EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (*2005*), *Number 3*

Pinchmaps: textures with customizable discontinuities

Marco Tarini[1] and Paolo Cignoni[2]

[1] DICOM - Università degli Studi dell'Insubria, Varese, Italy [2] Visual Computing Group, ISTI - CNR, Pisa, Italy

Abstract

We introduce a new texture representation that combines standard sampling, to be bilinearly interpolated in smoothly varying regions, with customizable discontinuities, to model sharp boundaries between these regions. The structure consists of a standard signal texture, plus a second texture we call pinchmap, which encodes discontinuities along generally curved lines; at rendering time the fragment processor efficiently decodes this structure with a single access to each texture. We also present a fully automatic way to compute a pinchmap and signal texture pair, starting from an original high resolution image. The final result on the screen is a comparable visual quality for a fraction of the texture storage and with a negligible impact on performance.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Comp. Graph.]: Line and Curve Generation

1. Introduction

In general terms, a 2D texture T_s stores a signal function s (e.g. color, normal, alpha value, or other attributes) that has to be applied over a 2D surface. It consists of a regular set of samples (texels) of that signal, that are typically interpolated at rendering time. Naturally, the quality of the results is strictly dependent on the resolution of the texture. High resolution textures are so determinant to achieve better visual result that texture memory is always a resource in short supply, notwithstanding the continuous increase of its availability. Hence the need to increase perceived texture quality by other means than just increasing the number of texels.

One promising direction is to resort to mixed 2D image representations. Combining the infinite precision of vectorial elements (e.g. lines or sharp boundaries) with the flexibility of sampled texels potentially leads to tremendous decrease of texture memory usage for a comparable visual quality.

However, to be useful in most applications, such a scheme must be efficiently interpreted in graphic hardware. Recently we witnessed to important advancements, but currently there is no solution that is really feasible to run on commodity hardware at an acceptable price in terms of consumed resources. In this paper we present such a solution.

After a focused, brief analysis of related work, we devote the next four sections to the description a new texture representation (showing the underlying concept, the structure itself, its actual implementation on graphic hardware, and additional effects that can be added in sections 3, 4, 5 and 6 respectively). Then in Sec. 6 we sketch a fully automatic method to construct an instance of that representation.

1.1. Previous Work

In this section we will address only the few previous research results that, to our knowledge, most closely share our objectives: to embed discontinuities into textures to improve their visual quality - especially when magnified- while keeping memory usage low. We refer the reader to the basic and advanced literature for other, conceptually related but technically distant problems, like those involving (as for texture mapping) on-the-fly texture synthesis, procedural textures, texture compression, or (as for image processing) automatic feature detection, image segmentation, or super-resolution.

The approaches presented in [TC04, RBW04, Sen04], and ours too, all share the idea of adopting an image (or texture) representation capable of encoding and displaying features (sharp discontinuities) over images that would otherwise describe smoothly varying values. Another shared characteristic is that the extra information is distributed across the image and stored over a regular grid (whose element are called "Bixels", or "feature-based-texture" pixels, or "silmap" texel, or the "pinchmap" texels that we are about to propose), so that only a limited number of accesses near

the current region will be necessary to locally interpret the image (a necessity if the algorithm is to be implemented in the GPU, where the number of per fragment accesses to texture is severely limited).

In the schema proposed in [TC04], pixels are enhanced to embed sharp boundaries (so becoming "bixels"), and pixel values are not interpolated across such boundaries. Boundaries are defined with linear or quadratic formulas, so they can be curved. This work is not designed in its details to be implemented in a programmable fragment shader. Its general structure would probably allow for such an adaptation, but this has not been investigated.

In [RBW04] a similar schema is proposed. Boundaries are defined as a set of splines. This results in a very expressive representation, but also leads to very complex worst cases that would rule out an implementation on programmable graphic hardware. Authors suggest that simple segments should substitute splines in that scenario, which probably would lead to a solution similar to [Sen04].

[Sen04] (a derivative work of [SCH03]) represents a breakthrough because for the first time this sort of algorithm is really implemented and tested on a programmable fragment shader. To achieve this, complexity of boundaries is kept to a minimum (straight segments). Results are impressive. However, the cost of the technique turns out to be prohibitive for most applications, requiring a total of eight texture accesses per fragment (five to a texture encoding discontinuities, three to the final signal texture) only to obtain a single texture value (the equivalent of a single texture fetch for a standard texture). This figure can possibly be reduced by some form of texel packing or other similar optimizations, but probably not drastically.

This is a consequence of the basic approach shared by all of the above proposals: in order to implement boundaries, final texture values are computed by averaging, near such boundaries, not the usual four but three, two or a single texel value. Unreachable texture values are weighted by zero, and the remaining weights are re-normalized. This makes the computation of the final texel value very heavy in terms of number of accesses. Also, it creates many different cases (even for very simple boundary primitives), that are difficult to dealt with in the fragment shader (extremely ill-equipped for densely branched code).

Additionally, the above approach has an important shortfall in terms of quality, because fewer than four texels are interpolated near boundaries. This is why the signal gradient in directions perpendicular to the boundaries is void; worse still, at corners, where only a single texel is "averaged", the color is bound to be constant.

Another common trait is that discontinuities are always defined piecewise, region by region, and within each region, in a way independent from neighbors. As we will discuss later, this ultimately results in the necessity to perform more accesses to the discontinuity encoding texture.

We approach the problem from a totally different direction, leading to a solution that uses only a single extra texture access (other than the one to the final texture), simplifies the per-fragment computations, improves the visual quality, and naturally supports a number of additional features - including curved boundaries, inexpensive anti-aliasing, smoothly starting discontinuities, optional solid lines, and others.

2. Preliminaries

Assume the texture signal s is defined over a 2D squared domain $\mathbb{T}^2 = [0,N) \times [0,N)$ for some $N = 2^n, n \in \mathbb{N}$. Texture T_s consists of texels located at each integer position in $\mathbb{T}^2$.

The process of scan converting a texture polygon on the screen will produce fragments with a corresponding region of $\mathbb{T}^2$. When that region covers multiple samples of T_s, the problem to combine them into a single value for that fragment can be efficiently solved by various forms of pre-filtering (MIP-mapping). When, on the contrary, the produced fragment corresponds to a region of $\mathbb{T}^2$ with an area smaller than one, then a magnifying filter f_M is needed. The function f_M, defined over $\mathbb{T}^2$, returns for any given position a value that is some combination of the samples of T_s.

Simply fetching the closest texel of T_s leads to severe aliasing artifacts. A common solution to alleviate them is to use:

$$f_M(u,v) = f_b(u,v) \quad (u,v) \in \mathbb{T}^2$$

where f_b is bilinear (first order) interpolation between the four closest samples of T_s. This solution is so widely applied that it is hard-wired on any modern GPU. Bilinear interpolation works well when the signal s to be represented is smooth; on the contrary when s presents 0-order discontinuities it leads to exceedingly blurred visual results.

