Content-adaptive Lenticular Prints
Supplemental Material

James Tompkin
Disney Research/MPI für Informatik

Simon Heinzle
Disney Research Zurich

Jan Kautz
University College London

Wojciech Matusik
MIT CSAIL

This document supports the main paper by providing further explanations, results, a worked example, and detailed pseudocode for our optimization techniques. Section 1 explains our lens optimization in detail, which determines optical aspherical lens shapes and complementary focal surfaces given an input lens width size in pixels and the field of view. Section 2 presents our discrete light field optimization, which places pre-determined lenses of set widths to optimally trade spatial and angular resolution over a light field given the desired physical dimensions of the output display. This includes a worked example using a simple formalization. Section 3 explains further the spatial vs. angular weighting. Section 4 explains further the multi-scanline solve. Section 5 provides further light field result comparisons with Figures 9, 10, 11, and 12. Finally, Section 6 contains pseudocode for the lens and discrete lens configuration optimizations.

1 Lens Optimization

The generated display is a series of scanlines, each split into a series of lenses. The physical size of the display and the size of a pixel determines the number of pixels in a scanline. The discrete optimization (Section 2) computes a set of lens widths in pixels (and assigns color to these pixels) such that the spatial and angular resolution of the input light field is optimally distributed. Given this lens width set, the lens optimization generates a refractive lens surface and an image surface which best focuses the output light field.

1.1 Plano-Convex Lenses

We begin by describing an optimization for 1D plano-convex lenses (Figure 4, main paper, left and center-left). For each lens, we have as input constants:

1. Lens width W, in pixels.
2. Field of view V, in degrees.
3. Refractive index of lens material.

Our strategy is to posit a number of viewing positions over the field of view equal to the number of pixels underneath the lens. From these viewing positions, we ray trace candidate lens surfaces given a parameterized lens surface model, refract the rays, and compute where they would intersect an image surface some focal distance from the lens surface. We split this lens surface into pixels, with each viewing position mapping to a target pixel. From this, we can compute an error function: the distance of each incoming ray from its target pixel center.

Figure 1 shows an example candidate lens surface and image surface at a focal distance. The rays incoming to this lens are distributed across 5 color-coded positions (purple, blue, green, yellow, and red). Each ray position is intended to view one pixel. The error function is then computed as the mean squared error from all distances of the ray/image surface intersections from the center of the ray's assigned pixel.

With our error function defined, we now need to optimize over our parameters. One of these will be the focal distance, and the others will describe our lens surface shape.

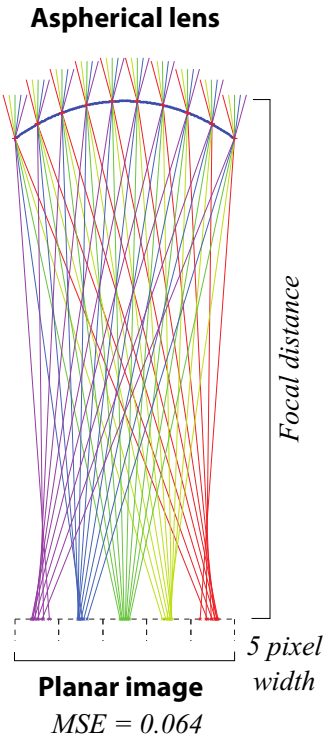


Figure 1: *Plano-convex lens optimization example.*

The lens surface is defined as [Pruss et al. 2008; Wikipedia a]:

$$z(r) = \frac{r^2}{R\left(1+\sqrt{1-(1+\kappa)\frac{r^2}{R^2}}\right)} + \alpha_1 r^2 + \alpha_2 r^4 + \alpha_3 r^6 + \cdots, \quad (1)$$

where the optic axis lies in the z direction, R is the radius of curvature (if α terms are 0), and κ is the conic constant which determines the type of conic section generated, with hyperbolas when $\kappa < -1$, parabolas when $\kappa = -1$, spheres when $\kappa = 0$, to ellipses when $-1 < \kappa < 0$ (a prolate spheroid) and $\kappa > 0$ (an oblate spheroid). We refer the reader to [Conics and Aberrations], Figure 23, for an elegant diagram exploring these relationships. Figure 2 shows a chord generated by our system, with accurate normals generated for all incoming ray points of intersection.

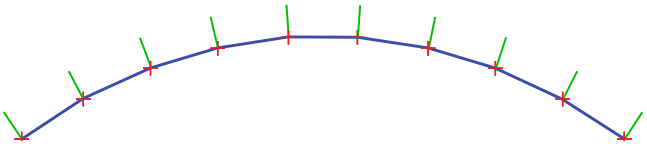


Figure 2: *Aspherical chord. Green lines are normals computed at the red points of ray intersection.*

Thus, the parameters we wish to optimize are:

1. R, the radius of curvature.
2. κ, the conic constant.
3. F, the focal distance.

with input constants W, the lens width, V, the field of view, and n, the refractive index of the lens material. In our case, using Objet VeroClear material, $n = 1.47$. We optimize these parameters using unconstrained nonlinear optimization. For our optimization, we do not use the α distortion terms, as these terms have only a tiny impact on the overall error while increasing the optimization space unnecessary.

We take our initial values from Kweon and Kim [Kweon and Kim 2007]. $R_0 = (1-n)/(n*F_0)$, assuming the index of refraction (IOR) = 1 for air ([Kweon and Kim 2007], Equ. 27). $\kappa_0 = -(1/n)^2$, again assuming IOR =1 for air ([Kweon and Kim 2007], Equ. 28). These values are optimal only for the center view and not for the whole field of view, thus requiring an optimization step. For the focal length, we use a simple field-of-view conversion: $F_0 = W/(2*tanV)$, which assumes a pinhole model.

It is also possible to use the Lensmaker's equation [Wikipedia b] as an initial approximation for F. For plano-convex lenses, we would set the second radius to infinity, so:

$$F_0 = R_0/(n-1) \tag{2}$$

However, this is not optimal for any views and is only an approximation, whereas Kweon and Kim [Kweon and Kim 2007] is at least optimal for optical-axis-aligned views.

Alternative error functions: Other error functions are possible. One example: given that a pixel has a non-zero width, we can set the ray error to be zero so long as the ray doesn't stray into a neighboring pixel. This allows us to additionally minimize the focal distance F to create as thin a display as possible, whereas the center-based error minimizes overall crosstalk.

1.2 Lenses with non-planar image surfaces

As can be seen in Figure 1 in this document, and Figure 4 in the main paper, the optimal focusing distance across all viewing directions does not necessarily lie on a single plane.

Given the simple ray tracing setup described, we can form a curved back plane to match a given aspheric lens surface which further reduces the mean square error of ray intersections. Each viewing position defines a bundle of rays, and depending on the incoming angle these will exhibit a certain coma. If we compute the point of intersection between each pair of rays in the bundle, and then take the mean position of all these intersections, we will find a point that is the point of best focus (in least-squares sense). We can construct surfaces from these points by varying the incoming viewing position and fitting a curve to the series of points.

