Real-Time Gradient-Domain Painting

James McCann*
Carnegie Mellon University

Nancy S. Pollard†
Carnegie Mellon University

Figure 1: *Starting from a blank canvas, an artist draws and colors a portrait using gradient-domain brush strokes.* Additive *blend mode used for sketching,* directional *and* additive *for coloring. Artist time: 30 minutes.*

Abstract

We present an image editing program which allows artists to paint in the gradient domain with real-time feedback on megapixel-sized images. Along with a pedestrian, though powerful, gradient-painting brush and gradient-clone tool, we introduce an *edge brush* designed for edge selection and replay. These brushes, coupled with special blending modes, allow users to accomplish global lighting and contrast adjustments using only local image manipulations – e.g. strengthening a given edge or removing a shadow boundary. Such operations would be tedious in a conventional intensity-based paint program and hard for users to get right in the gradient domain without real-time feedback. The core of our paint program is a simple-to-implement GPU multigrid method which allows integration of megapixel-sized full-color gradient fields at over 20 frames per second on modest hardware. By way of evaluation, we present example images produced with our program and characterize the iteration time and convergence rate of our integration method.

CR Categories: I.3.4 [Computing Methodologies]: Computer Graphics—Graphics Utilities;

Keywords: real-time, interactive, gradient, painting, multigrid

1 Introduction

Perceptual experiments suggest that the human visual system works by measuring local intensity differences rather than intensity itself [Werner 1935; Land and McCann 1971], so it seems reasonable that when creating input for the visual system one may wish to work directly in the gradient domain.

*e-mail: jmccann@cs.cmu.edu
†e-mail: nsp@cs.cmu.edu

Figure 2: *Before (*left*) and after (*right*) editing with the edge brush. The user captured the smoothness of the left wall and used it to paint out the edge separating the left and back walls. The user then selected a segment of downspout and created a more whimsical variant. Finally, the tile roof-line to the left was captured and extended. Finishing touches like the second lantern and downspout cleats were added with the clone brush. The edge was painted out with* over *blend mode, while elements were added using* over *and* maximum *modes. Artist time: 2.5 minutes. (Original by clayjar via flickr.)*

The primary problem with working in the gradient domain is that once image gradients have been modified, a best-fit intensity image must be reconstructed before the result can be displayed. This integration process has, until now, required users to wait seconds or minutes to preview their results[1]. Our fast, multigrid integrator removes this restriction, allowing for 20fps integration rates on 1 megapixel images. With real-time feedback – and a selection of brushes and blend modes – we put the user in direct control of her edits, allowing her to interact with gradient images in the same direct way she interacts with a conventional intensity-based paint program.

The workhorse brush in our paint program is the *edge brush*, which allows the user to record edges in the image and paint them elsewhere (e.g. Figure 2), We also provide a simple *gradient brush* which paints oriented edges. This brush is essential when sketching, as in Figure 1, and with appropriate blend modes is also useful when making adjustments. Finally, we retain the *clone brush*, which is a staple of the intensity painting world and has also appeared in previous gradient manipulation systems [Pérez et al. 2003]. Our brushes are explained in more detail in Section 4.

[1]Timing for 1 megapixel images – for 256x256 images, common CPU-based methods allow interactive (1-3fps) integration.

We provide different blend modes in addition to different brushes; the artist is free to use any brush with any blend mode. Basic blend modes *maximum* and *minimum* enhance and suppress detail, respectively, while *add* accumulates the effect of strokes and *over* replaces the existing gradients. Finally, the *directional* mode enhances gradients in the same direction as the current stroke, and is quite useful for shadow and shading editing. We explain these modes in more detail in Section 5.

2 Background

Perhaps the first instance of edge-focused image editing was the contour-based image editor of Elder and Goldberg [2001]. Their high-level edge representation enables users to select, adjust, and delete image edges. These interactions, however, are far from real-time; users queue up edits and then wait minutes for the results. Our paper is presented concurrently with the work of Orzan et al. [2008]; they present a vector graphics color-filling technique based on Poisson interpolation of color constraints and blur values specified along user-controlled splines – a representation similar in spirit to Elder and Goldberg's edge description. A GPU-based multigrid solver (similar to our own) and a fast GPU-based blur computation enable interactive editing of this representation.

Classic gradient-domain methods include dynamic range compression [Fattal et al. 2002], shadow removal [Finlayson et al. 2002], and image compositing [Levin et al. 2003; Pérez et al. 2003]. (One key to seamless image stitching is smoothness preservation [Burt and Adelson 1983], which gradient methods address directly.) Indeed, there are simply far too many interesting gradient-domain techniques to list here; readers wishing to learn more about this rich field would be well served by the recent ICCV course of Agrawal and Raskar [2007].