The bilinear interpolation f_b is a continuous function defined piecewise as follows: $\mathbb{T}^2$ is implicitly subdivided into $N \times N$ unit squares, and inside each square f_b is defined as the 1st order bilinear interpolation of the four texels at the corner of that square. We will refer to these squares as *fexel*s (from "four texels"). Each fexel has four corner texels, and adjacent fexels share two corner texels (note that f_b is C^∞ inside fexels, but only C^0 across fexels).

3. Main Idea

Wherever the signal to be represented presents sharp (0-order) discontinuities, there are some fexels containing an unwanted smooth transition between the texels sampled on either side of that discontinuity. We call those fexels *undesired* fexels. Fexels that are not undesired are *visible*.

The idea is to perturb the locations at which bilinear interpolation is computed so that, intuitively speaking, undesired

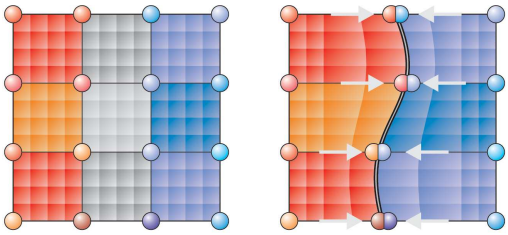

Figure 1: *The concept behind a pinching operation. On the left: a* 4×4 *closeup of a standard texture (in this case, a color texture). The 16 texels are shown as colored spheres. A* fexel *is the space where 4 texels are bilinear interpolated. Here fexels are separated by a solid line and shown with a superimposed regular grid, to illustrate how the fexel space is warped during the pinching. Fexels that interpolate between similar texels are color-coded with shades of blue or red.* Undesired *fexels, i.e. those that interpolate across very different texels, are color-coded with grays. Right, the undesired fexels have been "pinched" creating a sharp discontinuity between the red and the blue regions along a (generally) curved line.*

fexels are shrunk to a line creating a discontinuity in the final result (see Fig. 1); in practice, we ensure undesired fexels will never be accessed. We call this process *pinching*. In other words, we use, as magnification filter f_M, the function:

$$f_M(u,v) = f_b((u,v) + p(u,v)) \quad (u,v) \in \mathbb{T}^2$$

where p is called the *pinch* function. In locations away from discontinuities, p is valued $(0,0)$ and the magnification filter becomes a standard bilinear interpolation. At discontinuities, p will present a matching step-discontinuity.

In all cases, a single bi-linearly interpolated final texture access from the signal texture T_s is performed per fragment. The advantages of this basic choice are manyfold:

- *efficiency*: bi-linearly interpolated texture accesses are hardware optimized, making best use of on-chip texture RAM bandwidth;
- *visual quality*: every final texture value is interpolated between *four* texels;
- *non-branched code*: we always have a single texture texture access to T_s, and that texture access is performed in a non-branched part of the fragment shader;
- *additional effects*: manipulating the above formula, it is easy to obtain several additional effects, including anti-aliasing (Sec. 6.1), smooth start of sharp boundary lines (Sec. 6.2), and others (rest of Sec. 6).

Note that no texel of T_s is wasted, as every texel will still affect an area of $\mathbb{T}^2$; rather we conceal fexels, that is, regions *between* texels.

The vector function p implicitly determines: which fexels are pinched, the pinching directions (which for example is perfectly horizontal in Fig. 1), and the shape of the line to which the pinched fexels will be collapsed.

We encode p in an auxiliary texture T_p, the *pinchmap*, that is pre-computed and paired with the main signal texture T_s, and is loaded on the graphic card at rendering time. To save texture memory T_p should be as compact as possible; to save on-card texture bandwidth, $p(u,v)$ should be computed using the least number of accesses to T_p.

One natural choice would be to define p piecewise, by subdividing $\mathbb{T}^2$ into as many pieces as there are texels in T_p and then separately storing each piece in a corresponding texel of T_p, encoding it with a configuration index and a limited number of parameters. However, in order to enforce continuity of p across adjacent texels, one would need to either perform additional accesses to neighbor texels in T_p, or alternatively to replicate some data from neighbor texels inside each texel of T_p.

To bypass this problem we designed a scheme where p is computed using a single bilinearly interpolated texture access to the pinchmap T_p, and $p(u,v)$ is computed starting from the four recovered channel values (again, bilinearly interpolated texture accesses are highly optimized and their cost in performance and bandwidth is very similar to that of a direct texture access).

This scheme, described in Section 4, allows for curved discontinuity lines. It also lets the resolution of the pinchmap T_p and of the signal texture T_s to be chosen somewhat independently (see Section 6.4).

The resulting per-fragment algorithm to process a fragment with associated texture coordinates (u,v) is conceptually as follows (see Section 5.2 for a more detailed description):

1. fetch texel t_p at pos. (u,v) from pinchmap T_p;
2. compute pinch-function $p(u,v)$ using t_p (and u and v);
3. fetch final texel t_s at pos. $(u,v) + p(u,v)$ from signal texture T_s;
4. process t_s normally.

Both texture accesses are bi-linearly interpolated. The last step is the same as any other 2D texture mapping application; for example it can consist of a verbatim copy of t_s to the current pixel (if T_s stores a pre-shaded color), or of a shading of t_s (if T_s stores normal values) and so on.

Pinchmaps and mipmapping: Although the pinchmap perturbation is designed for magnification filters, it produces final texture coordinates that are valid for all MIP-map levels. This means that the same algorithm sketched above works regardless of the magnification level (differently from other approaches, we do not need to identify it in the fragment shader). MIP-map levels can be computed for T_s normally, and accessed in step 3 with standard trilinear interpolation. Clearly, MIP-mapping must be turned off in the access of step 1.

4. Pinch functions

In this section we detail how a proper pinch function is defined over $\mathbb{T}^2$ to perturb texture positions in order to "pinch away" undesired fexels. For illustration purposes, we first tackle a $1D$ case where a single fexel is pinched.

4.1. One dimensional case

In a one-dimension analogue, a fexel is the segment between two consecutive texels of a one-dimensional array of samples. Fexels delimited by texels that should not be interpolated are *undesired*, just like before. Note that two consecutive fexels cannot both be undesired, otherwise the shared texel would not be part of any visible fexel.

An undesired fexel will be collapsed into a point γ located somewhere inside it, by expanding the two adjacent fexels on its left and right side (which are both visible ones). In particular, we choose to expand only the closest halves of the two adjacent fexels. In this way, the two further halves are left untouched, and can be expanded over the possibly undesired fexel on the opposite side, if needed.

We define a local one-dimensional pinch function $p_\gamma^{1D} : \mathbb{R} \to [-1..+1]$ that is parameterized with γ and is used to perturb the texture location in order to pinch away a single undesired fexel. The final texture coordinate for a texel with initial texture coordinate k will be $k + p_\gamma^{1D}(k)$.