Further, the shape of the back pixel surface can be optimized jointly with the lens surface within the same unconstrained non-linear optimization. Given the simple ray tracing optimization described in the previous section, we modify the procedure as follows:

- To achieve axis-aligned pixel patches, each back pixel i is assigned its own independent focal length F_i. Subsequently, the optimization parameter F (same focal length for all pixels) is substituted with the optimization parameter $F = [F_1, F_2, , F_n]$ (individual focal length for each pixel), where n is the number of pixels. The ray-tracing routine then intersects with the staircase surface rather than with the even surface, as illustrated in Figure 4 in the main paper.
- To achieve the linear approximation of a curved surface, we change the optimization parameter to $F = [F_1, , F_{n+1}]$. Then, the pixel surface of pixel i is described by the linear patch defined by points F_i and F_{i+1}, see Figure 4 in the main paper.

Section 6.1 provides pseudocode for the plano-convex optimization.

2 Discrete Light Field Optimization

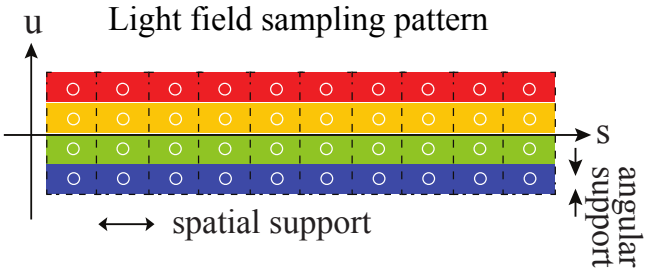


Figure 3: *EPI slice regular sampling, showing fix-sized lenses covering the light field section.*

Our discrete optimization takes as input an epipolar image (EPI) slice from a light field. Each slice represents a scanline in the final display. The EPI slice has, for its s axis, the spatial content, and for its u axis, the angular content (Figure 3). Each column (dashed black) represents one lens. Each box within the lens containing a circle is one pixel in the output image, representing the different view positions in space that this would generate with a lens on top. In this example, each view contains different content — rotating the display in front of your eyes would produce a display which switches from red to yellow to green to blue.

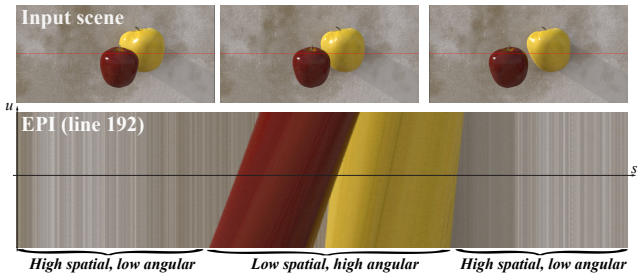


Figure 4: *EPI slice showing different spatial and angular content within a light field.*

However, different light field content induces different lens/pixel sampling requirements. In Figure 4, we see a real example. The plot axes are the same, u and s. The crux of understanding how we can adaptively sample comes with the intuition that a pixel in the display output has a fixed area in this u,s plot.

With no lens, an array of pixels can only provide one view. When trading off angular and spatial resolution, this is the maximum spatial resolution and the minimal angular resolution. Overlaid on our u,s plot, these single output pixels would be unit columns — all the u pixels are merged into one output pixels, giving minimal angular resolution. The color assigned to the output pixels is the mean color of all input pixels underneath the u extent, and so the unit wide column (no lens) then blurs content in the u direction.

If we now look at a lens with 20 pixels, we see the opposite situation. Now, our lens would be a 20 unit column in width. This lens has 20 pixels, each directed by refraction to different view directions in space, and so the u dimension is partitioned into 20 sections to provide good angular sampling. By the fact that a pixel has constant area in the u,s plot, each of these output pixels has horizontal extent and is 20 units wide (e.g., Figure 2 main paper). Again, the output color assigned is the mean color of all input pixels underneath the u,s extent of that output pixel, an so each pixel blurs content in the s direction.

Our goal is to analyze the light field and consider which lenses are best suited to which part of this EPI image light field slice. We wish to find a configuration of lenses which optimally samples the content to minimize the difference between the input and output light fields.

The general strategy is straightforward: place all lenses in all possible configurations, compare the input and output light fields, and see which configuration produces the least error. However, even for small numbers of lenses, a brute-force computation method soon becomes intractable. Instead, we apply a dynamic programming approach, which works recursively and stores previous results to cut down on the number of possible lens configurations. We will demonstrate this with a worked example, before presenting pseudocode (Section 6.2).

2.1 General Case

Notation:

- Input EPI slice EPI_I with width W.
- Output EPI slice EPI_O with width W.
- Candidate lens set L with widths L_w, where $L[y]$ denotes the y-th lens in the candidate lens set.
- Optimal lens set L_O.
- Data table T.

Operations:

- $EPI[1:n]$ means to take a subset of the EPI slice along the s axis.
- $applyLens(x, L[y])$ means to apply the candidate lens $L[y]$ at s axis position x in EPI_I. Applying the lens means computing the color of each of the lens' constituent pixels by taking a mean of EPI_I under that pixel's extent. This generates a candidate EPI_O in the range $[x : x + L_w[y]]$.
- $error(x)$ means to compare x by L2 distance to EPI_I.

We begin our approach by first applying the smallest lens to cover the right-most part of the slice:

$$applyLens(W - L_w[1], L[1]) \tag{3}$$

Comparing against EPI_I for this region, we generate an error for placing this lens in that position:

$$error(applyLens(W - L_w[1], L[1])) \tag{4}$$

The hypothetical remainder of the slice, from 0 to the lens just placed at the right-most part of the slice, is $EPI_O[0 : W - L_w[1]]$. At this stage, $error(EPI_O[0 : W - L_w[1]])$ is unknown as no minimal error has been computed; but, the total error for placing $L[1]$ in the right-most position would be:

$$error(EPI_O[0 : W - L_w[1]]) + error(applyLens(W - L_w[1], L[1])) \tag{5}$$

Now, let's do the same for the second smallest lens, applying it in the right-most position possible in the input slice. The error for this placement is the error of applying the lens, plus the hypothetical error of the rest of the slice:

$$error(EPI_O[0 : W - L_w[2]]) + error(applyLens(W - L_w[2], L[2])) \tag{6}$$

Let $E(q) = error(EPI_O[0 : W - L_w[q]]) + error(applyLens(W - L_w[q], L(q)))$. Then, if we suppose $E(q)$ for all lenses in L, placing each at the right-most position possible, we come to form the hypothetical error for L_O applied to EPI_I:

$$L_O\text{error} = min(E(1), E(2), \ldots, E(n)); \tag{7}$$

Given this, we can generate an algorithm which recurses to solve for every $EPI_O[0 : W - L_w[n]]$. The algorithm becomes dynamic programming when we start adding computed errors to T. We want to store the lens combinations and their produced errors for all x positions along the s axis.

For instance, during the recursion, if we know that the least erroneous way to apply lenses to $EPI_O[0:3]$ is with a lens of width 3 (i.e., $L_O = \{3\}$, rather than with $L_O = \{1,1,1\}$, $L_O = \{1,2\}$, or $L_O = \{2,1\}$), then we never again need to compute lens configurations for this portion of the slice and can just look up the error. This significantly cuts down on the number of lens combinations that need to be tested to find the one with the least error.

