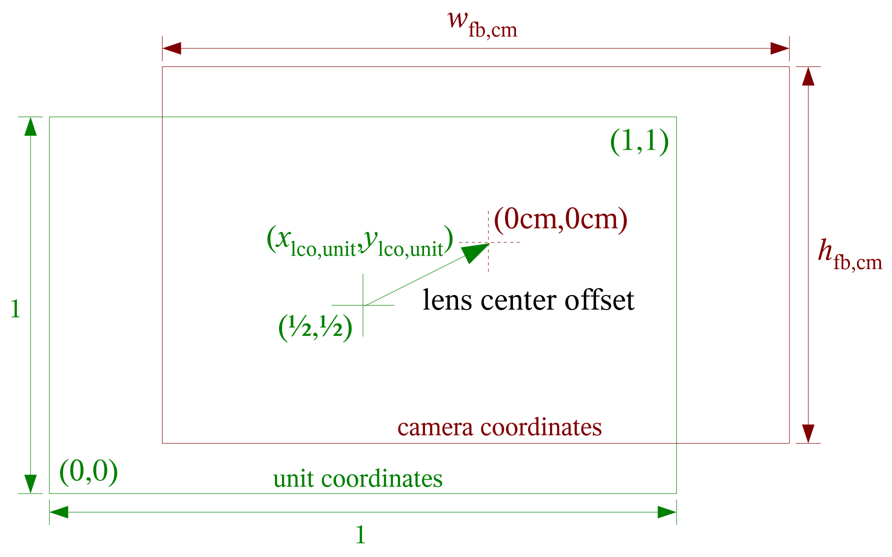

Figure 2: Camera coordinates and unit coordinates

2.4.1.3 Diagonally normalized coordinates

Unit coordinates have the draw back, that they are not isometric: a length value along x does not represent the same length in real world measures as it does along y. Therefore, all our built-in lens distortion models **are defined in diagonally normalized coordinates**. In these units, the image has a diagonal diameter of 2. We use the following symbols to convert to and from this system:

$$\psi_{\text{unit}\leftarrow\text{dn}}: \begin{array}{c} x_{\text{dn}} \\ y_{\text{dn}} \end{array} \mapsto \begin{array}{rcl} x_{\text{unit}} &=& x_{\text{dn}}\frac{r_{\text{fb,cm}}}{w_{\text{fb,cm}}} + \frac{x_{\text{lco,cm}}}{w_{\text{fb,cm}}} + \frac{1}{2} \\ y_{\text{unit}} &=& y_{\text{dn}}\frac{r_{\text{fb,cm}}}{h_{\text{fb,cm}}} + \frac{y_{\text{lco,cm}}}{h_{\text{fb,cm}}} + \frac{1}{2} \end{array} \tag{24}$$

$$\psi_{\text{dn}\leftarrow\text{unit}}: \begin{array}{c} x_{\text{unit}} \\ y_{\text{unit}} \end{array} \mapsto \begin{array}{rcl} x_{\text{dn}} &=& \left(x_{\text{unit}} - \frac{1}{2}\right)\frac{w_{\text{fb,cm}}}{r_{\text{fb,cm}}} - \frac{x_{\text{lco,cm}}}{r_{\text{fb,cm}}} \\ y_{\text{dn}} &=& \left(y_{\text{unit}} - \frac{1}{2}\right)\frac{h_{\text{fb,cm}}}{r_{\text{fb,cm}}} - \frac{y_{\text{lco,cm}}}{r_{\text{fb,cm}}} \end{array}$$

$$\psi_{\text{cm}\leftarrow\text{dn}}: \begin{array}{c} x_{\text{dn}} \\ y_{\text{dn}} \end{array} \mapsto \begin{array}{rcl} x_{\text{cm}} &=& x_{\text{dn}} r_{\text{fb,cm}} \\ y_{\text{cm}} &=& y_{\text{dn}} r_{\text{fb,cm}} \end{array} \tag{25}$$

$$\psi_{\text{dn}\leftarrow\text{cm}}: \begin{array}{c} x_{\text{cm}} \\ y_{\text{cm}} \end{array} \mapsto \begin{array}{rcl} x_{\text{dn}} &=& \frac{x_{\text{cm}}}{r_{\text{fb,cm}}} \\ y_{\text{dn}} &=& \frac{y_{\text{cm}}}{r_{\text{fb,cm}}} \end{array} \tag{26}$$

where we have defined the filmback radius by

$$r_{\text{fb,cm}} = \frac{1}{2}\sqrt{w_{\text{fb,cm}}^2 + h_{\text{fb,cm}}^2} \tag{27}$$

2.4.2 Decentering

All lens systems suffer from certain mechanical inaccuracies. One of these inaccuracies is referred to in the literature as *decentering*. It means, that not all lenses in the lens system are precisely centered on the optical axis. Up to a certain degree, all built-in models we present in the following sections are able to account for decentering, since lens center offset can be optimized for all models. However, only one model is able to separate decentering effects from lens center offset, namely *Radial - Standard, Degree 4*. This model is already used in production and we recommend it for non-anamorphic lenses.