In order to avoid repetition artifacts, we must make sure that $k + p_\gamma^{1D}(k)$ is strictly monotonically increasing with k. This is equivalent to the constraint $\partial p_\gamma^{1D}(k)/\partial k > -1$.

Let us describe p_γ^{1D} in a reference system where the origin is in the middle of the 1D fexel to be pinched away (see Fig. 2). Consequently the locations of the two texel delimiting the fexel will be at -0.5 and at 0.5. The function p_γ^{1D} is non-zero only inside the interval $(-1..+1)$ (as we want to affect only the two halves of the adjacent fexels), and moreover it must be zero in ± 1 to ensure continuity. The parameter γ, which is the position where the undesired fexel is collapsed, typically ranges in $[-0.5..+0.5]$.

Since we want to "pinch" both delimiter texels into position γ, we need that $p(\gamma^-) = -h - \gamma$ and $p(\gamma^+) = h - \gamma$, where h is the value 0.5 (we are using h as a parameter because later we will need to change its value, see Sec. 6.4). By interpolating linearly between these fixed values at -1, $+1$, γ^- and γ^+ we get

$$p_\gamma^{1D}(k) = \begin{cases} 0 & k \le -1 \\ -\frac{(1+k)(h+\gamma)}{\gamma+1} & -1 < k \le \gamma \\ -\frac{(1-k)(h-\gamma)}{\gamma-1} & \gamma < k \le 1 \\ 0 & k > 1 \end{cases} \qquad (1)$$

Note that the constraint on the derivative of p_γ^{1D} is satisfied for any $\gamma \in [-0.5..+0.5]$, or even $(-1..+1)$.

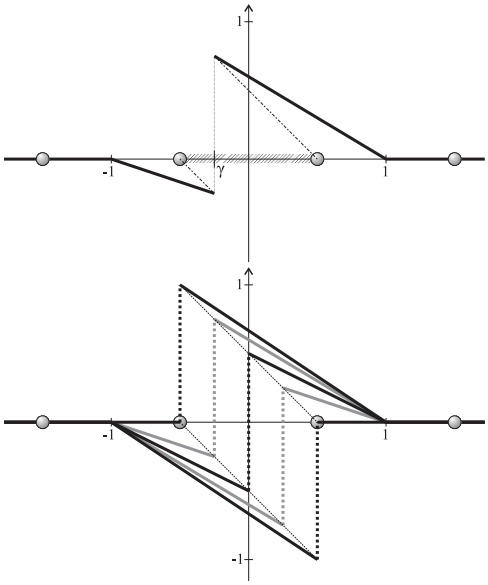

Figure 2: *Above: the one dimensional pinch function p_γ^{1D} for a given γ. The horizontal axis is centered in the middle of the fexel to be pinched (hatched with diagonal lines). Texels are shown as gray balls. Below: the function for $\gamma = -0.5, -0.25, 0, +0.25, +0.5$.*

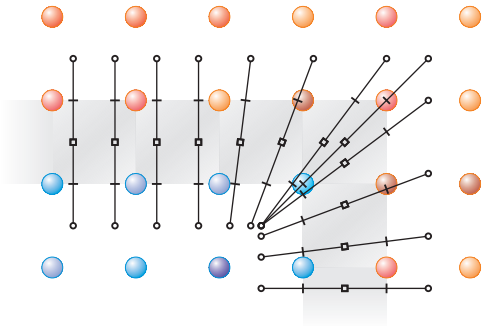

Figure 3: *An (arbitrarily chosen) discrete subset of the continuous set of segments along which the function p_γ^{1D} is to be applied. In each shown segment, the midpoint ($k = 0$) is identified by a square, the extreme points ($k = \pm 1$) by a circle, and the points at position $k = \pm 0.5$ by a small crossing line. Undesired fexels are grayed (note that they correspond to the points at positions $-0.5 < k < +0.5$). For clarity we do not show the pinching positions γ (γ varying across segments) which form a discontinuity line.*

In summary, the function p_γ^{1D} is such that $k + p_\gamma^{1D}(k)$ is never in $(-0.5..+0.5)$, but will assume any other value for some k. The interval $(-0.5..+0.5)$, which correspond to an undesired fexel, is effectively "pinched away".

4.2. Two dimensions

Getting back to the two dimensional case, we want undesired fexels to be shrunk to a line, their area covered by stretching neighbor visible fexels over them. Similarly to the 1D case, we want only the closer half of the visible neighbors fexel to be stretched, so that the other half remain unaffected and can, if needed, take part of an unrelated pinching operation of the other side (more precisely, subdividing every visible fexels in four squared subregion, only those adjacent, even diagonally, to an undesired fexel will be expanded).

The extension to 2D however is not straightforward, as the pinching operation as we defined in Sec. 4.1 is fundamentally a 1D operation. Our solution is perform that operation along proper pinching direction over $\mathbb{T}^2$.

More specifically, we will cover the areas of $\mathbb{T}^2$ affected by pinch operations with a continuous set of *pinch segments*, and then we apply p_γ^{1D} along each segment. Each pinch segment is identified by its midpoint (m_u, m_u) and direction (d_u, d_v), with $\|(d_u, d_v)\|_\infty = 1$, and it is parameterized as:

$$(m_u, m_v) + k \cdot (d_u, d_v) \quad k \in [-1, +1]$$

Finally each segment has a parameter γ that determines where, over that segment, the discontinuity is to appear. Over each segment, the pinch function p is defined as:

$$p((m_u, m_v) + k \cdot (d_u, d_v)) = p_\gamma^{1D}(k) \cdot (d_u, d_v) \qquad (2)$$

Since γ will vary continuously across the segments, the resulting discontinuity line, defined by all the points at position $k = \gamma$ of every segment in the set, is in general a curved line.

4.3. Disposition of pinch segments

For equation (2) to be defined, every point in T_s affected by a pinch operation must belong to a single segment (with the exception that a point can be either end of any number of segments, because the $p_\gamma^{1D}(\pm 1) = 0$ for every γ).

To get the desired effect *all and only* the points inside undesired fexels must be at position $k \in [-0.5, +0.5]$ of their segment (so that these fexel will be "pinched away", according to equation (1)).

It follows that, in our schema, the set of mid-points of the segments ($k = 0$) forms a poly-line passing through the center of every two adjacent undesired fexels; for $k = \pm 0.5$ the poly-line corresponds to a boundary between visible and undesired fexels; for $k = \pm 1$ the poly-line is the line connecting the center of the visible fexels surrounding the undesired ones (see Fig. 3).

4.4. Constraints on fexel configuration

Just as in one dimension we could not have two consecutive pinched fexels, we have similar consistency constrains in two dimensions.