Broadly, current gradient-domain techniques would appear in the "filters" menu of an image editing program, performing a global operation automatically (range compression [Fattal et al. 2002], shadow removal [Finlayson et al. 2002]), or in a way that can be guided by a few strokes (color editing [Pérez et al. 2003]) or clicks (compositing [Levin et al. 2003; Pérez et al. 2003]). For the seconds-long integration processes used in these techniques, it is important to have such sparse input, as the feedback loop is relatively slow.

Recently, interactive performance was demonstrated for tone-mapping operations on 0.25 megapixel images [Lischinski et al. 2006]; this performance was obtained through preconditioning, which – due to the high calculation cost – is only suitable for edits that keep most of the image the same so that the preconditioner may be reused. A different preconditioning approach is presented in Szeliski [2006].

Agarwala [2007] addressed the scaling behavior of gradient compositing by using quadtrees to substantially reduce both memory and computational requirements. While his method is not real-time, it does scale to integrating substantially larger gradient fields than were feasible in the past – with the restriction that the answer be smooth everywhere except at a few seams.

Perhaps surprisingly, no great breakthrough in technology was required to provide the real-time integration rates at the core of our program. Rather, we simply applied the multigrid method [Press et al. 1992], a general framework for solving systems of equations using coarse-to-fine approximation. We use a relatively straightforward version, overlooking a faster-converging variant [Roberts 2001] which is less suitable to hardware implementation. Our solver is similar in spirit to the generalized GPU multigrid methods presented by Bolz et al. [2003], and Goodnight et al. [2003] though some domain-specific customizations (e.g. not handling complicated boundary shapes) make ours both simpler and somewhat faster.

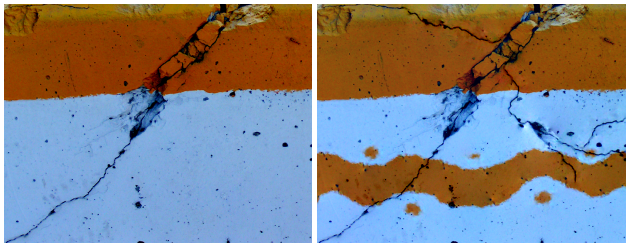

Figure 4: *Before (*left*) and after (*right*) editing with the edge brush. The user extended the crack network using a segment of existing crack and added her own orange stripe to the image with just three strokes – one to select the original edge as an example, and two to draw the top and bottom of the new stripe. All strokes use* additive *blend mode. Artist time: 3 minutes. (Original by Tal Bright via flickr.)*

3 Overview

This paper presents a system which enables artists to paint in the gradient domain much like they are accustomed to painting in the intensity domain. We explain how this system works by first describing the brushes which artists use to draw gradients; then the blending modes by which these new gradients are combined with the existing gradients; and, finally, the integrator that allows us to display the result of this process to the artist in real-time.

4 Brushes

In this work, we are interested in tools analogous to those used in intensity paint programs, allowing the user to draw on the image and receive real-time feedback. A quirk of gradient-domain editing is that instead of editing a single intensity image, one must work with two gradient images – one for the x-gradient and one for the y-gradient; thus, our brushes rely on both mouse position and stroke direction to generate gradients. Examples of each brush's operation are provided in Figure 3.

The **gradient brush** is our simplest gradient-domain paint tool. As a user paints a stroke with the gradient brush, the brush emits gradients of the current color and perpendicular to the stroke direction. Sketching with the gradient brush, an artist is able to define volume and shading as she defines edges, without ever touching the interior of shapes. An example portrait created with gradient-domain sketching is shown in Figure 1.

The trouble with edges produced by the gradient brush is that they don't have the subtle texture and impact of natural edges found in real images. The **edge brush** is our simple and direct solution to this problem. The user first paints an edge selection stroke along a segment of an edge she fancies. The system captures the gradients around this stoke and represents them in local (stroke-relative) coordinates. Now, as the user paints with the edge brush, these captured gradients are "played back" – transformed to the local coordinate system of the current stroke and emitted. We loop edge playback so the user can paint long strokes. Images whose production depended on edge capture and playback are shown in Figure 2 and Figure 4.