2.4.3 3DE Classic LD Model

This is the model which has been used in 3DE4 before the plugin concept was realized. It combines degree-2 anamorphic terms and degree-4 radial terms. As already mentioned, we use diagonally normalized coordinates in our lens distortion models. We denote coefficients for the x-component and y-component by $c^x_{\dots}$ and $c^y_{\dots}$, respectively.

$$\begin{aligned} x' &= x(1 + c^x_x x^2 + c^x_y y^2 + c^x_{xx} x^4 + c^x_{xy} x^2 y^2 + c^x_{yy} y^4) \\ y' &= y(1 + c^y_x x^2 + c^y_y y^2 + c^y_{xx} x^4 + c^y_{xy} x^2 y^2 + c^y_{yy} y^4) \end{aligned} \tag{28}$$

where the coefficients are represented by five parameters $\delta, \epsilon, \eta_x, \eta_y, q$:

$$\begin{array}{lllll} c^x_x = \frac{\delta}{\epsilon} & c^x_y = \frac{\delta + \eta_x}{\epsilon} & c^x_{xx} = \frac{q}{\epsilon} & c^x_{xy} = 2\frac{q}{\epsilon} & c^x_{yy} = \frac{q}{\epsilon} \\ c^y_x = \delta + \eta_y & c^y_y = \delta & c^y_{xx} = q & c^y_{xy} = 2q & c^y_{yy} = q \end{array} \tag{29}$$

The names of these parameters in 3DE4 are:

δ	*Distortion*
ϵ	*Anamorphic Squeeze*
η_x	*Curvature X*
η_y	*Curvature Y*
q	*Quartic Distortion*

The reference implementation can be found in

`$LDPK/include/ldpk/ldpk_classic_3de_mixed_distortion.h`

The Jacobi-matrix is

$$\begin{aligned}
J_{00} &= 1 + 3c^x_x x^2 + c^x_y y^2 + 5c^x_{xx} x^4 + 3c^x_{xy} x^2 y^2 + c^x_{yy} y^4 \\
J_{01} &= 2c^x_y xy + 2c^x_{xy} x^3 y + 4c^x_{yy} xy^3 \\
J_{10} &= 2c^y_x xy + 4c^y_{xx} x^3 y + 2c^y_{xy} xy^3 \\
J_{11} &= 1 + c^y_x x^2 + 3c^y_y y^2 + c^y_{xx} x^4 + 3c^y_{xy} x^2 y^2 + 5c^y_{yy} y^4. \qquad (30)
\end{aligned}$$

Although this model is widely spread, we recommend to select either *Anamorphic - Standard, Degree 4* or *Radial - Standard, Degree 4*.

2.4.4 Anamorphic, Degree 6

The anamorphic model uses a lot of parameters, probably more than would be necessary, to model anamorphic lenses when no decentering is involved. We split $g(p, c)$ into x- and y-component:

$$\begin{aligned}
x' = x(1 \quad & + c^x_{02} r^2 && + c^x_{04} r^4 && + c^x_{06} r^6 \\
& + c^x_{22} r^2 \cos 2\phi && + c^x_{24} r^4 \cos 2\phi && + c^x_{26} r^6 \cos 2\phi \\
& && + c^x_{44} r^4 \cos 4\phi && + c^x_{46} r^6 \cos 4\phi \\
& && && + c^x_{66} r^6 \cos 6\phi) \\
y' = y(1 \quad & + c^y_{02} r^2 && + c^y_{04} r^4 && + c^y_{06} r^6 \\
& + c^y_{22} r^2 \cos 2\phi && + c^y_{24} r^4 \cos 2\phi && + c^y_{26} r^6 \cos 2\phi \\
& && + c^y_{44} r^4 \cos 4\phi && + c^y_{46} r^6 \cos 4\phi \\
& && && + c^y_{66} r^6 \cos 6\phi) \qquad (31)
\end{aligned}$$