2.2 Worked Example

We describe this worked example using the formulation from the previous section, with the help of Figure 5 for visualization. In each recursion step, all possible lenses from a discrete set of candidates are placed on one side of the input light field (shaded in gray). Then, the same procedure is evaluated recursively on the remaining subset of the light field not covered by the lens (shaded in blue). The recursion stops as soon as the width of the remaining subset is equal or smaller to the smallest candidate lens (shown in row 5). Then, the error induced by the current lens placement is propagated up, and in each recursion stage the lens arrangement with the lowest error is selected (green boxes).

Level 5: We start with EPI_I slice width $W = 5$ and a lens set L with widths $L_w = \{1,2,4\}$. Then:

$$\begin{aligned} L_O\text{error} = min(&error(EPI_O[0:4]) + error(applyLens(4, L[1])),\\ &error(EPI_O[0:3]) + error(applyLens(3, L[2])),\\ &error(EPI_O[0:1]) + error(applyLens(1, L[3]))). \end{aligned}$$

We can directly compute the error terms for applying the lenses, but we need to recurse to compute $error(EPI_O[0:4])$, $error(EPI_O[0:3])$, and $error(EPI_O[0:1])$.

Level 4: $error(EPI_O[0:4])$

$$\begin{aligned} L_O\text{error} = min(&error(EPI_O[0:3]) + error(applyLens(3, L[1])),\\ &error(EPI_O[0:2]) + error(applyLens(2, L[2])),\\ &error(applyLens(0, L[3]))). \end{aligned}$$

Again, we can directly compute the *applyLens* terms, but must recurse for $error(EPI_O[0:3])$. At this level, we see that there is no further error for $L[3]$ — the EPI_I subset is 4 wide, and $L[3]$ is 4 wide. Now, we need to recurse again.

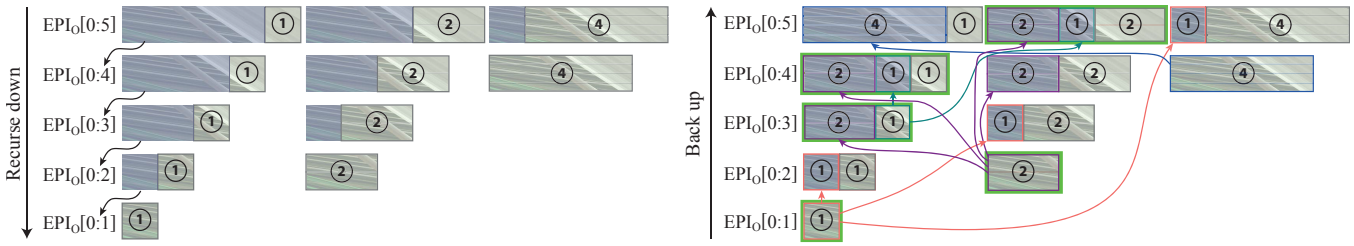


Figure 5: *This figure sketches an example of our discrete optimization method to optimally distribute a set of lenses based on a local analysis of the 2D light field. We are given an input light field of width $W = 5$ and a set of candidate lenslets L with widths $L_w = \{①, ②, ④\}$. Each lenslet covers a (rectangular) sub-area of the input light field. The algorithm starts with the full light field width $EPI_O[0:5]$. It places candidate lenslets (starting with $L[1] = ①$) on the right of the light field and recurses to find the optimal lens distribution for the remaining left subset. This recursion continues until the base case of $EPI_O[0:1]$ and $L[1] = ①$. The incurred error for placing this single lens is then used (orange arrow) to find the optimal lenslet distribution for $EPI_O[0:2]$, as it now allows us to compute the error for each of the two possible lens configurations ①① and ②. ② has the lower error and is recorded as the optimal lens distribution (marked with bold green box) for $EPI_O[0:2]$. This continues back up to $EPI_O[0:5]$, where ②①② has the lowest error and is the final optimal distribution.*

Level 3: $error(EPI_O[0:3])$

$$L_O\text{error} = min(error(EPI_O[0:2]) + error(applyLens(2, L[1])), \\ error(EPI_O[0:1]) + error(applyLens(1, L[2]))).$$

We continue to recurse.

Level 2: $error(EPI_O[0:2])$

$$L_O\text{error} = min(error(EPI_O[0:1]) + error(applyLens(1, L[1])), \\ error(applyLens(0, L[2]))).$$

At this level, we see that there is no further error for $L[2]$ — the EPI_I subset is 2 wide, and $L[3]$ is 2 wide.

Level 1: $error(EPI_O[0:1])$

$$L_O\text{error} = min(error(applyLens(0, L[1]))).$$

We've now hit the bottom of our recursive stack, and we can start adding values to table T. As the application of $L[1]$ is at position 0, we can also add it to the table T. We imagine this lens placement has an error of 10:

EPI subset	Error	Lens configuration
0:1	10	{L[1]}

Level 2: $error(EPI_O[0:2])$

$$L_O\text{error} = min(10 + error(applyLens(1, L[1])), \\ error(applyLens(0, L[2]))).$$

Now, we can begin to work up the stack, using entries from table T to replace terms. Lets suppose that, in this case, the error for two single lenses is more than for one 2-wide lens. Let's add it to T, and work back up the stack.

EPI subset	Error	Lens configuration
0:1	10	{L[1]}
0:2	22	{L[2]}

Level 3: $error(EPI_O[0:3])$

$$L_O\text{error} = min(22 + error(applyLens(2, L[1])), \\ 10 + error(applyLens(1, L[2]))).$$

From T, we look up the two subset errors. The lens configurations presented are either $L_O = \{L[1], L[2]\}$ or $L_O = \{L[2], L[1]\}$. We suppose the second has the lowest error of 24 and add it to T:

EPI subset	Error	Lens configuration
0:1	10	{L[1]}
0:2	22	{L[2]}
0:3	24	{L[2],L[1]}

Level 4: $error(EPI_O[0:4])$

$$L_O\text{error} = min(24 + error(applyLens(3, L[1])), \\ 22 + error(applyLens(2, L[2])), \\ error(applyLens(0, L[3]))).$$

We suppose that applying the 4-wide lens ($L[3]$) or the $L_O = \{L[2], L[2]\}$ configuration have higher errors. Hence, we add $L_O = \{L[2], L[1], L[1]\}$ to T.

EPI subset	Error	Lens configuration
0:1	10	{L[1]}
0:2	22	{L[2]}
0:3	24	{L[2], L[1]}
0:4	28	{L[2],L[1],L[1]}

Level 5: $error(EPI_O[0:5])$

$$L_O\text{error} = min(28 + error(applyLens(4, L[1])), \\ 24 + error(applyLens(3, L[2])), \\ 10 + error(applyLens(1, L[3]))).$$

Finally, we will suppose that the second lens configuration option is cheapest and add it to T. We have arrived back at the full EPI slice width. We can now read the $W = 5$ lens configuration from T. This recursive method, which continually saves the best result for EPI subsets, saves us from having to brute-force search every possible option.