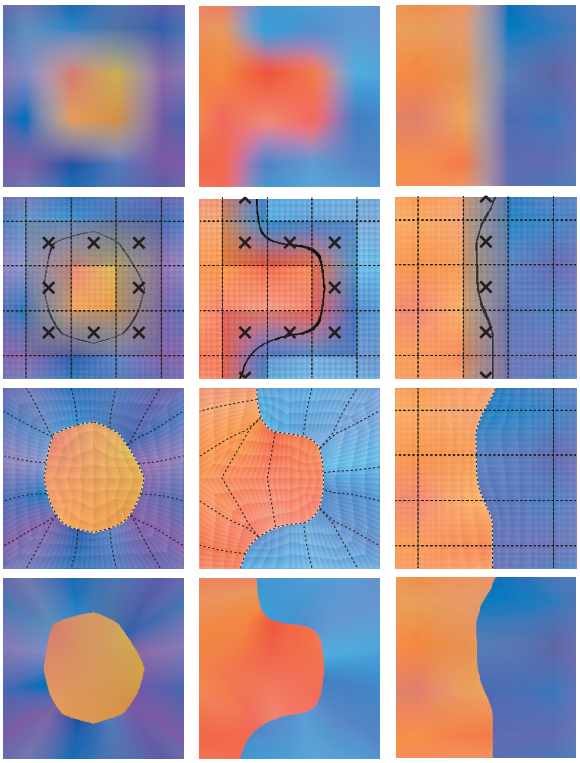

Figure 4: *Examples of pinching operations, over a minimal 4×4 signal texture. Each column shows a different combination. Top row: standard bilinear interpolation, with no pinching. Second row: fexels are shown before pinching. A 8×8 subgrid is superimposed, and undesired fexels are darkened and identified with a black cross. The discontinuity line (resulting from randomly chosen values of γ) is also shown in black. Third row: undesired fexels are collapsed to that line, and neighbors fexels are expanded over them. Bottom row: final result (actual snapshot).*

First of all, the combination where a group of 2×2 adjacent fexels are all undesired does not make sense, as the signal texel shared by the four fexels would not belong to any visible fexel.

Another combination to be ruled out is the one where a texel is at the same time a corner of two diagonally opposite undesired fexels, and two diagonally opposite visible fexels. If that was the case, the texel in question would be involved in two different incompatible pinch operations pushing it in opposite directions.

Any other combination is valid (see Fig. 4 for examples).

5. Encoding and decoding pinch functions

In this section we detail how a proper pinch function can be encoded in a pinchmap texture T_s, and, during fragment processing, recovered from it and then applied.

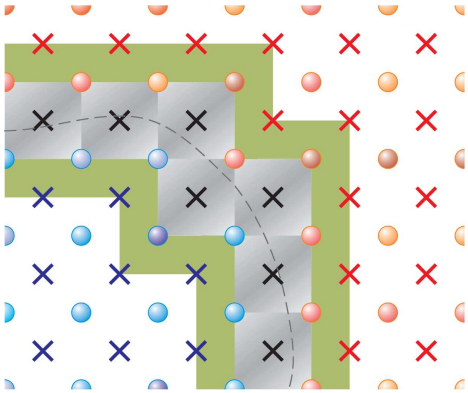

Figure 5: *A detail of a signal- (in this case, color-) texture T_s with its pinchmap texture T_p superimposed. Texels of T_s are shown as colored balls, while texels of T_p are shown as crosses. There is one texels of T_p for each fexel of T_s, so the two grids are displaced by half a texel size. Undesired fexels of T_s (grayed areas) correspond to active texels of T_p, shown as black crosses, where the k channel is zero. The other texels of T_p have a k channel with a value of either minus one (red crosses) or one (blue crosses). The pinch-function is nonzero only in zones colored green or gray, so the pinch affects only these regions. In particular, the green region will expand over the gray region, covering it (in this way the gray region will be pinched into a discontinuity line - here dotted).*

5.1. Pinchmap structure

In order to apply equation (2) to the texture position (u,v) of each fragment, we need the values used in that formula: k (parametric position of (u,v) inside its pinch segment), d_u, d_v (orientation of that segment), and γ (pinch location over that segment). We store appropriate values $(d_u^T, d_v^T, k^T, \gamma^T)$ in the 4-channel texels of the pinchmap T_p so that used values d_u, d_v, k and γ can be recovered adjusting the tuple $(d_u^B, d_v^B, k^B, \gamma^B)$ returned by the single bilinear interpolated access to T_p.

Here it will be useful refer both to fexels of the signal-texture T_s and to fexels of pinchmap T_p. When we refer to the latter will be always specify *pinchmap* fexel.

All the stored (and read) values range in $[-1,+1]$. Texture values natively range in $[0,1]$ so a remapping is needed. Care must be taken to be able to represent the value 0 precisely. For example, if the textures have 8-bit per channel, the range $[-1,+1]$ must be remapped over $[0..254]$, not $[0..255]$, so that the value 0 is represented by the value 127.

In our schema every pinchmap texel of T_p is in the center of a fexel of the signal texture T_s. In other words, the two textures are reciprocally displaced by a constant $h = 0.5$ in both directions (see Fig. 5). This way it is easy to obtain the 1-order discontinuities that the channel k must present at $k = \pm 1$ (compare Fig. 3), as they naturally appear at the borders between pinchmap fexels. We refer to the pinchmap texels corresponding to undesired fexels of T_s (that will be pinched) with the term *active* texels.

Active texels, where k^T is set to 0, form lines of adjacent texels that breaks the pinchmap T_p into zones of non-active texel, inside which k^T is constantly set to $+1$ or -1. No pinching will occur internally to a zone (as $p_\gamma^{1D}(k)$ is 0 for $k = \pm 1$). This setup may at first seem to severely limit the expressive power of a pinchmap, allowing only for closed discontinuity lines. However this problem is bypassed as pinching can be locally disabled even in active texels (by zeroing its d_u^T and d_v^T channels, see Section 6.3 later).

Active texels can be subdivided into *straights* (horizontal or vertical), and *turns*. Active texels are boundaries between $k^T = +1$ and $k^T = -1$ non-active ones, and d_u^T and d_v^T are set to form a vector pointing toward the $k^T = -1$ region; for example in the *turn* texels both d_u^T and d_v^T are set to ± 1, in horizontal *straight* texels d_u^T is zeroed and d_v^T is set to ± 1, and in vertical ones d_v^T is zeroed and d_v^T is set to ± 1. Cross active texel, where two discontinuity lines cross, must privilege one of them; in the final result the other one will unfortunately show blurred in the close proximity of the crossing.

All non active texels have (k^T, d_u^T, d_v^T) always respectively set to $(\pm 1, 0, 0)$, so that it is trivial to deal with the case when a non-active pinchmap texel is adjacent to multiple active texels.

Each channel of the pinchmap require some implementation consideration.

Channel k: position inside pinching segment.