An implementation of this model, which you can use for compositing plugins is in

`$LDPK/include/ldpk/ldpk_generic_anamorphic_distortion.h`

2.4.5 Anamorphic - Standard, Degree 4

The standard anamorphic model is a degree-4 anamorphic model with additional parameters which allow modelling a slightly rotated anamorpic lens and scaling in two directions. We shall discuss the details in a later version of this document. The pure anamorphic part reads

$$\begin{aligned}
x' = x(1 \quad & + c^x_{02} r^2 && + c^x_{04} r^4 \\
& + c^x_{22} r^2 \cos 2\phi && + c^x_{24} r^4 \cos 2\phi \\
& && + c^x_{44} r^4 \cos 4\phi) \\
y' = y(1 \quad & + c^y_{02} r^2 && + c^y_{04} r^4 \\
& + c^y_{22} r^2 \cos 2\phi && + c^y_{24} r^4 \cos 2\phi \\
& && + c^y_{44} r^4 \cos 4\phi) \qquad (32)
\end{aligned}$$

2.4.6 Radial - Fisheye, Degree 8

The current model for compensating fisheye distortion uses an equisolid-angle mapping function and an even-degree 8 polynomial. We'll describe the details in a later version of this document.

2.4.7 Radial - Standard, Degree 4

This model is a slight modification of the famous distortion model by Brown[1966] and Conradi[1919]. It is a model for radially symmetric lenses which accounts for slight decentering of lenses. In coordinates around the lens center, the model up to and including including order five[2] reads

$$\begin{aligned} x' &= x(1 + c_2 r^2 + c_4 r^4) + \left[t_1(r^2 + 2x^2) + 2t_2 xy\right]\ (1 + t_3 r^2) \\ y' &= y(1 + c_2 r^2 + c_4 r^4) + \left[t_2(r^2 + 2x^2) + 2t_1 xy\right]\ (1 + t_3 r^2). \end{aligned} \tag{33}$$

As you see, this model is not linear in its coefficients, since t_3 appears as a product with t_1 and t_2. We have modified this model by introducing an additional parameters in the following way. We define

$$\begin{aligned} u_1 &= t_1 & u_3 &= t_3 t_1 \\ v_1 &= t_2 & v_3 &= t_3 t_2. \end{aligned} \tag{34}$$

Rewriting the original model by means of these four parameters leads to the linear form (i.e. no products of coefficients)

$$\begin{aligned} x' &= x(1 + c_2 r^2 + c_4 r^4) + (r^2 + 2x^2)(u_1 + u_3 r^2) + 2xy(v_1 + v_3 r^2) \\ y' &= y(1 + c_2 r^2 + c_4 r^4) + (r^2 + 2y^2)(v_1 + v_3 r^2) + 2xy(u_1 + u_3 r^2). \end{aligned} \tag{35}$$

We have added two more parameters for compensating artefacts resulting from the beam-splitter used in certain stereo rigs. We shall describe them in a later version of this document.
In practice this means, we add one degree of freedom to the system, but in turn we get a faster and more robust method for computing the coefficients. An implementation of this model, which you can use for compositing plugins is in

```
$LDPK/include/ldpk/ldpk_radial_decentered_distortion.h}
```

The Jacobi-matrix $J_{\mu\nu}$ (without compensating for beam-splitter) is:

$$\begin{aligned} J_{00} &= 1 + c_2(y^2 + 3x^2) + c_4(y^2 + 5x^2)r^2 \\ &\quad + 6u_1 x + u_3(8xy^2 + 12x^3) + 2v_1 y + v_3(2y^3 + 6x^2 y) \\ J_{01} &= 2c_2 xy + 4c_4 xyr^2 \\ &\quad + 2u_1 y + u_3(8x^2 y + 4y^3) + 2v_1 x + v_3(2x^3 + 6xy^2) \\ J_{10} &= 2c_2 xy + 4c_4 xyr^2 \\ &\quad + 2u_1 y + u_3(6x^2 y + 2y^3) + 2v_1 x + v_3(4x^3 + 8xy^2) \\ J_{11} &= 1 + c_2(x^2 + 3y^2) + c_4(x^2 + 5y^2)r^2 \\ &\quad + 6v_1 y + v_3(8x^2 y + 12y^3) + 2u_1 x + u_3(2x^3 + 6xy^2). \end{aligned} \tag{36}$$

[2]Our nomenclature "Degree 4" refers to the power of r at c_4

3 Implementation

3.1 General remarks

On a low technical level, removing lens distortion is done in two ways for two different purposes:

- We would like to remove distortion for a given set of point positions. These point positions can be e.g. feature points from tracking or image analysis.
- On the other hand we would like to remove distortion from pixel-based image material. In this case we already *know* the target positions, i.e. the position of all pixels, but would like to know where a particular pixel originates from.