EPI subset	Error	Lens configuration
0:1	10	{L[1]}
0:2	22	{L[2]}
0:3	24	{L[2], L[1]}
0:4	28	{L[2],L[1],L[1]}
0:5	34	{L[2],L[1],L[2]}

3 Spatial vs. Angular Weight Variations

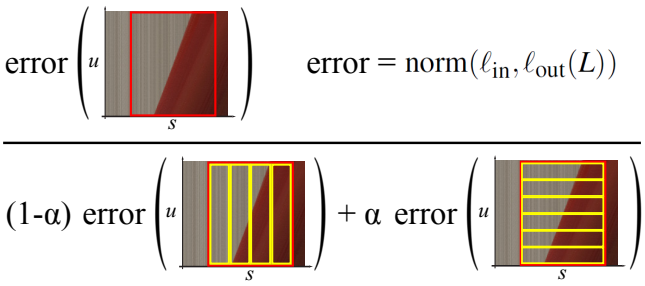


Figure 6: *Spatial vs. angular weight variations.* Top*: With no weighting terms, the error for the application of a lens (in red) to the EPI slice is computed by comparing the output color of each lens pixel (as the mean of all pixels within the pixel's extent) against all pixels in the input light field within the pixel's extent.* Bottom*: The spatial error and the angular error are computed independently within the lens. Two error values are computed, one row-wise and one column-wise: each row/column of pixels in the input light field (in yellow), within each output pixel extent, are compared using the L2 norm, and these two different error sums are weighted using* α *to produce the final error for this lens application.*

The L2 error described in Equation 1 in the main paper can directly be computed in the light field domain; however, the angular and spatial dimensions can be re-weighted if desired. Instead of computing a 2D box filter along both dimensions, we can compute the angular and spatial errors individually: the L2 norm is computed along the spatial direction (summing up 2D box columns) and the angular direction (summing up 2D box rows). The resulting errors can then be combined using a spatial-angular weighting factor. This way, the lens generation can be steered to either focus on reproducing the angular variation or on reproducing spatial variation. Figure 6 demonstrates this for one lens application in an EPI slice. All results in this paper are computed with equal weights for both dimensions. Figure 7 overleaf shows this effect on the Train and Elephant datasets. Predictably, high spatial weight maintains crisp edges and blurs content with large angular variation, and high angular weight places large lenses for good angular extent but lowers spatial resolution. All our results produced used a balanced weighting.

4 Multi-scanline Solve

We use a sliding window of multiple scanlines to compute the lens arrangement for each particular scanline. The lens error on one scanline is then computed by summing the errors supposing the lens were also applied to neighboring scanlines (Figure 8).

Section 6.2 provides pseudocode for the base discrete optimization case with even spatial vs. angular weighting and single scanline solving.

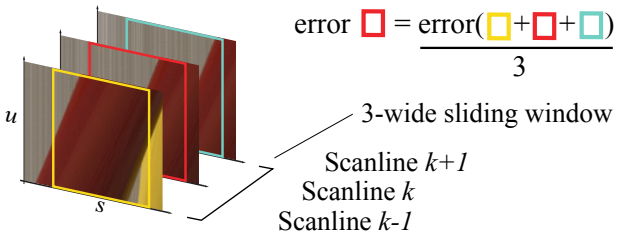
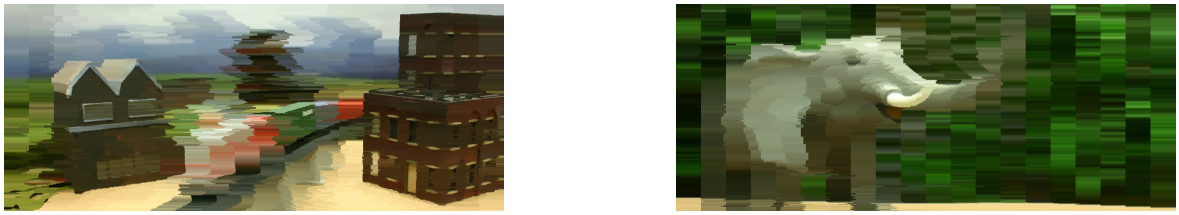


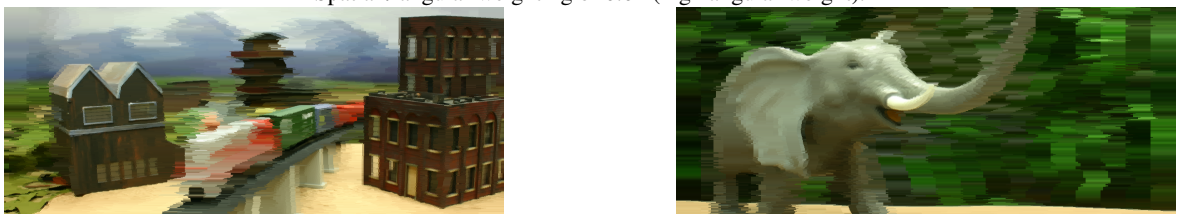
Figure 8: *Multi-scanline solve. EPI slices in a window, i.e., neighbouring scanlines, have the candidate lens applied. The L2 errors from each scanline are averaged to create the final lens error.*

5 More Simulation Results

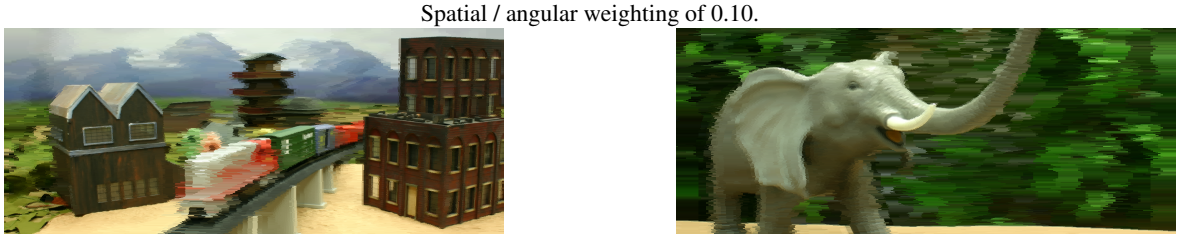
Figures 9, 10, 11, and 12 present further result comparisons.



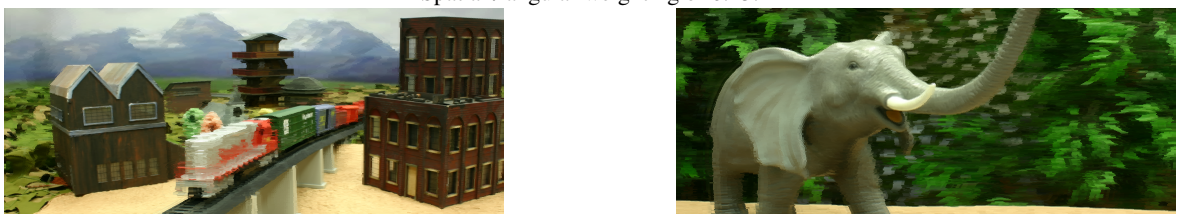
Spatial / angular weighting of 0.01 (high angular weight).