Because image compositing is a classical and effective gradient-domain application [Pérez et al. 2003; Levin et al. 2003], we include it in our program. By replacing copy-and-paste or cumber-

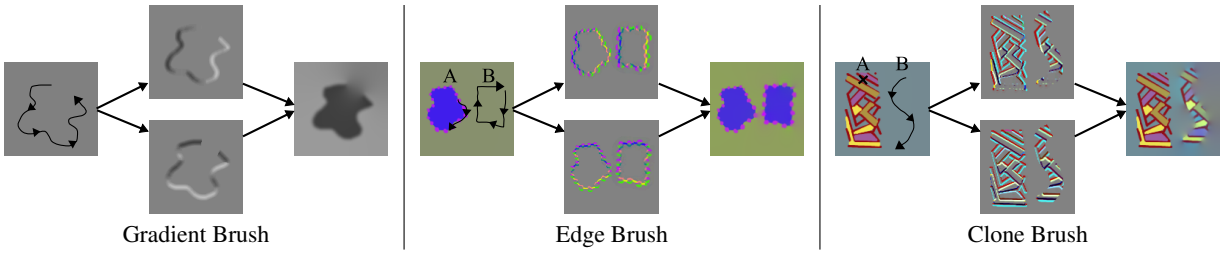

Figure 3: *The operation of our brushes illustrated by original image and stroke(s), produced gradients, and final image. Our system provides real-time feedback during stroke drawing. The* **gradient brush** *paints intensity differences. The* **edge brush** *allows selection (A) and playback (B) of an edge; the edge is looped and re-oriented to allow long strokes. The* **clone brush** *copies gradients relative to a source location (A) onto a destination stroke (B).*

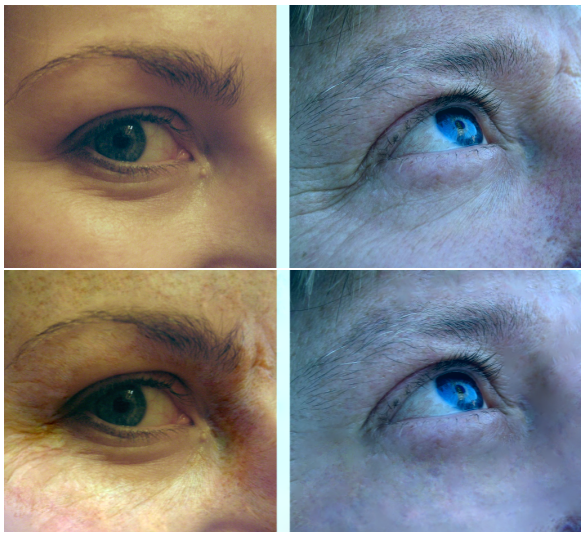

Figure 5: *Before (*top*) and after (*bottom*) swapping wrinkles. Crow's foot wrinkles were copied to the left face with the edge brush. Skin textures were exchanged using the clone tool. Painting with directional blending was used to enhance shading on the right face lost during the cloning.* Maximum *and* over *blending used when copying wrinkles and skin texture;* minimum *blending used to smooth the eye to the right; and* directional *and* additive *blending used to adjust shading. Artist Time: 15 minutes. (Original images by iamsalad,* left*, and burnt out Impurities,* right*, via flickr.)*

some draw-wait cycles with a direct **clone brush**, we give the user more freedom, granularity, and feedback in the cloning process. In fact, because our integrator is global, the user may drag entire cloned regions to, say, better align a panorama – all in real-time. Cloning can be used to move objects, as in Figure 11 and Figure 13. Cloning with multiple blend modes allows copying an object and its lighting effects as in Figure 10. A more subtle clone effect is the skin texture adjustment in Figure 5.

5 Blend Modes

Depending on the situation, the artist may change how her new strokes are blended with the background. Below, we describe the blending modes our paint program supports. Blending is a per-pixel operation; in the following equations, we denote the current background gradient[2] (g^x, g^y) and the current gradient from the brush (b^x, b^y). The same equation is applied across all color channels.

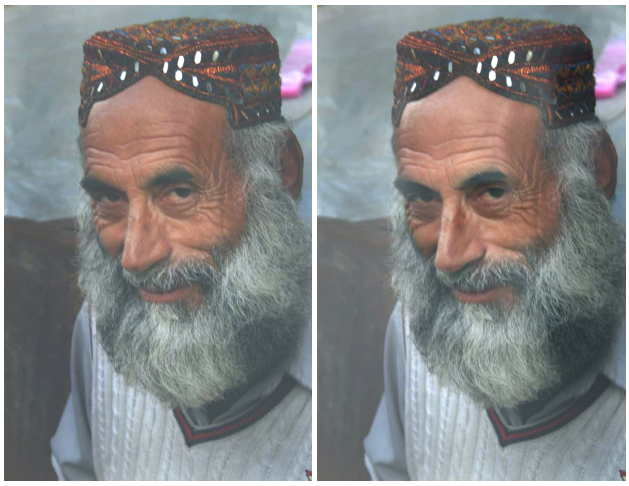

Figure 6: *Before (*left*) and after (*right*) contrast enhancement. The user applied the gradient brush with* directional *and* additive *blends. Strokes along the sides of the face enhance the shading, while strokes along the nose and eyebrows give the features more depth. Artist time: 3 minutes. (Original by babasteve via flickr.)*