Let us consider a distortion function g, which maps a distorted point p to an undistorted point q. It's inverse mapping is denoted by g^{-1} and maps q back to p. The two situations are shown in fig. 3.

In this document we will *always* call g the **distortion function** and g^{-1} the **inverse distortion function**.

For all distortion models, we postulate that the distortion function g shall be calculated without recurse to any kind of initial value (i.e. non-iteratively), while the inverse distortion function may be implemented as iterative function and require initial values. This way of formulating the distortion function and its inverse is widely spread in the literature, and it makes sense for the following reason: In 3DE4 we need fast, precise and robust access to undistorted tracking data without initial values. For compositing nodes used for image processing, however, complete images or large parts of an image have to be undistorted. This justifies the use of a lookup table in order to calculate the inverse distortion function, and this makes it easy to calculate functions based on initial values like g^{-1}.

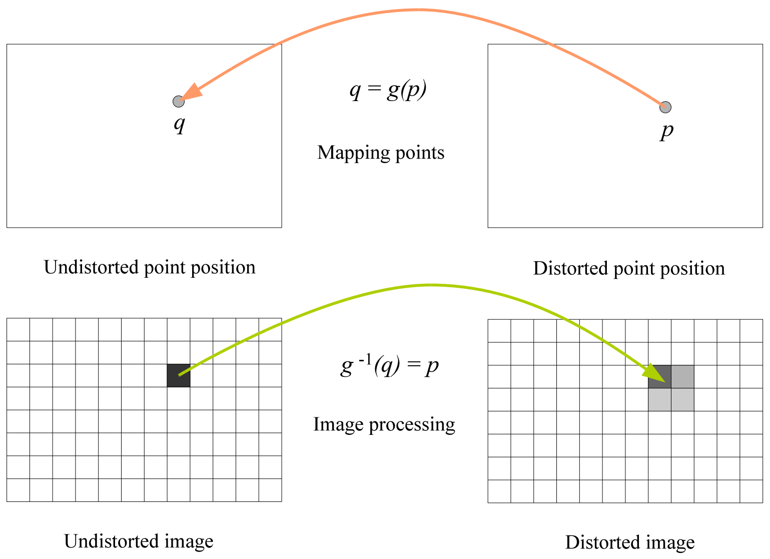

Figure 3: Mapping and inverse mapping

3.2 The Lens Distortion Plugin Concept

3DE4's lens distortion plugin concept is built upon an abstract class `tde4_ld_plugin`, from which the developer of a plugin derives their own class. The resulting class is compiled as shared object library and placed at `/user_data/ld_plugins/` in 3DE4's installation directory.

3.2.1 The API

In the following, we will have a closer look at the plugin API. The header file is called `tde4_ld_plugin.h`. The class `tde4_ld_plugin` is an abstract class and therefore starts like this:

```
class tde4_ld_plugin
       {
public:
       virtual ~tde4_ld_plugin() {}
       ...
       };
```

3.2.1.1 Model and parameter identifiers Each distortion model has a unique name, and it will be identified by this name. The length of this name is restricted to 100 characters. The derived class provides this name by implementing the method `getModelName()`. By default, each method has a boolean return value in order to indicate an error while the method is called. A return value of `true` indicates, that no error has occured.

```
       ...
virtual bool getModelName( char *model) = 0;

virtual bool getNumParameters( int &n) = 0;

virtual bool getParameterName( int i,
                               char *identifier) = 0;

       ...
```

Each model will have a number of parameters, which characterize the distortion function. The number of parameters is obtained by calling `getNumParameters()`. Each parameter has a type and an identifier. Parameters are addressed by means of their identifier in every method of the plugin class. The method `getParameterName()` is used to get this identifier. Its length is restricted to 100 characters.