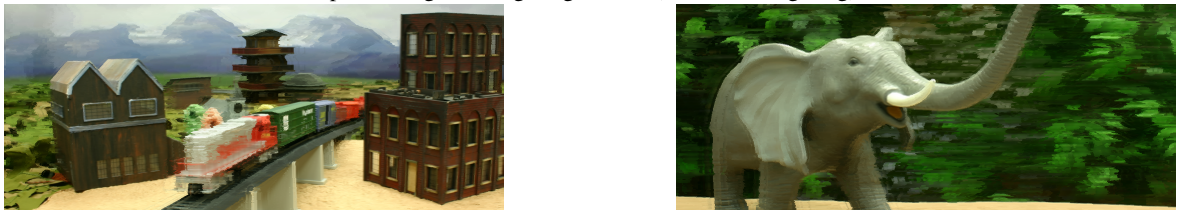
Spatial / angular weighting of 0.10.



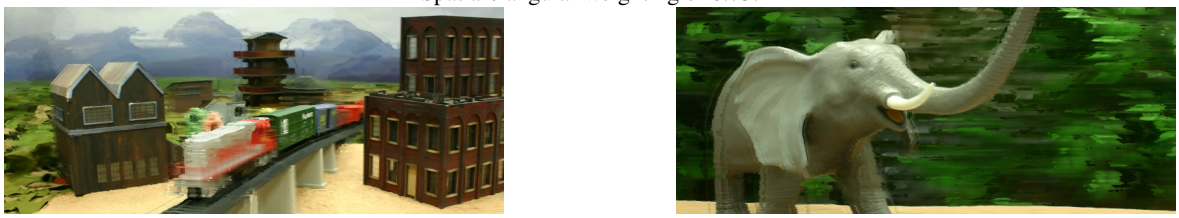
Spatial / angular weighting of 0.25.



Spatial / angular weighting of 0.50 (balanced weighting).



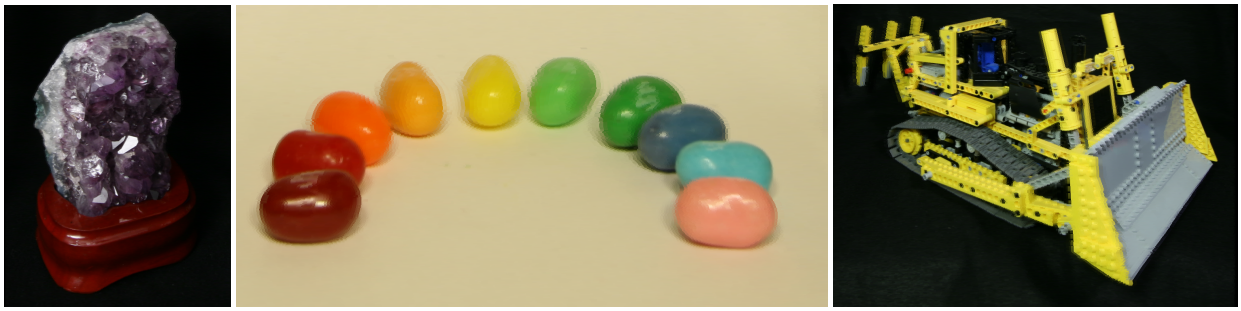
Spatial / angular weighting of 0.75.



Spatial / angular weighting of 0.90.

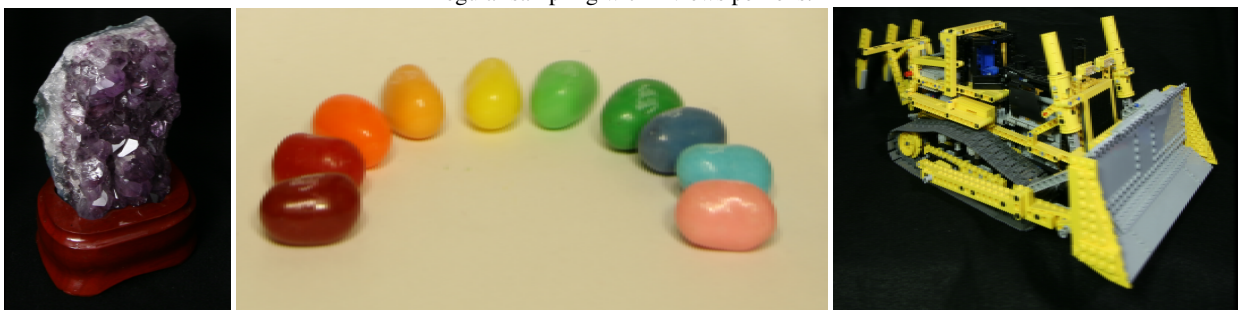
Spatial / angular weighting of 0.99 (high spatial weight).

Figure 7: *Spatial and angular weight variations for the train and elephant test scenes, with the front of the house set to be at zero depth ('on' the resulting lenticular display) in the train scene and the tusk similarly in the elephant scene. We show one frame simulating a view of the display parallel to the optical axis of the lenses. High spatial weighting causes angular components to blur, such as the backgrounds, but maintains spatial resolution. High angular weighting places large lenses (single strips of horizontal color in this visualization) to capture the angular content.*

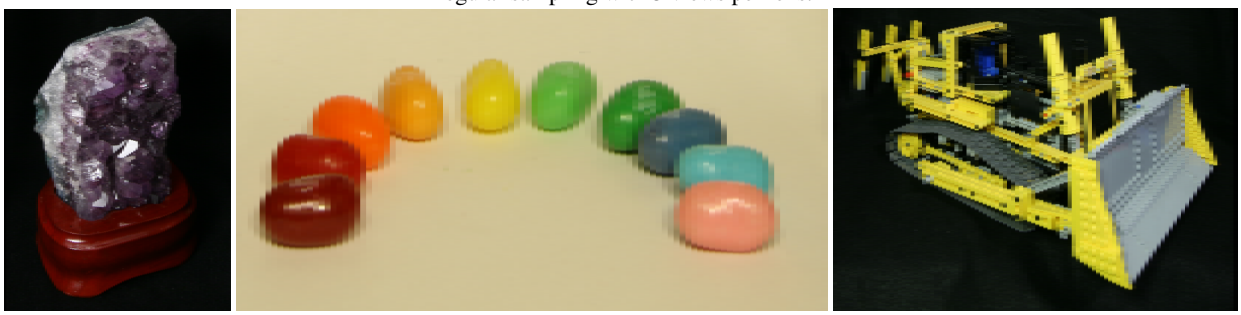


Adaptive sampling with 2 to 17 views per lens.

Regular sampling with 2 views per lens.



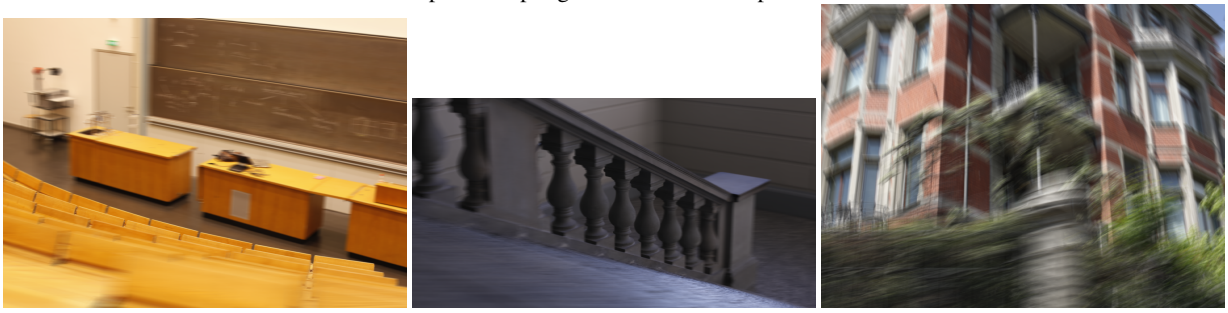
Regular sampling with 5 views per lens.