We need the value of k to be linear along all the segments (because it needs to be equal to ± 0.5 corresponding to the border between undesired and visible fexels, see Sec. 4.2). Even if the value of k^T is set to k in all pinchmap texels, the value of k^B is in some case not linear. This happens in a pinchmap fexel that has exactly one or exactly three active pinchmap texels as its corners (see Fig. 6).

Luckily it is easy to correct the interpolated value k^B and recover linear k, knowing the current texture position (u,v) over T_p (in texels) and the signs of d_u^B, d_v^B. If one of d_u^B, d_v^B is zero, then simply $k \leftarrow k^B$. Else, k can only be one of two alternatives

$$\text{sign}(k^B)/2 - \text{sign}(d_u^B) \cdot (\text{frac}(u) - 0.5)$$
$$\text{sign}(k^B)/2 - \text{sign}(d_v^B) \cdot (\text{frac}(v) - 0.5)$$

where frac returns the fractional part of its argument, i.e. $\text{frac}(x) = x - \lfloor x \rfloor$.

To know which, we check: if one of them is equal (within a tolerance) to k^B, than $k \leftarrow k^B$ again. Else, if they are both smaller than k^B, then we assign to k^B the smaller of the two values; else the larger. The proof, based on exhaustive analysis of possible cases, is omitted for space reasons. While this

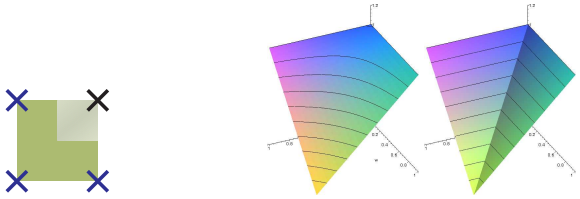

Figure 6: *Left: a single pinchmap fexel from the pinchmap shown in Fig. 5. This particular fexel has the k^T channel set to 0 in one (active) corner texel (in black), and the other three corner texels with a k set to $+1$ (in blue). This is one case where the result of the bilinear interpolation of the k^T channel (plotted in the middle) is not equal to the signal k that we need (plotted in the right). The latter is linear along the segments defined in Sec. 4.2, the former is not.*

procedure may seem at first complicated, the fragment program that implements it uses only relatively few operations (mostly conditional assignments).

Channels d_u and d_v: segment orientation. In our schema bilinear interpolation of these channels automatically ensures that the segment direction changes smoothly across segments (and that it is constant along each segment). However, the length of the vector (d_u^B, d_v^B) will be shorter, because these values are interpolated with the values $(0,0)$ that we defined at inactive texels.

The original (d_u, d_v) vector can be recovered from (d_u^B, d_v^B) simply normalizing the latter (with respect to the infinity norm, that is, dividing it by $\max(|d_u^B|, |d_v^B|)$). By an exhaustive analysis of all cases, we can also predict $|d_u^B, d_v^B|_\infty$ to be $(1-|k^T|)$. We prefer the latter formulation for reasons that will be explained in Section 6.2.

In a minority of cases (in the internal parts of L turns) this is an slight underestimation, but this is acceptable as it results ultimately in an overestimation of (d_u, d_v) and therefore in the pinching of a small part of a visible signal fexel. However, for more accurate results we can also detect these cases and correct them (formulas omitted for reason of space).

Channel γ: position of the discontinuity.

The channel γ is the sole responsible to determine the location of discontinuity line within the zones affected by the pinch (colored in green and gray in Fig. 5). The γ^T can be set arbitrarily in each texel of T_p, different choices leading to different discontinuity lines but to equally coherent pinch functions, in which the same undesired fexel are pinched away. For example, in Fig. 4 we have assigned random values to γ. During the construction of the pinchmap, the task will be to find proper values for γ^T so that given shapes for the discontinuity line are reproduced (see Section 7 for an illustration of how to do this automatically).

To be more precise, discontinuity lines are defined by points where $k = \gamma$ (because of the structure of the pinch function p_γ^{1D}, see Fig. 2). In our experience we found that, if we directly use as γ the interpolated value γ^B, then the discontinuity line would tend to show a steep bend corresponding to the diagonal of pinchmap fexel with a of *turn* active texels at a corner, because we are comparing a bilinearly interpolated value with a linear value like that plotted in Fig. 6, right. In most cases these bends at constrained locations worsen the quality of the final result, and they are very difficult to get rid of manipulating γ^T values.

To avoid this problem it is enough to use $\gamma \leftarrow \gamma^B + (k - k^B)$. This way the equality $k = \gamma$ becomes equivalent to the equality $k^B = \gamma^B$, leading to a more useful set of discontinuity lines that are producible setting γ^T values.

Finally it should be noted that γ is allowed to vary along a segment as well as across segments. However this does not affect the location of the discontinuity ($k = \gamma$), nor the final texture positions at either side of that discontinuity, and does not create any glitch or repetition artefact.

5.2. Fragment Program

The complete fragment program, for a fragment with initial texture coordinates (u, v), is therefore:

1. Fetch values (r, g, b, α) from pinchmap T_p at $(u+h, v+h)$ (with bilinear interpolation on);
2. Remap to $[-1, +1]$: $(d_u^B, d_v^B, k^B, \gamma^B) \leftarrow \mathrm{Map}(r, g, b, \alpha)$;
3. Compute the "rectified" value k form recovered values;
4. Normalize direction: $(d_u, d_v) \leftarrow (d_u^B, d_v^B)/(1-|k^B|)$;
5. Compute $\gamma \leftarrow \gamma^T + (k - k^B)$;
6. Pinch the texture coordinate: $(u', v') \leftarrow (u, v) + p_\gamma^{1D}(k) \cdot (d_u, d_v)$
7. Fetch final signal value from (u', v') (with bilinear interpolation and mip-mapping);
8. Process fetched signal value as usual.

The entire program, including all the extensions that we are going to illustrate in the next section, has been implemented using less than 50 ARB fragment program instructions (the most expensive parts being Steps 3 and 5).

Numerical analysis shows that the algorithm is stable even when the divisor in step 4 is arbitrarily close to zero (in which case, also the final displacement will be close to zero). Of course, care must be taken to avoid a direct division by zero - for example, adding a small constant to the divisor.

6. Additional Features

In this section we show some additional feature that can be added, with a small or null impact on efficiency, to the pinchmap algorithm.