3.2.1.2 Parameter types and values Once we have the identifier of a parameter we can obtain its properties and control it. `getParameterType()` delivers the type. The types are given by the following `enum`-declaration in `tde4_ld_plugin.h`:

```
// parameter types...
enum tde4_ldp_ptype {
       TDE4_LDP_STRING, TDE4_LDP_DOUBLE, TDE4_LDP_INT,
       TDE4_LDP_FILE, TDE4_LDP_TOGGLE, TDE4_LDP_ADJUSTABLE_DOUBLE };

...
// returns type of given parameter...
virtual bool getParameterType( const char *identifier,
                                      tde4_ldp_ptype &type) = 0;
...
```

In 3DE4's user interface, the type of a parameter determines, how it is represented; `string` parameters are represented by a single line text field, `file` parameters have a button in order to open a file browser. `double` and `adjustable double` parameters are represented as a floating point number. An adjustable parameter can be calculated in 3DE4's *Matrix Tool* or in the *Parameter Adjustment Window*. The next four methods are used for obtaining the default value for each parameter.

```
virtual bool getParameterDefaultValue( const char *identifier,
                                       double &v) = 0;
virtual bool getParameterDefaultValue( const char *identifier,
                                       char *v);
virtual bool getParameterDefaultValue( const char *identifier,
                                       int &v);
virtual bool getParameterDefaultValue( const char *identifier,
                                       bool &v);
```

The following function makes sense for `adjustable double` parameters. By means of this method 3DE4 will know the domain of definition of the parameter, which plays a certain role in optimization. Although 3DE4's *Matrix Tool* does not take into account these values in the current implementation, a reasonable parameter range is important for the *Parameter Adjustment Window*.

```
// returns range for adjustable double parameters...
virtual bool getParameterRange( const char *identifier,
                                double &a, double &b) = 0;
```

3.2.1.3 Modifying parameter values

The following four methods are used for setting the value of a parameter. There are seven pre-defined parameter names, and every plugin class must be able to understand them. These parameters are focal length, filmback width and height, lens center offset x and y, and pixel aspect. Since each distortion model has double parameters (at least the seven mentioned before), the method for double parameters is pure virtual, i.e. it must be implemented, while the others are only virtual. If your model does not have `int`, `toggle`, `string` or `file` parameters, you can simply ignore them.

```
// set parameter values...
// parameters predefined by 3DE4:
//  "tde4_focal_length_cm", "tde4_filmback_width_cm", "tde4_filmback_height_cm",
// "tde4_lens_center_offset_x_cm", "tde4_lens_center_offset_y_cm", "tde4_pixel_aspect",
// "tde4_custom_focus_distance_cm"
virtual bool setParameterValue(const char *identifier, double v) = 0;
virtual bool setParameterValue(const char *identifier, char *v)
       { return false; }
virtual bool setParameterValue(const char *identifier, int v)
       { return false; }
virtual bool setParameterValue(const char *identifier, bool v)
       { return false; }
```

3.2.1.4 Preparing the model

The following method must be called, whenever one or more parameters have been changed. Some distortion models may require preparations when one or more parameters have been changed. These are done within this method.

```
// prepare the current set of parameters...
virtual bool initializeParameters() = 0;
```

3.2.1.5 Jacobi-Matrix

The following method calculates the Jacobi-Matrix by means of difference quotients. If you know the analytic form of this matrix, please implement this method in your derived class. We use this in order to generate export data for compositing systems.

```
virtual bool getJacobianMatrix( double x0, double y0,
       double &m00, double &m01, double &m10, double &m11) {...}
```

3.2.1.6 Remove and apply lens distortion

Finally, there are three methods which apply or remove lens distortion from a point. In 3DE4, we assume, that removing distortion from a point is a *simple* function in the sense that it can be done non-iteratively. At least, if it is done iteratively no initial values are required. All built-in polynomial models of 3DE4 are constructed this way. On the other hand, even the simplest polynomial models can only be inverted by using an iterative function, which requires good initial values. Therefore, there is only one method `undistort()`, while `distort()` comes in two flavours: one without initial values and one that demands initial values. 3DE4 will use the initial value version whenever possible. If your model does not need initial values for applying distortion, you may simply ignore the second version of this method (see default implementation below).

```
// warp/unwarp 2D points...
virtual bool undistort( double x0, double y0,
                        double &x1, double &y1) = 0;
virtual bool distort( double x0, double y0,
                        double &x1, double &y1) = 0;
virtual bool distort( double x0, double y0,
                        double x1_start, double y1_start,
                        double &x1, double &y1)
       { return(distort(x0,y0,x1,y1)); }
```

The point (x_0, y_0) passed to either of these methods as well as the resulting point (x_1, y_1) are given/calculated in unit coordinates as described in section 2.4.1.2.