Regular sampling with 10 views per lens.

Figure 9: *Results for the Amethyst, Jelly Beans, and Lego Bulldozer data set.*

Adaptive sampling with 2 to 20 views per lens.



Regular sampling with 2 views per lens.

Regular sampling with 5 views per lens.

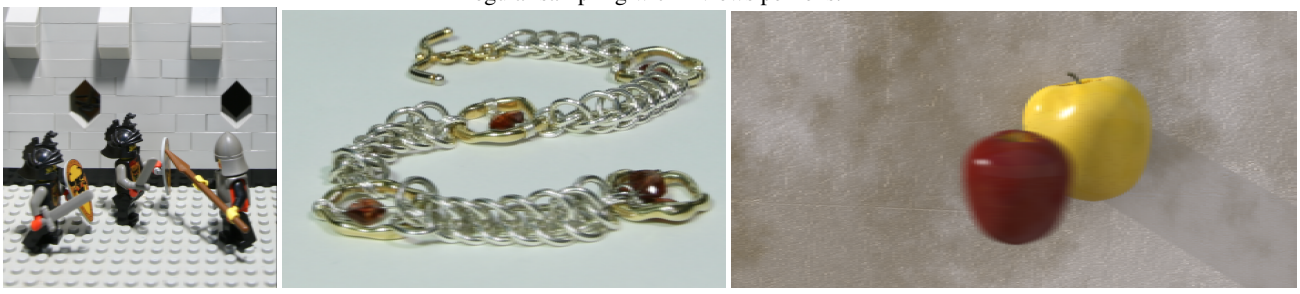
Regular sampling with 10 views per lens.

Regular sampling with 20 views per lens.

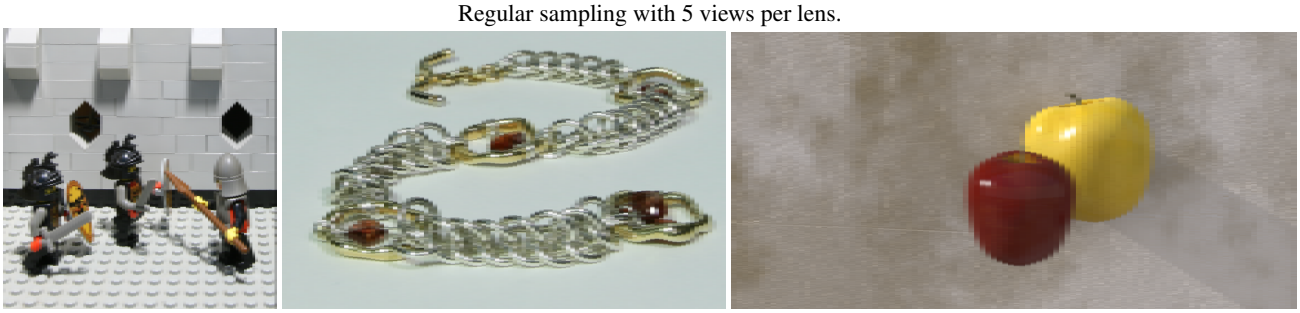
Figure 10: *Results for the Lecture, Foyer, and Mansion data set.*

Adaptive sampling with 2 to 17 views per lens (2 to 20 for Pomme).

Regular sampling with 2 views per lens.



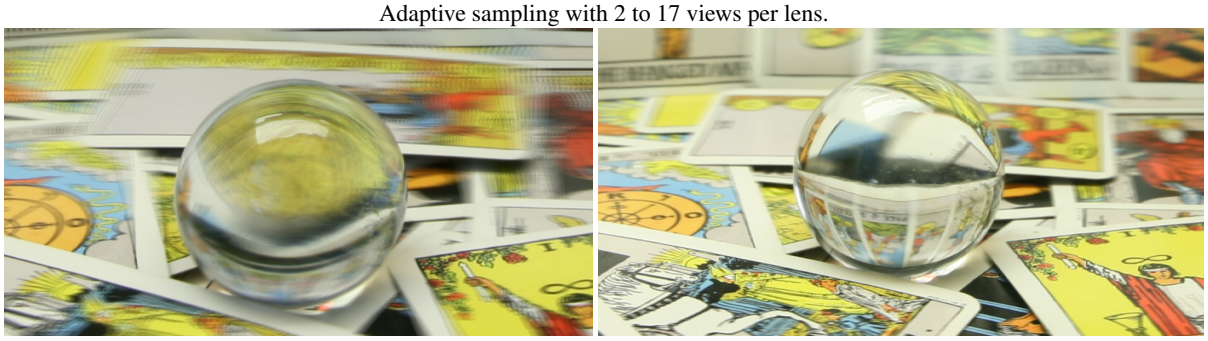
Regular sampling with 5 views per lens.



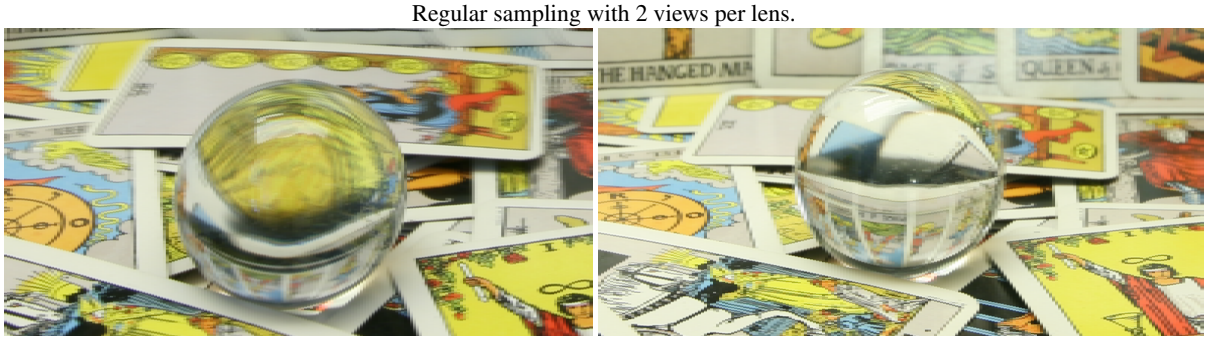
Regular sampling with 10 views per lens.