Additive blending sums the gradients of the brush and background:

$$(g^x, g^y) \leftarrow (b^x, b^y) + (g^x, g^y) \tag{1}$$

It is useful when sketching, allowing one to build up lines over time, and for simple color and shadow adjustments. It is also useful when building up texture over multiple cloning passes.

Maximum blending selects the larger of the brush and background gradients:

$$(g^x, g^y) \leftarrow \begin{cases} (g^x, g^y) & \text{if } |(g^x, g^y)| > |(b^x, b^y)| \\ (b^x, b^y) & \text{otherwise} \end{cases} \tag{2}$$

This blend is useful when cloning or copying edges, as it provides a primitive sort of automatic matting [Pérez et al. 2003]. We also support **minimum blending**, the logical opposite of maximum blending, which is occasionally useful when cloning smooth areas over noisy ones (e.g. removing wrinkles; see Figure 5).

[2]We use the term "gradient" to indicate the intended use, as these vector fields are often non-conservative and thus not the gradient of any real image.

Figure 7: *Before (*left*) and after (*right*) shadow adjustments. The middle shadow was darkened using* directional *blending; the bottom shadow was removed by cloning over its top and left edges using* over *blending , then repainted as a more diffuse shadow using the gradient brush with* additive*; the top shadow remains unmodified, though the balcony has been tinted by applying colored strokes with* directional *blending to its edges. Artist time: 2.5 minutes. (Original by arbyreed via flickr.)*

Over blending simply replaces the background with the brush:

$$(g^x, g^y) \leftarrow (b^x, b^y) \tag{3}$$

It is useful when cloning; or, with the gradient brush, to erase texture.

Directional blending, a new mode, enhances background gradients that point in the same direction as the brush gradients and suppresses gradients that point in the opposite direction:

$$(g^x, g^y) \leftarrow (g^x, g^y) \cdot \left(1 + \frac{b^x \cdot g^x + b^y \cdot g^y}{g^x \cdot g^x + g^y \cdot g^y}\right) \tag{4}$$

Directional blending is useful for lighting and contrast enhancement, as the artist may "conceal" her edits in existing edges, as in Figures 6 and 7.

6 Integration

Finding the image u with gradients close, in the least-squares sense, to given – potentially non-conservative – edited gradient images G^x, G^y is equivalent to solving a Poisson equation:

$$\nabla^2 u = f \tag{5}$$

Where f is computed from the edited gradients:

$$f_{x,y} = G^x_{x,y} - G^x_{x-1,y} + G^y_{x,y} - G^y_{x,y-1} \tag{6}$$

We solve this equation iteratively using the multigrid method. For a general overview, the reader may refer to Numerical Recipes in C [Press et al. 1992], whose variable-naming conventions we follow. We describe our specific implementation in this section as an aid to those wishing to write similar solvers; our solver is a special case, customized for our specific domain and boundary conditions, of existing GPU multigrid solvers [Goodnight et al. 2003; Bolz et al. 2003].

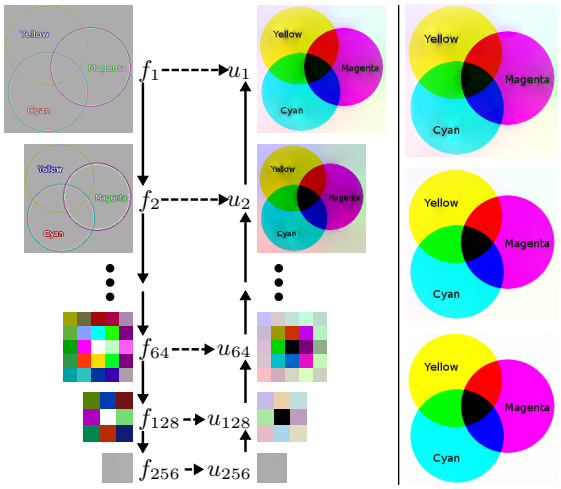

Figure 8: Left*: illustration of a single call to* `VCycle`*, with f_1 set to the gradient field of a 257x257 test image. Arrows show data flow.* Right*: The approximation is not perfect, but improves with subsequent iterations.*