3.3 Building a compositing node

3.3.1 Classes

Let us assume, the compositing node is represented by some class `NODE`. In principle, there are several ways to connect `NODE` to the distortion models.

1. Implement the mathematics from scratch in `NODE`.
2. Implement a distortion class, derived from `ldpk::general_distortion_base` and use it as member in `NODE`.
3. Implement a complete 3DE4-plugin, based on `tde4_ld_plugin` and use it as member in `NODE` (or even load it dynamically!?).

The third method has the advantage, that **once you have done this, it works for all plugins**, since the plugin base class enforces a common API for all distortion models. In the following we will assume that in fact all warping and unwarping is done by a plugin class within `NODE`. The plugin classes for all built-in models of 3DE4 are part of the LDPK.

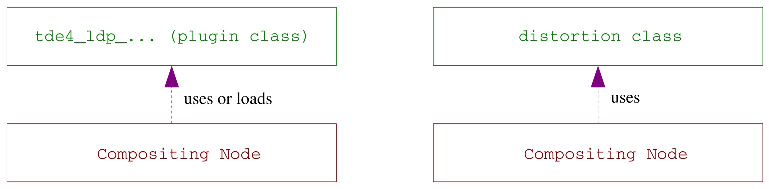

Figure 4: Compositing node

3.3.2 Unit coordinates and pixel coordinates

In developing a compositing node, implementation details will depend on the specifications of the compositing system. But it is very likely, that we will have to deal with some kind of pixel-based coordinate system, and for this case we should discuss the transformation between unit coordinates as used in the LDPK plugin API and pixel coordinates as used in image processing. For simplicity we shall assume, that the origin of the pixel coordinates is the lower left pixel. The data in an image file do not necessarily coincide with the image data relevant for processing. The situation may be as described in fig. 5 Filmback width and filmback height correspond to

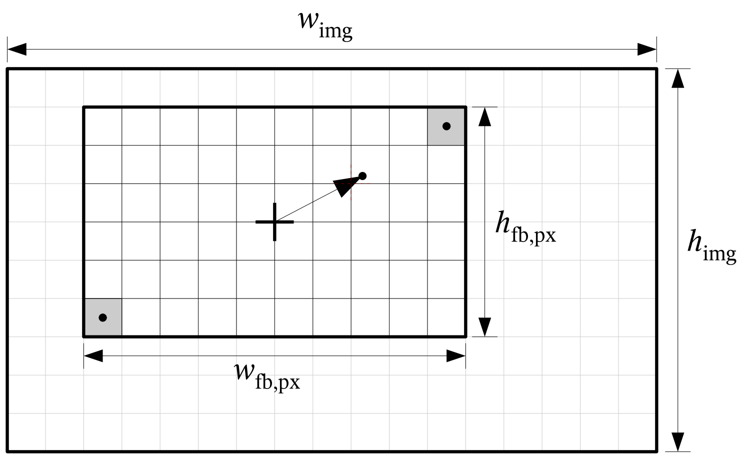

Figure 5: Image file and image data

width and height of the subimage, not to width and height of the image file. When we deal with pixel positions in the following, we *always* talk about pixel positions with respect to the subimage. This is important, because otherwise distortion will not be removed or applied correctly.
Let us assume the subimage of the image, that represents the filmback has a size of

$$w_{\text{fb,px}} \times h_{\text{fb,px}}. \tag{37}$$

A single pixel, like e.g. $(0,0)$ in the figure is some color information *associated to the center* of a (by definition) square-shaped area. So, a pixel $(x_{\text{px}}, y_{\text{px}})$ is mapped from unit coordinates $(x_{\text{unit}}, y_{\text{unit}})$ by the following mapping:

$$\begin{aligned} x_{\text{px}} &= x_{\text{unit}} w_{\text{fb,px}} - \frac{1}{2} \\ y_{\text{px}} &= y_{\text{unit}} h_{\text{fb,px}} - \frac{1}{2} \end{aligned} \tag{38}$$

The inverse mapping is:

$$\begin{aligned} x_{\text{unit}} &= \frac{x_{\text{px}} + \frac{1}{2}}{w_{\text{fb,px}}} \\ y_{\text{unit}} &= \frac{y_{\text{px}} + \frac{1}{2}}{h_{\text{fb,px}}}. \end{aligned} \tag{39}$$

We denote the mapping from unit to pixel coordinates and its inverse by

$$\psi_{\text{px}\leftarrow\text{unit}} \text{ and } \psi_{\text{unit}\leftarrow\text{px}} \tag{40}$$