Figure 11: *Results for the Lego Knights, Necklace, and Pomme data set.*

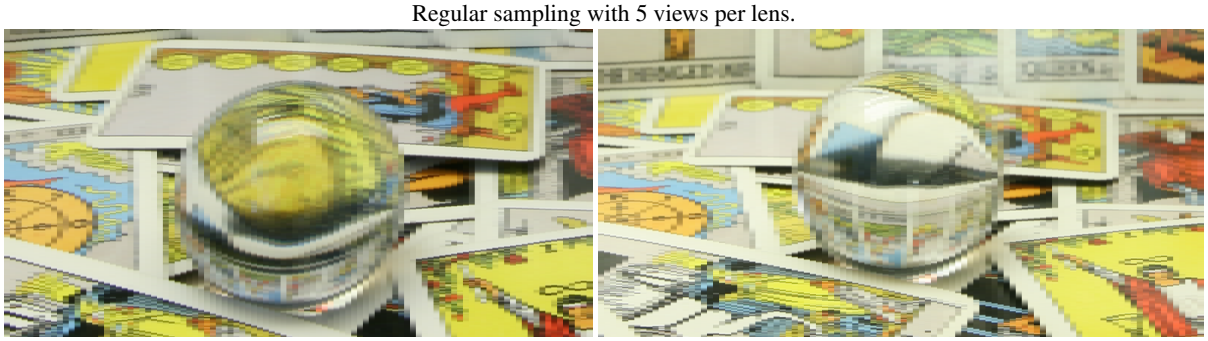
Adaptive sampling with 2 to 17 views per lens.



Regular sampling with 2 views per lens.



Regular sampling with 5 views per lens.



Regular sampling with 10 views per lens.

Figure 12: *Results for the Tarot Coarse and Tarot Fine data set.*

6 Pseudocode

6.1 Lens Optimization

6.1.1 Aspherical Lens

First, we show pseudocode to generate an aspheric conic section (Figure 2). This is represented as a chord of piecewise planar sections; for accurate refraction, normals are generated at all ray intersection points.

```
% Generate an aspherical chord conic section.
%
% Input:
%       points  = linspace(-W/2,W/2,NOP)
%                   where   W is lens width, NOP is number of points to generate
%       R       = radius of curvature
%       Kappa   = conic constant
%       Alpha   = distortion terms (does nothing currently)
%
% Output:
%       x       = x coordinates vector
%       y       = y coordinates vector
%       n       = 2 x length(points) vector holding normals
%
function [x,y,n] = getAsphericalChord( points, R, Kappa, Alpha )

% Determine points
psqr = points.^2;
pit  = psqr;

% Distortion term computation
a = zeros(size(psqr));
for i=1:size( Alpha,2 )
    a = a + Alpha(1,i)*pit;
    pit = pit .* psqr;
end

% Equation 2. Doesn't consider distortion terms
z = psqr ./ (R*(1+sqrt(1-(1+Kappa)*psqr/R^2)));

% Determine normals
pcube = points.^3;
sqrtTerm = sqrt(1-(1+Kappa)*psqr/R^2);

% Diff of first term
n = 2*points./(R*(1+sqrtTerm));
n = n + (1+Kappa)*pcube./(R^3*(1+sqrtTerm).^2 .*sqrtTerm);
n = [n;-ones(size(n))];

% Diff of perturbation terms
a = zeros(size(psqr));
pit = pcube;

% Distortion term diff
for i=1:size(Alpha,2)
    a = a + (2+2*i) * Alpha(1,i)*pit;
    pit = pit .* psqr;
end

% Normalize
norms = sqrt(sum(n.^2,1));
n = n./[norms;norms];

% Invert so that lens looks upward
x = -points;
y = -z;
n = -n;

end
```

6.1.2 Lens Error - Plano-Convex

Next, we see how to compute the error for all rays from all viewing positions refracting in the lens surface — this is our objective function. We omit `refractRay` and `intersectRay` functions as these should be well understood given the normals generated from `getAsphericalChord`.

```
% Calculate the error for all directions.
%
% Input:
%       eyePos          = Position of viewer (very far away from lens)
%       targetPC        = The Pixel Center x coord on the image plane for corresp. directions
%       X               = Parameters (R,kappa,alpha,F)
%       W               = Width of lens
%       n               = Refractive index of lens
%       NOP             = Number of points on lens to sample
%
% Output:
%       accumError      = Total error
%       allErrors       = Vector of errors per ray
%
function [accumError, allErrors] = lensError( X, eyePositions, targetPC, W, n, NOP )

R = X(1);
Kappa = X(2);
Alpha = X(3:end-1);
targetF = X(end);

% Generate points lying on aspherical lenslet surface
[cX,cY,N] = getAsphericalChord( linspace(-W/2,W/2,NOP), R, Kappa, Alpha );


% Calculate error for each eye pos
for j = 1:size(eyePositions,2)

    allX = [];
    for i=1:numel(cX);
        % For each point on lens, set up ray to eye point
        l_p1 = eyePositions(:,j)';
        l_p2 = [ cX(i), cY(i) ];

        % Refract ray defined by endpoints l_p1, l_p2
        r1 = refractRay( l_p2-l_p1, N(:,i)', 1, n );
        p1 = l_p2;

        % Intersect ray p1+t*r1 with pixel plane
        pInter = intersectRays( p1, r1, [targetPC(j),-targetF], [1,0] );

        % Record intersection
        allX = [allX pInter(1)];
    end

    allErrors(j,:) = allX-targetPC(j);
    accumError += sum( abs(allErrors(j,:)) .^2 );

end
```

6.1.3 Setup and Calling

Finally, we show how to set up the optimization call and set initial parameters given the input constants.

```
% Setup constants
%
fieldOfView = 30; % Degrees
W = 5; % Units depend on pixel width; assume 1.
n = 1.47
nRays = 100;
eyeDepth = 1000;

% Derive looking directions
%
% The eye positions are at a fixed depth, and are based on the field
% of view.
theta = fieldOfView/2;
% Gradient of widest ray in field of view
m = 1 / tand(theta);

% Total horizontal range of all eye positions
eyeWidth = 2*(eyeDepth/m);
eyePositions = [linspace(-eyeWidth/2, eyeWidth/2, W), ones(1,W)*eyeDepth];

% Linearly interpolate pixel positions,
% assuming pixel width is 1.
%(Sign must be invese to eye locations due to refraction)
targetPC = linspace(W/2 - 1/2, -W/2 + 1/2, W);

% Objective function
%
objFunc = @(x)lensError( x, eyePositions, targetPC, W, n, nRays );

F  = W / ( 2 * tand( fieldOfView / 2 ) );

% Distortion parameters (unused)
% Increase to use alpha params.
dParams = 0;

% Initial optimization parameters
% R, kappa, alpha, F
x0 = [-(1-n)/n*f, -(1/n)^2, zeros(1,dParams), F];

% Call optimization routine
[result, error] = fminsearch( objFunc, x0, ...
                    optimset('TolX', 1e-8, 'MaxFunEvals', 10000 * length(x0) ) );

% Collect result
R = result(1);
kappa = result(2);
alpha = result(3:end-1);
F = result(end);
```

6.2 Discrete Light Field Optimization

6.2.1 Setup

First, we show the setup of the discrete optimization and the initialization of table T.

```
% Compute an adaptive sampling over an epipolar image.
%
%   Input:
%           epiSlice    = Raster EPI representing a 'scanline slice' of a light field.
%           lensWidths  = Vector of lens sizes in pixel units, e.g., [1:20]
%
%   Output:
%           minError    = The computed minimum error from applying...
%           lensConfig  = The optimal lens set configuration.
%
function [minError,lensConfig] = disOpt( epiSlice, lensWidths )
global indexTable errorTable widthsTable;

% Table T.
% Global 'best lens combination' tables.
indexTable = [];
errorTable = [];
widthsTable = [];

% Extent of a pixel in the output is assumed
% to be equal to one column in epiSlice.
pixelArea = size(epiSlice,1);

% 'Height' of each pixel in the raster (its u extent)
pixelHeights = ones( length(lensWidths), 1 ) ./ pixelArea;

% Subset of epi slice we are considering
epiSubset = size(epiSlice,2);

[minError,lensConfig] = discreteError( epiSlice, epiSubset, lensWidths, pixelHeights, 0 );

end
```