`VCycle`(f_h):
  if (size(f_h) == 1x1) **return** [0]
  $f_{2h} \leftarrow \mathcal{R} f_h$;Restrict f to coarser grid
  $u_{2h} \leftarrow$ `VCycle`(f_{2h}) ;Approximate coarse solution
  $u_h \leftarrow \mathcal{P} u_{2h}$;Interpolate coarse solution
  $u_h \leftarrow$ `Relax`(u_h, f_h, x_{h0}) ;Refine solution
  $u_h \leftarrow$ `Relax`(u_h, f_h, x_{h1}) ;Refine solution (again)
  return u_h

`Relax`(u_h, f_h, x):
  return $\frac{1}{m_h - x}\left(f_h - (\mathcal{L}_h - (m_h - x)I)u_h\right)$;Jacobi

Figure 9: *Pseudo-code implementing our multigrid-based integration algorithm. Each frame,* `VCycle` *is called on the residual error, $f_1 - \mathcal{L}_1 u_1$, to estimate a correction to the current solution. The variable m_h refers to the m associated with $\mathcal{L}_h$ (Equation 8). We set $x_{h0} = -2.1532 + 1.5070 \cdot h^{-1} + 0.5882 \cdot h^{-2}$ and $x_{h1} = 0.1138 + 0.9529 \cdot h^{-1} + 1.5065 \cdot h^{-2}$.*

In one iteration of our multigrid solver (in the jargon, a V-cycle with no pre-smoothing and two post-smoothing steps), we estimate the solution to the linear system $\mathcal{L}_h u_h = f_h$ by recursively estimating the solution to a coarser[3] version of the system $\mathcal{L}_{2h} u_{2h} = f_{2h}$, and then refining that solution using two Jacobi iterations. We provide an illustration in Figure 8, and give pseudo-code in Figure 9.

To effect these changes in grid resolution, we require two operators $\mathcal{R}$ and $\mathcal{P}$. The restriction operator, $\mathcal{R}$, takes a finer f_h to a coarser f_{2h}. The interpolation operator, $\mathcal{P}$, expands a coarse u_{2h} to a finer u_h. Our interpolation and restriction operators are both defined by the same stencil:

$$\mathcal{P} = \mathcal{R} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{7}$$

In other words, $\mathcal{P}$ inserts values and performs bilinear interpolation while $\mathcal{R}$ smooths via convolution with the above, then subsamples. The astute will notice that $\mathcal{R}$ is four times larger than the standard;

[3]The subscript h denotes the spacing of grid points – so, u_{2h} contains one-quarter the pixels of u_h.

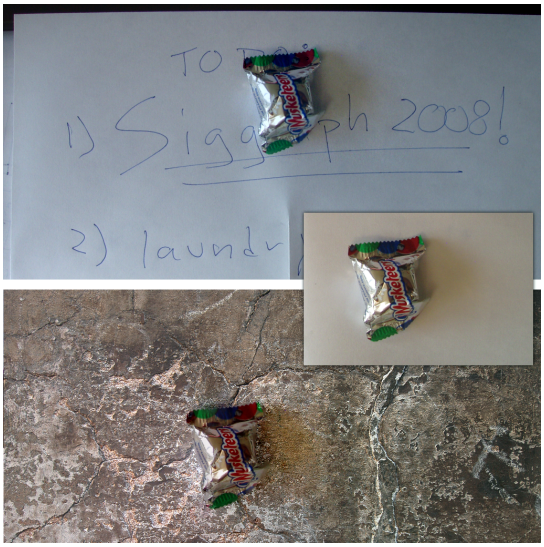

Figure 10: *Cloning with multiple blend modes. The candy (*middle right*) is cloned onto two different backgrounds (*top, bottom*) with* over *blending, while its shadow and reflected light are cloned with* additive *blending. Artist time: 4 minutes. (Bottom background by Stewart via flickr.)*

this tends to keep f_h of a consistent magnitude – important, given the 16-bit floating point format of our target hardware.

The operator $\mathcal{L}_h$ we solve for at spacing h is given by a more complicated stencil:

$$\mathcal{L}_h = \begin{bmatrix} c & e & c \\ e & m & e \\ c & e & c \end{bmatrix}, \text{with } \begin{bmatrix} m \\ e \\ c \end{bmatrix} = \frac{1}{3h^2} \begin{bmatrix} -8h^2 - 4 \\ h^2 + 2 \\ h^2 - 1 \end{bmatrix} \tag{8}$$

Notice $\mathcal{L}_1 = \nabla^2$. These coefficients have been constructed so that the solution at each level, if linearly interpolated to the full image size, would be close as possible, in the least-squares sense, to the target gradients. This choice is consistent with Bolz et al.[2003], though they do not provide an explicit formula and use a different justification.