3.3.3 Removing and applying lens distortion

Now, the aim of an image processor is to calculate the pixel $(x_{\text{px}}, y_{\text{px}})$ of the target image. We would like to use the inverse distortion mapping g^{-1} in order to find out, from which pixels the color at $(x_{\text{px}}, y_{\text{px}})$ has to be mixed, as already shown in fig.3. In order to do this, we have to transform the inverse distortion function from unit coordinates to pixel coordinates. Let g^{-1}_{unit} be the inverse distortion function in unit coordinates, i.e. as defined in the LDPK plugin. Then the inverse distortion function in pixel space is

$$g^{-1}_{\text{px}} = \psi_{\text{px}\leftarrow\text{unit}} \circ g^{-1}_{\text{unit}} \circ \psi_{\text{unit}\leftarrow\text{px}}, \tag{41}$$

or in words: map from pixel to unit coordinates, apply the inverse plugin distortion function and map back to pixel coordinates. Similar, applying distortion to an image in pixel space is done with

$$g_{\text{px}} = \psi_{\text{px}\leftarrow\text{unit}} \circ g_{\text{unit}} \circ \psi_{\text{unit}\leftarrow\text{px}}, \tag{42}$$

When you implement an image processor, your pixel positions $(x_{\text{px}}, y_{\text{px}})$ will probably be pairs of integer values. The result after applying g^{-1}_{px} however will in general be a pair of real numbers, the non-integer part of which is used to interpolate between neighbouring pixels in the original image. This is most likely done by the compositing system you are writing the node for.

3.3.4 Resize and reapply

In warp4, SDV's image processing tool, several modes for removing or applying lens distortion are available. If you implement a compositing node, it might be helpful to describe, how these modes are implemented. A workflow which is occasionally asked for by users is the following:

1. Given a sequence of original, distorted images, warp4 is used to remove distortion.
2. The undistorted sequence is used in compositing.
3. warp4 is then used to re-apply distortion.

If this is done without modifying the size of the image, undefined (i.e. black) areas occur at the edge of the image, because removing lens distortion usually *magnifies* the image content[3]. Then some parts of the content are lost. Therefore warp4 is able to *resize* the image while removing distortion. In the following, we describe how this is done.

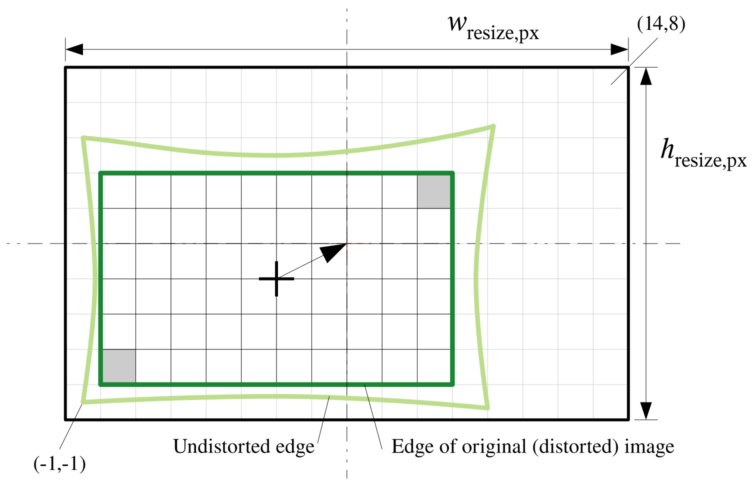

Figure 6: Resize and reapply, details

3.3.4.1 Resize Fig. 5 shows the original content (dark green area). Now, we imagine that removing distorion required some additional space (light green area). The outer box, i.e. the resized area is determined by two conditions:

1. The undistorted content is inside the resized area.
2. The lens center is exactly the center of the resize area.

The first of these conditions is clear: we do not want to lose any image content. The second condition has more technical reasons: after removing lens distortion we do not know the position of lens center with respect to the resized image. Therefore we place it at a special position, namely the image center.