6.2.2 Recursive Function

Next, we show the function which recurses through the possible lens set configurations and adds to T.

```
% Compute an adaptive sampling over an epipolar image.
%
%   Input:
%           epiSlice    = Raster EPI.
%           epiSubset   = The width of the subset of epiSlice that we are concerned with.
%           lensWidths  = Vector of lens sizes in pixel units.
%           pixelHeights= Vector of heights of output pixels.
%           depth       = Depth of recursion - for debug.
%
%   Output:
%           minError    = The computed minimum error.
%           lensConfig  = The optimal lens set
%
function [minError,lensConfig] = discreteError( epiSlice, epiSubset, lensWidths, ...
                                                pixelHeights, depth )
global indexTable errorTable widthsTable;

% Collect an error for each width
widthErrors = zeros( length(lensWidths), 1 );
widthPatterns = cell( length(lensWidths), 1 );

for i=1:length(lensWidths)

    lensWidth = lensWidths(i);
    pixelHeight = pixelHeights(i);

    % First, get remaining subset width after placing
    % this lens at right-most part of epiSlice
    upToWidth = epiSubset - lensWidth;

    % If this width is negative, i.e., if the lens width is larger,
    % then this lens cannot fit and we assign a large error
    if upToWidth < 0
        widthErrors(i) = Inf;
        widthPatterns(i) = {-1};
    % If the lens width equals the remaining subset width
    elseif upToWidth == 0
        % This is a base case.
        % Compute the error directly.
        %
        widthErrors(i) = applyLens( epiSlice, lensWidth, upToWidth, pixelHeight );
        widthPatterns(i) = {lensWidth};

        % It is possible for a length of 'lensWidth' to already
        % exist from a deeper level plus a lens (e.g., 10 will
        % already exist from 5 + 5.
        index = indexTable == lensWidth;
        if sum( index ) == 0
            % Doesn't exist in the table; add
            indexTable = [indexTable lensWidth];
            errorTable = [errorTable widthErrors(i)];
            widthsTable = [widthsTable widthPatterns(i)];
        else
            % Replace existing value if error is less.
            errors = errorTable;
            errors( ¬index ) = Inf;
            [minE,ind] = min(errors);

            if widthErrors(i) < minE
                errorTable( ind ) = widthErrors(i);
                widthsTable( ind ) = widthPatterns(i);
            end
        end
    % Else
    else
```

```
        % Look up this width in T. If empty, recurse.
        index = indexTable == upToWidth;
        % If there's no entry in errorTable for this width
        if sum( index ) == 0
            % We need to recurse and find the error
            [upToError,widths] = discreteError( epislice, upToWidth, lensWidths, ...
                                                pixelHeights, depth+1 );
        else
            errors = errorTable;
            errors( ¬index ) = Inf;
            [¬,ind] = min(errors);
            upToError = errorTable( ind );
            widths = widthsTable( ind );
        end

        % The combined error for this lensWidth is equal to the
        % upToError plus the lens error placed at the end.
        lensError = applyLens( epislice, lensWidth, upToWidth, pixelHeight );

        % Set error and width pattern that achieves this error.
        widthErrors(i) = upToError + lensError;
        widthPatterns(i) = {[widths{:} lensWidth]};

        % Add this entry to the index table ONLY if it hasn't been
        % added before OR if it is the minimum error
        % (and if so, replace the existing minimum error)
        newIndex = upToWidth+lensWidth;
        index = indexTable == newIndex;
        if sum( index ) == 0
            % Doesn't exist in the table; add
            indexTable = [indexTable newIndex];
            errorTable = [errorTable widthErrors(i)];
            widthsTable = [widthsTable widthPatterns(i)];
        else
            % Replace existing value if error is less.
            errors = errorTable;
            errors( ¬index ) = Inf;
            [minE,ind] = min(errors);

            if widthErrors(i) < minE
                errorTable( ind ) = widthErrors(i);
                widthsTable( ind ) = widthPatterns(i);
            end
        end
    end
end

% Find minimum error and return best pattern of widths.
[minError,index] = min( widthErrors );
lensConfig = widthPatterns(index);

end
```

6.2.3 Applying Lenses

Finally, we show the function to apply a lens and computes the error per output pixel.

```
% Compute an adaptive sampling over an epipolar image.
%
%   Input:
%           epislice    = Raster EPI.
%           episubset   = The width of the subset of epislice.
%           lensWidths  = Vector of lens sizes in pixel units.
%           pixelHeights= Vector of heights of output pixels.
%
%   Output:
%           allErrors   = The sum of errors for each output pixel
%           allMeans    = The colors of the output pixels
%
function [allErrors,allMeans] = applyLens( epislice, numPixels, upToWidth, lensWidth, pixelHeight )

% Get errors for each pixel in the output
allPixelErrors = zeros( numPixels, 1 );
allPixelMeans = zeros( numPixels, 3 );

for iPixel = 1:numPixels
    % Compute pixel bounds
    xMin = upToWidth + 1;
    xMax = xMin + lensWidth;
    % Sampling inaccuracy here in rounding.
    yMin = round( (iPixel-1)*pixelHeight + 1 );
    yMax = yMin + pixelHeight;

    % Sample from epislice
    colors = epislice( yMin:yMax, xMin:xMax, : );
    [m n o] = size(colors);
    % Reshape for easy comparison
    cv = reshape( colors, [m*n o] );

    % Average these colors.
    mc = mean( cv, 1 );
    allPixelMeans( iPixel, : ) = mc;

    % Compute squared distance for each color from the mean.
    % Easiest way to do this is to linearize colors.
    sub = repmat( mc, m*n, 1 ) - cv;
    allPixelErrors(iPixel) = sum(sum(abs(sub).^2)) / (m*n);
end

allErrors = sum( allPixelErrors );
allMeans = {allPixelMeans};

end
```

References

CONICS, AND ABERRATIONS. Telescope optics. http://www.telescope-optics.net/conics_and_aberrations.htm. Accessed: 2013-04-07.

KWEON, G.-I., AND KIM, C.-H. 2007. Aspherical lens design by using a numerical analysis. *Journal of the Korean Physical Society 51*, 1, 93–103.

PRUSS, C., GARBUSI, E., AND OSTEN, W. 2008. Testing aspheres. *Opt. Photon. News 19*, 4 (Apr), 24–29.

WIKIPEDIA. Aspheric lens. http://en.wikipedia.org/wiki/Aspheric_lens. Accessed: 2013-04-07.

WIKIPEDIA. Lens (optics), subsection 'lensmaker's equation'. http://en.wikipedia.org/wiki/Lens_(optics)#Lensmaker.27s_equation. Accessed: 2013-04-07.