In our GPU implementation we store all data matrices as 16-bit floating-point textures, and integrate the three color channels in parallel. We use 0-derivative (i.e. Neumann with value zero) boundary conditions, since these seem more natural for editing; however, this implies that the solution is only defined up to an additive constant. We resolve this ambiguity by white-balancing the image. As interactive and consistent performance is important for our application, we run one `VCycle` every frame instead of imposing a termination criterion.

7 Evaluation

We evaluated our integrator on gradient fields taken from images of various sizes and aspect ratios. Our test set included a high-resolution full scan of the "Lenna" image and 24 creative-commons licensed images drawn from the "recently added" list on flickr. To characterize how quickly our integrator converges to the right solution, we modeled a worst-case scenario by setting u to a random initial guess, then calculated the root-mean-square of the residual, $\nabla^2 u - f$, and of the difference between the x-gradient of u and G^x; both are plotted in Figure 12 (left). In practice, images are recognizable after the first integration step (i.e. call to `VCycle`) and

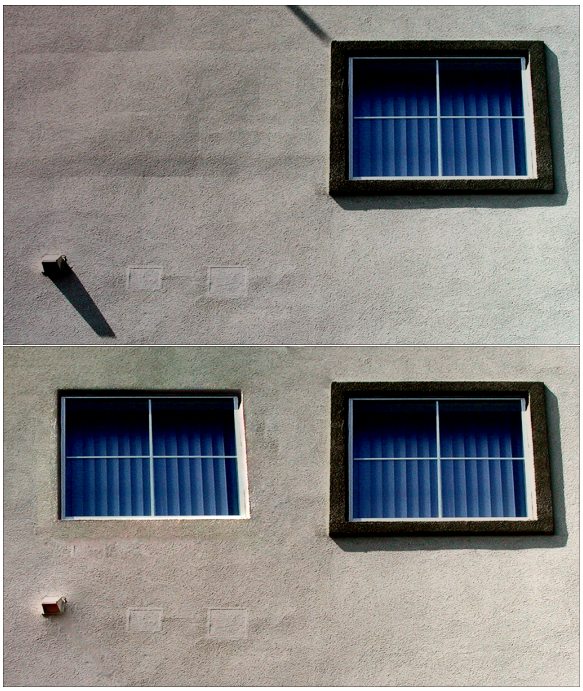

Figure 11: *Before (*top*) and after (*bottom*). The left window is a clone of right (with* over *blend mode), with the outer edge of the surround omitted. Color adjustment (gradient brush with* directional *blend mode) was used to make the inside of the left window the same shade as the right. Two shadows were removed by cloning out edges. Artist time: 1.5 minutes. (Original by P Doodle via flickr.)*

nearly perfect after the second – Figure 12 (center).

We also performed an eigenvalue analysis of a 65x65 problem, similar to that used by Roberts to analyze his method [2001]. We achieve a convergence rate, $\bar{p}_0 = 0.34$, indicating that we remove more than 65% of the remaining error each iteration. This is on par with Roberts' reported value for conventional multigrid. The constants x_{h0} and x_{h1} appearing in our algorithm (Figure 9) were selected by numerical optimization of this $\bar{p}_0$.

Timing for a single integrator cycle (the bulk of the work we do each frame) is recorded in Figure 12. Because integration time is not content-dependent, we augmented our test set with 37 blank images to check integration speeds over a wider range of sizes and aspect ratios. The two computers used for testing were a several-year-old desktop with an Athlon64 3400 CPU, 1 GB of RAM, and a GeForce 6800 GT GPU, running Gentoo Linux – the sort of system that many users have on/under their desks today – and a moderately priced ($1500) workstation with a Core2 Quad Q6600 CPU, 4 GB of RAM, and a GeForce 8600 GTS GPU, running Windows Vista. Even on the more modest hardware, our software comfortably edits 1 megapixel images at around 20fps. Algorithmic scaling is theoretically linear in the number of pixels, and this tendency is very pronounced as long as all textures are resident in video memory.