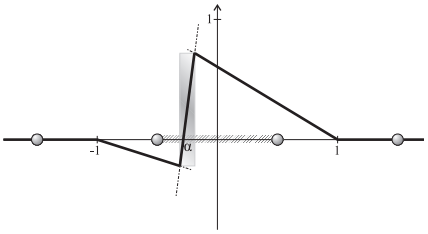

Figure 7: *The plot of the 1 Dimensional Anti-Aliased pinch function $p_\gamma^{AA1D}(k)$, for $\gamma = -0.25$. In the horizontal axis the anti-aliased part i is grayed.*

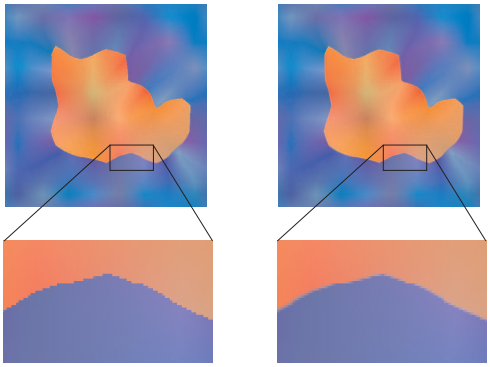

Figure 8: *A quad textured with a 8×8 pinchmap and a rgb-colormap (actual screenshot). Below: a post-processed close-up to show individual screen pixels. Left and right: results with and without anti-aliasing.*

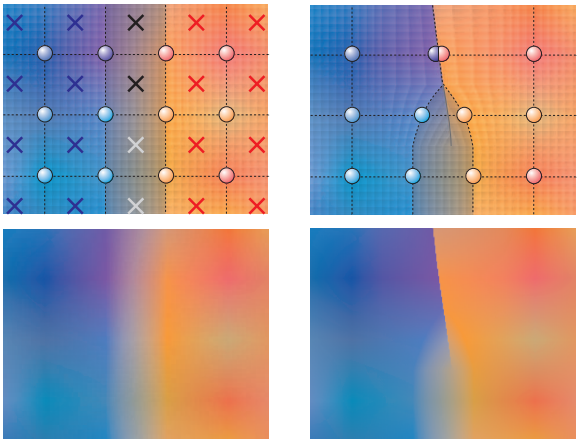

Figure 9: *Pinching can be locally turned off, and discontinuity lines can start smoothly. Top left: by a 4×3 subset of a signal texture T_s (spheres).The white pinch-map active texels (crosses) have a pinch strength p_{str} of zero, the black ones of one. As a result (top right) the top undesired fexel is pinched away, the bottom one is not and the two central ones show a smooth transition between these two states. Bottom: actual screenshots without (left) and with (right) pinching.*

6.1. Anti-aliasing

It is easy to add an anti-aliasing on screen over the 0th order discontinuities that we introduce in the texture. Conceptually, the idea is simply to perform only incomplete pinch operations, so that undesired fexels are shrunk to a very small but positive area rather than to a line. This effectively results in an anti-aliasing, as undesired fexels are exactly the region where the values from the two sides of the 0-order discontinuity are interpolated together.

The only thing that we need to change is the function p_γ^{1D}, which is to be substituted in equation (2) with its anti-aliased version p_γ^{AA1D} defined as:

$$p_\gamma^{AA1D}(k) = \begin{cases} \beta(k-\gamma) & \text{if } (k<\gamma) \otimes (\beta(k-\gamma) < p_\gamma^{1D}(k)) \\ p_\gamma^{1D}(k) & \text{otherwise} \end{cases} \tag{3}$$

where the $\otimes$ symbol denotes the logical exclusive-or, and β, with $\beta \gg 1$, is the anti-aliasing parameter, intuitively denoting the speed of texture coordinate over the undesired fexels. The meaning of the formula is that the value of p_γ^{1D} is overridden in proximity if its discontinuity by a steep, but not instantaneous, connecting function (see Fig. 7).

The anti-aliasing parameter β determines the width of the region where the undesired fexel will be squeezed. It is easy to see that its size will be $1/\beta$ of a texel. Ideally, for a perfect anti-aliasing, this should be equal to one screen pixel. One possibility is to actually use the per-fragment texture-speed to adequately set β for every texel. Alternatively, that can be approximated with a global parameter β set for each textured object.

The function p_γ^{AA1D} is just an approximation of the ideal anti-aliasing function (in fact, contrarily to the ideal case, the anti-aliased regions on the left and on the right of γ are in general different in size). However, the function is very simple to compute and it is good enough for most practical purposes (see for example Fig. 8).

6.2. Smoothly starting discontinuities

An important feature of our system is the ability to have a smooth spatial transition between pinched and not pinched regions. Intuitively the effect is obtained by only "half-pinching" a given undesired fexel, leaving one of its side un-pinched while pinching the other (see Fig. 9).

The concept is straightforward. We use an extra attribute *pinch strength* $p_{str} \in [0,1]$ varying over $\mathbb{T}^2$, to multiply the pinch function p_γ^{1D} in equation 2, which becomes:

$$p((m_u, m_v) + k \cdot (d_u, d_v)) = p_\gamma^{1D}(k) \cdot p_{str} \cdot (d_u, d_v) \tag{4}$$

This is implemented in a simple way, by encoding p_{str} in the (d_u^T, d_v^T) channels of each texels. In active texels we store, as (d_u^T, d_v^T), the values $(p_{str} \cdot d_u, p_{str} \cdot d_v)$. Step 4 of the

Figure 10: *Here, the signal texture is a normal-map encoding a curved surface with a crease. The pinch strength is changed from zero to one over the space of three fexels to make the crease start smoothly. Left: fexels shown with a superimposed* 8×8 *grid, right: final result (screenshot).*

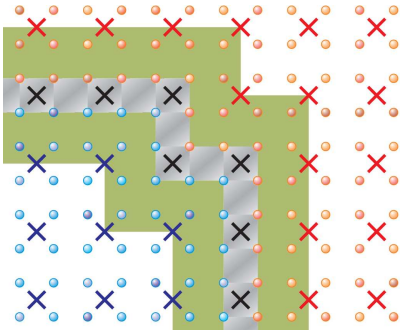

Figure 11: *A detail of a signal-texture* T_s *paired with a pinchmap* T_p *where* T_p *has a resolution that is an half of the one of* T_s*, to show the disposition of the texels of the two textures. For a legend of symbols, see the caption of Fig. 5.*

fragment program in Section 5.2 will now rightly compute $p_{str} \cdot (d_u, d_v)$ instead of (d_u, d_v). Thanks to bilinear interpolation, strength p_{str} will vary smoothly across $\mathbb{T}^2$.

The ability to make boundaries smoothly start over several texels is especially useful when the signal texture T_s represents a normal field, and the 0th order discontinuity lines represents creases - like, for example, in an automobile chassis (see Fig. 10).

6.3. Open boundaries

Another important application of the above is when the pinch strength p_{str} is uniformly set to zero in a area, in which case no pinching at all occurs in that area (see Figures 10 and 9). The ability to locally turn off pinching let us have non closed discontinuity lines across the texture. A non closed discontinuity line is (correctly) bound to start smoothly.