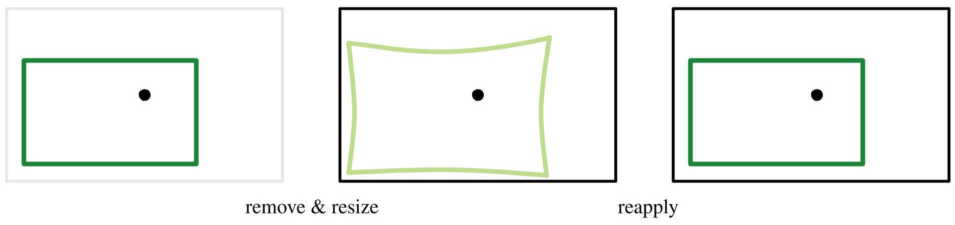

Figure 7: Resize and reapply in warp4

3.3.4.2 Reapply After removing lens distortion with resizing we have a new image with size $w_{\text{resize,px}} \times h_{\text{resize,px}}$. Since we defined a distortion function as invertible, it should be possible to retrieve the original image. The problem is: expressed in real length units, our resized image has a different filmback. A distortion function for retrieving the original image would have completely different parameters. The solution is to redefine the pixel-based coordinate system. For our example in fig. 6 this means, instead of pixel coordinates [0,0] to [15,9] we use pixel coordinates [-1,-1] to [14,8]. By doing so, we can avoid dealing with the (unreal) filmback of the resized image, and we do not need to keep track of the lens center. The reason for us to handle resizing and reapplying like this is more or less induced

[3]More precise, in a small area around lens center the image remains unchanged, while towards the edge it is stretched.

by warp4. You may handle this according to your needs in a different way for your compositing node.

A Table of symbols

$\delta_{\mu\nu}$	*delta*	KRONECKER-delta
$a \cdot b$	*dot*	Inner product of two vectors
$a \otimes b$	*dyadic*	Dyadic product of two vectors
f_{cm}		Focal length in centimeter
g_{px}		Distortion function for pixel coordinates (compositing)
g_{unit}		Distortion function for unit coordinates (plugin API)
I		Unit interval [0,1]
$\lvert \cdots \rvert$	*norm*	EUCLIDIAN norm of a vector
$\psi_{\mathrm{unit}\leftarrow\mathrm{cm}}$	*psi*	Eq. (22) Map from centimeter to unit coordinates
$\psi_{\mathrm{cm}\leftarrow\mathrm{unit}}$	*psi*	Eq. (23) Map from unit coordinates to centimeter
$\psi_{\mathrm{unit}\leftarrow\mathrm{dn}}$	*psi*	Eq. (24) Map from diag norm to unit coordinates
$\psi_{\mathrm{dn}\leftarrow\mathrm{unit}}$	*psi*	Eq. (25) Map from unit to diag norm coordinates
$\psi_{\mathrm{dn}\leftarrow\mathrm{cm}}$	*psi*	Eq. (26) Map from centimeter to diag-norm coordinates
$\psi_{\mathrm{cm}\leftarrow\mathrm{dn}}$	*psi*	Eq. (25) Map from diag-norm coordinates to centimeter
$\psi_{\mathrm{px}\leftarrow\mathrm{unit}}$	*psi*	Eq. (38) Map from unit coordinates to pixel coordinates
$\psi_{\mathrm{unit}\leftarrow\mathrm{px}}$	*psi*	Eq. (39) Map from pixel coordinates to unit coordinates
$\mathbb{R}$		The real numbers
$\mathbb{R}^n$		The space of real-valued n-tuples
$r_{\mathrm{fb,cm}}$		Filmback diagonal radius in centimeter
r_{pa}		Pixel aspect (ratio)
$w_{\mathrm{fb,cm}}, h_{\mathrm{fb,cm}}$		Filmback width and height in centimeter
$w_{\mathrm{fb,px}}, h_{\mathrm{fb,px}}$		Filmback width and height in pixel
$w_{\mathrm{resize,px}}, h_{\mathrm{resize,px}}$		Resized filmback width and height in pixel
$x_{\mathrm{lco,cm}}, y_{\mathrm{lco,cm}}$		Lens center offset in centimeter
$x_{\mathrm{lco,unit}}, y_{\mathrm{lco,unit}}$		Lens center offset in unit coordinates
$x_{\mathrm{cm}}, y_{\mathrm{cm}}$		A point in centimeter
$x_{\mathrm{dn}}, y_{\mathrm{dn}}$		A point in diagonally normalized coordinates
$x_{\mathrm{px}}, y_{\mathrm{px}}$		A point in pixel coordinates
$x_{\mathrm{unit}}, y_{\mathrm{unit}}$		A point in unit coordinates

References

[1] Bernhard Haumacher: `svg2office`, `http://www.haumacher.de/svg-import/`, last verified: 2011-02-21