8 Other Applications

Because our integrator is so fast, it might be useful for real-time gradient-domain compositing of video streams – for instance, of a computer generated object into a real scene for augmented reality, or of a blue-screened actor into a virtual set. Care would have to

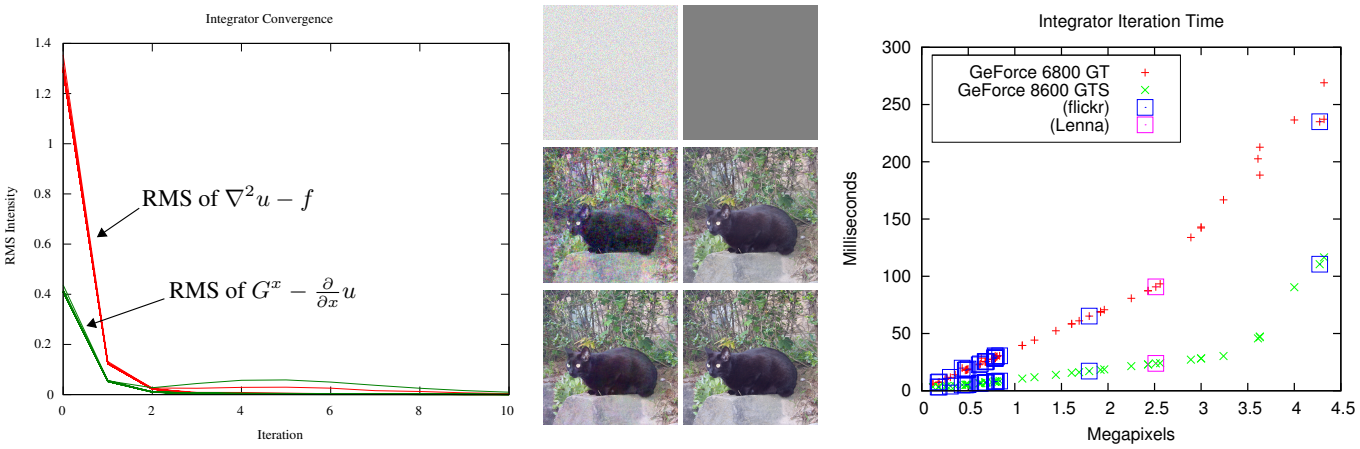

Figure 12: **Left:** *Root-mean-square (over all pixels) error after a given number of V-cycles. Results for all test images are overplotted. Image intensities are represented in the* $[0, 1]$ *range. The increase in error around the fourth iteration is due to approximations made when handling boundary conditions of certain image sizes.* **Center:** *Integrator convergence on an example image – initial condition (top row) and result of one (middle row) or two (bottom row) V-cycles.* **Right:** *Milliseconds per integrator cycle; performance is linear until data exceeds video memory – about 2.5 megapixels on modest hardware, 3.5 on the newer hardware. Un-boxed points are synthetic examples.*

be taken with frame-to-frame brightness and color continuity, both of the scene and of the pasted object, either by adding a data term to the integration or by introducing more complex boundary conditions.

With a DSP implementation, our integrator could serve as the core of a viewfinder for a gradient camera [Tumblin et al. 2005]. A gradient camera measures the gradients of a light field instead of its intensity, allowing HDR capture and improving noise-resilience. However, one drawback is the lack of a live image preview ability. Some computational savings would come from only having to compute a final image at a coarser level, say, a 256x256 preview of a 1024x1024 image. Our choice of $\mathcal{L}_h$ and $\mathcal{R}$ guarantees that this coarser version of the image is still a good approximation of the gradients in a least-squares sense.

Our method may also be applicable to terrain height-field editing. Users could specify where they wanted cliffs and gullies instead of having to specify explicit heights.

9 Discussion

With a real-time integrator, gradient editing can be brought from the realm of edit-and-wait filters into the realm of directly-controlled brushes. Our simple multigrid algorithm, implemented on the GPU, can handle 1 megapixel images at 20fps on a modest computer. Our editor allows users to paint with three brushes, a simple gradient brush useful for sketching and editing shading and shadows, an edge brush specifically designed to capture and replay edges from the image, and a clone brush. Each brush may be used with different blending modes, including directional blending mode which "hides" edits in already existing edges.

However, our approach has limitations. More memory is required than in a conventional image editor; G^x and G^y must be stored, as well as multi-scale versions of both f and u, not to mention temporary space for computations – this requires about 5.6 times more memory than just storing u; for good performance, this memory must be GPU-local. Conventional image editors handle large images by tiling them into smaller pieces, then only loading those tiles involved in the current operation into memory. Such an approach could also work with our integrator, as all memory operations in our algorithm are local. Alternatively, we might be able to minimize main-memory to GPU transfers by adapting the streaming multigrid presented by Kazhdan and Hoppe [2008].