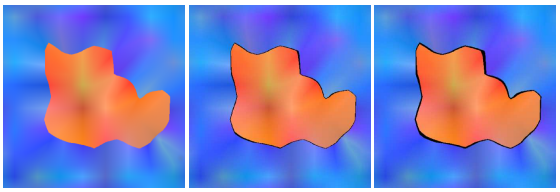

Figure 12: *Actual screenshots featuring the same data as Fig. 8. Left: pinched result, without solid lines. Middle and right: solid black lines of different thickness are added in the way described in Section 6.5.*

6.4. Different res. for the pinchmap and signal textures

Until now we assumed the pinchmap T_p and the signal texture T_s had the same resolution. However, in general, the resolution of the pinchmap can be 2^K (with $K \geq 0$) times smaller than the signal texture. When $K > 0$, the texels of T_s must be grouped in cluster sized $2^K \times 2^K$, that will be placed on one same side of the discontinuity. The displacement of T_p with respect to T_s is, in the general case, $2^{(-K-1)}$ (see Fig. 11).

In the fragment shader, the only difference is that the undesired fexel to be pinched are now smaller with respect to the size of one fexel of T_p. Therefore the function p_γ^{1D} need to be smaller in magnitude. The value of the parameter h (in equation (1) and in Step 1 of the fragment code in Section 5.2) is in the general case $2^{(-K-1)}$ (see Fig. 11).

On the contrary, it is not currently possible in our schema to use a signal textures smaller than the pinchmap.

Within this limit, the two resolutions can be chosen independently. Intuitively, the resolution of the pinchmap determines the complexity of the discontinuities - the higher it is, the more complex and dense the discontinuity lines can be (see Sec. 5.1); the resolution of the signal texture determines instead the possible complexity of the color-map in the smoothly varying zones.

6.5. Solid lines

Another easy-to-obtain effect consists of adding thick lines of a constant color in correspondence of the discontinuities lines, an effect that can be useful, for example, when the signal texture contains color information and we are targeting a non-realistic rendering (see Fig. 12). This is applicable only when all discontinuity lines are closed.

To get this effect, it is sufficient to apply the anti-aliased pinch function p_γ^{AA1D}, as defined in (3), and detect when the 1st of its cases is applied. In this case anti-aliasing would normally take effect. Instead, we simply overwrite the current fragment color with a globally defined line color. The parameter β of p_γ^{AA1D} determines line thickness (which however will not be perfectly constant, due to the variation of the Euclidean length of diagonal segments).

In some application it could be appropriate to store line color and thickness in each vertex of the model, or in an additional texture, in order to make it possible to vary them over the surface.

7. Automatic pinchmap creation tool

So far we described the way pinchmap-enhanced textures work. In this section we detail how one can build, in a fully automatic way, a pinchmap/signal texture pair to represent a given signal s.

7.1. Inputs and outputs

The **input** of this process must be a description of the original signal s, for example a vector-based representation of an image, or a procedurally defined image, or a high-resolution raster image I_{HiRes}. In our implementation we adopted the latter approach because of its better flexibility (if the signal is originally defined in any other way, it can be sampled to form a hi-res raster image with arbitrary precision).

Moreover, we use I_{HiRes} images with a extra alpha (transparency) channel, that is used by the final tool user to specify where the signal discontinuities must appear: the intended meaning of this channel is as follows: wherever two regions of I_{HiRes} are separated in the alpha channel by a steep jump from zero to one or from one to zero, it means a sharp border between the two corresponding texture regions is required. On the contrary, where the alpha channel present no such jump, the signal in the final result will be blended.

For example, the user can provide as input a $rgb\alpha$ color image consisting of a foreground shape (where alpha is set to one), silhouetted against the background (where alpha is set to zero), to obtain a final result where that shape appears separated from the background by a sharp boundary, and the color varies smoothly both internally and externally of the shape (see for example Fig. 13).

Finally, we are assuming that the resolution res_p of the required pinchmap T_p and the resolution res_s of signal texture T_s are also given as input. The two resolutions are supposed to be much lower than the resolution of I_{HiRes}.

The **output** of the process is a pinchmap and a signal-texture at the asked resolutions that can be combined to approximate the signal originally stored in I_{HiRes}: in particular, the discontinuity encoded in the pinchmap will mimic the boundaries discretely specified via the alpha channel; the signal texture will store the signal specified in the other channels of I_{HiRes} (whether they represented color, normal, transparency, etc).

Once built, the two output textures can be stored and, later, used independently from the original image I_{HiRes} that was used to create them.

7.2. Algorithm

The algorithm consists of the following phases:

1. down-sample the alpha channel α of I_{HiRes} into a bit-mask I_{Layer} of resolution res_p;
2. enforce consistency constraints on the bit-mask I_{Layer};
3. build d_u^T d_v^T and k^T channels of T_p from the bitmask;
4. optimize the channel γ^T channels of T_p to match $I_{HiRes}.\alpha$
5. fill T_s combining values of the channel other than α from proper locations of I_{HiRes} (and set the strength of each active texel of T_p);

In the first four steps the pinchmap T_p is constructed.

Step 1 consists of a direct down-sampling of α channel of I_{HiRes}. We build a temporary I_{Layer} bit-mask with a bit for each pinchmap fexel of T_p. All pixels of the corresponding area of I_{HiRes} are checked against the threshold 0.5, and the I_{Layer} bit is snapped to 0 or to 1 according to the majority of the results (we keep track of the magnitude of the rounding for the next step).

In **step 2** we enforce the constrains described in Section 4.4. Problematic areas are located, and then an iterative local search is performed: in each iteration step we find and apply the cheapest move that diminishes the global number of broken constraints. An atomic move consists in assigning a previously mixed 2×2 patch of I_{Layer} to all ones or all zeros (overwriting their previous values), and its cost is computed as the worsening of the total rounding error.

In **step 3** I_{Layer} will be used to identify active and inactive pinchmap texels, and to consequently build the d_u^T d_v^T and k^T channels of each texel of T_p. This is straightforward: each texel of T_p corresponds to the intersection of 4 pixels of I_{Layer}: if the four pixels are uniformly zero or ones, then texel is not active and k^T is assigned to -1 or $+1$ respectively; else, the texel is active and k^T is assigned to zero. Values of d_u^T d_v^T are then assigned as described in Sec. 5.1. The previous step ensured that we do not incur in forbidden configurations.

Step 4 is the core part of the tool, where the γ^T channel of T_p is optimized. Our objective is to assign a value to the γ^T component of each texel that is either active or around an active texel, so that the number of matching pixels of I_{HiRes} is maximized. A pixel is considered to be matching if its location ends up in the right side of the discontinuity, that is when its α value is bigger than 0.5 if and only if in the corresponding point of T_p the interpolated value γ^B is bigger than the interpolated value k^B (see Section 5.1). We do this using a randomized simulated annealing approach.

We subdivide I_{HiRes} in equal sized squared *cells* each corresponding to a fexels of T_p. First, we optimize locally values of γ^T in each texel separately. A local optimization consists of a series of tries: γ^T is increased and decreased by a progressively smaller delta and the new value is accepted only

if it performs better than the old one. In each step the performance is measured checking the number of matching pixels in the four *cells* corresponding to the pinchmap-texels sharing the pinchmap-texel being optimized. This first search ends in a local minima usually very far from the global one.

After this initial optimization, we iteratively choose a random location of T_p, perturb γ^T values in a 2×2 patch of texels around it, and then reoptimize separately each γ^T values in a 4×4 patch of texels around it, like before. After each iteration we accept the combined change only if it performs better, in total, than the previous situation. This effectively moves the solution away from the local minima.

The most expensive part (and the tighter sub-loop) of this process consists the error computation for a given *cell*. That part is highly optimized: for each row of pixels of each *cell* we can pre-compute and store once and for all the number of guessed pixel for each possible integer position just before the intersection, along that line, of the channels γ^B and k^B (before, and after, that position all point of that cells are on the same side of the discontinuity). The global error for a cell is computed as the sum of those values. Of course, the error associated to each *cell*, once computed, is also stored for any later query (changing a γ^T value invalidates this stored value for all the *cells* around it).

In **step 5**, now that the pinchmap T_p is ready, all we have to do is to fill the signal-texture T_s. To do that we first compute the inverse g^{-1} of the function $g(u,v) = (u,v) + p(u,v)$ by brute force: we compute function g for a dense sampling of its argument, storing in each location $g(u,v)$ of a temporary buffer the starting location (u,v). Then to each texel of T_s, corresponding to the position (u,v) on T_p, we assign a signal value that is a combination of the pixels of I_{HiRes} around $g^{-1}(u,v)$.

When we compute this combination, we make sure that we do not mix in T_s pixels of I_{HiRes} that belong to different regions. That is, we must be sure to combine only pixels that have all $\alpha > 0.5$ (if in position (u,v) of T_p where $\gamma > k$) or $\alpha < 0.5$ (otherwise).

In this step we also detect when the input I_{HiRes} image presents values of α around the discontinuity that are far from 0 and 1. In this case we set the strength of the pinchmap-texel to zero.

7.3. Results of the Pinchmap Creation Tool

We tested our method with a variety of input of different sizes and with different parameters, with very good results. Some example is visible in Fig. 13.

The computation times are contained, keeping in mind that this is just a preprocessing phase. The fourth phase, being an iterative search over a very large domain, is the most expensive one and takes proportionally almost all of the time. However, thanks to the optimizations, total times are in the order of a few seconds for a 32×32 pinchmap mimicking an original texture of 1024×1024.

8. Conclusions

We presented a texture representation that uses an auxiliary pinchmap to describe and embed custom discontinuities along generally curved lines over standard bi-linearly interpolated 2D textures. We showed how the obtained results are visually comparable to those usually obtainable only at a cost of a severely larger texture memory usage. For example, the 32×32 pinchmap encoded image, visible in Fig. 13 (second image), shows a far better visual quality than the 128×128 downsampled image (third image), which occupies eight times as much memory. The former also behaves better under extreme magnification.

Our approach is based on two basic ingredients. First, the pinching approach means that we always resort to a single final bi-linearly interpolated in the last texture access, with positive effects both on visual quality and efficiency. Second, the preceding texture access to the auxiliary pinchmap is also bilinear interpolated, further improving efficiency (both reducing the number of texture accesses and leading to a basically single-case algorithm), and also unlocking simple solutions to achieve anti-aliasing, mip-mapping, smoothly starting discontinuities, resolution differences between pinchmaps and signal textures, and so on.

A pinchmap/signal-texture pair is rendered in real time, as it was predictable from the fragment program instruction counts; moreover many per-fragment resources (texture accesses and ALU instructions) are left untapped for any further computation.

The system is designed for minimal performance impact, maximal quality and features during the rendering phase; the cost is that becomes difficult to design a pinchmap that delivers the wanted discontinuities at the wanted locations. This is why the work presented here would have been limitedly useful without an automatic procedure to perform that task with high performance and in acceptable times.

The resulting system -consisting in a preprocessing module to create pinchmap/signalmap pairs, and a fragment program to display them- can be used as a black box by final user.

Limitations: Our approach is not free from limitations: when the complexity of the discontinuity lines defy the pinchmap expressiveness power, e.g. when two or more discontinuity lines cross each other, or three discontinuity lines generate from a point, or too many lines run too near to each other, then some of the lines are lost (and the image appears blurred). Also, in the texture access to the pinchmap we are relying on a HW accelerated bilinear interpolation. This can lead to artifacts if the hardware on the contrary performs some complex (non linear) texel interpolation assum-

Figure 13: *Some results of the automatic pinchmap creation tool. Left: original starting image (alpha channel, which in both cases separates foreground from background, is not shown). Second column: the result pinchmapped texture (actual screenshot of the HW implementation): above both signal and pinchmap texture are sized* 32×32*, below* 64×64*. Third column: for comparison, a* 128×128 *anti-aliased down-sampling of the original image. Last column: the signal map-and the color-map composing the pinchmap.*

ing that it is manipulating colors, for example to take in account gamma correction. These feature must be disabled in order for the pinchmap mechanism to work.

Pinchmap expressiveness: The lines that can be produced are those that obtainable as the intersection of two channels defined at pinchmap texels and bilinearly interpolated in between. This translates in a very powerful expressiveness, as shown by the fact that complex images can be reproduced by very small pinchmap (see Fig. 13).

The constraints described in Sec. 4.4 limit somehow the expressiveness of the representation. For example, it is not possible for a region of the texture separated by discontinuity lines to consist of a single texel. This is an unavoidable consequence of our basic choice: since we always interpolate between four texels, then by definition the smallest separable region consists of four texels. However, this is not a limitation on the shapes that are obtained in the final result after the pinching operations. Parameters γ, which typically range in $[-0.5+0.5]$, can also be assigned to values in $(-1,+1)$ without causing any artefact: the effect is that the discontinuity line appears out of the zone originally covered by the undesired fexels (note that now also one of the two regions at the side of a pinched fexel, instead of being *expanded* over it, is *shrunk* a little).

References

[RBW04] RAMANARAYANAN G., BALA K., WALTER B.: Feature-based textures. In *Proc. of the Eurographics Symposium on Rendering* (June 2004), Eurographics Association. 1, 2

[SCH03] SEN P., CAMMARANO M., HANRAHAN P.: Shadow silhouette maps. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003) 22*, 3 (July 2003), 521–526. 2

[Sen04] SEN P.: Silhouette maps for improved texture magnification. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (August 2004), Eurographics Association. 1, 2

[TC04] TUMBLIN J., CHOUDHURY P.: Bixels: Picture samples with sharp embedded boundaries. In *Proc. of the Eurographics Symposium on Rendering* (June 2004), Eurographics Association. 1, 2

Other examples, updates, executables and FP code available at `http://vcg.isti.cnr.it/pinchmaps`