One of the main speed-ups of our integrator is that its primitive operations map well onto graphics hardware. A matlab CPU implementation ran only a single iteration on a 1024x1024 image in the time it took for a direct FFT-based method – see Agrawal and Raskar [2007] – to produce an exact solution (about one second). Thus, for general CPU use we recommend using such a direct solution. This result also suggests that GPU-based FFTs may be worth future investigation; presently, library support for such methods on older graphics cards is poor.

The interface features of our current gradient paint system are somewhat primitive. The looping of the edge brush could be made more intelligent, either using a video-textures-like approach [Schödl et al. 2000], or by performing a search to find good cut points given the specific properties of the current stroke. Edge selection might be easier if the user were provided with control over the frequency content of the copied edge (e.g. to separate a soft shadow edge from high-frequency grass). And tools to limit the effect of edits, through either modified boundary conditions or a data term in the integrator, would be convenient. Like any image editing system, our paint tools can produce artifacts – one can over-smooth an image, or introduce dynamic range differences or odd textures. However, also like any image editing system, these can be edited-out again or the offending strokes undone.

We have informally tested our system with a number of users (including computer graphics students and faculty, art students, and experienced artists). Users generally had trouble with three aspects of our system:

- **Stroke Directionality** – artists are accustomed to drawing strokes in whatever direction feels natural, so strokes in different directions having different meanings was a stumbling block at first.
- **Relative Color** – both technical users and artists are used to specifying color in an absolute way. Users reported that it was hard to intuit what effect a color choice would have, as edges implicitly specify both a color (on one side) and its complement (on the other).

Figure 13: *Before (*left*) and two altered versions (*middle, right*). Shadow adjustments and removal, cloning, and edge capture all come into play.* Over *blending used for erasing;* maximum, over, *and* additive *blending used for re-painting. (Original by jurek d via flickr.)*

- **Dynamic Range** – it is easy to build up large range differences by drawing many parallel strokes. Users were often caught by this in an early drawing.

The interactivity of the system, however, largely mitigated these concerns – users could simply try something and, if it didn't look good, undo it. Additionally, some simple interface changes may help to address these problems; namely: automatic direction determination for strokes, or at least a direction-switch key; a dual-eyedropper color chooser to make the relative nature of color more accessible; and an automatic range-compression/adjustment tool to recover from high-dynamic range problems.

In the future, we would like to see gradient-domain brushes alongside classic intensity brushes in image editing applications. This paper has taken an important step toward that goal, demonstrating an easy-to-implement real-time integration system coupled to an interface with different brush types and blending modes. Painting in the gradient domain gives users the ability to create and modify edges without specifying an associated region – something that makes many tasks, like the adjustment of shadows and shading, much easier to perform.

Acknowledgments

The authors wish to thank: Moshe Mahler for his "skills of an artist" (teaser image; skateboard fellow in video); NVIDIA for the GeForce 8600 used in testing; all those, including the anonymous reviewers, who helped in editing the paper; and everyone who tried our system.

References

AGARWALA, A. 2007. Efficient gradient-domain compositing using quadtrees. *ACM Transactions on Graphics 26*, 3.

AGRAWAL, A., AND RASKAR, R., 2007. Gradient domain manipulation techniques in vision and graphics. ICCV 2007 Course.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖODER, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3, 917–924.

BURT, P. J., AND ADELSON, E. H. 1983. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics 2*, 4, 217–236.

ELDER, J. H., AND GOLDBERG, R. M. 2001. Image editing in the contour domain. *IEEE Transactions on Pattern Analysis and Machine Intelligence 23*, 3, 291–296.

FATTAL, R., LISCHINSKI, D., AND WERMAN, M. 2002. Gradient domain high dynamic range compression. *ACM Transactions on Graphics 21*, 3, 249–256.

FINLAYSON, G., HORDLEY, S., AND DREW, M. 2002. Removing shadows from images. In *ECCV 2002*.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03*, 102–111.

KAZHDAN, M., AND HOPPE, H. 2008. Streaming multigrid for gradient-domain operations on large images. *ACM Transactions on Graphics 27*, 3.

LAND, E. H., AND MCCANN, J. J. 1971. Lightness and retinex theory. *Journal of the Optical Society of America (1917-1983) 61* (Jan.), 1–11.

LEVIN, A., ZOMET, A., PELEG, S., AND WEISS, Y., 2003. Seamless image stitching in the gradient domain. Hebrew University Tech Report 2003-82.

LISCHINSKI, D., FARBMAN, Z., UYTTENDAELE, M., AND SZELISKI, R. 2006. Interactive local adjustment of tonal values. *ACM Transactions on Graphics 25*, 3, 646–653.

ORZAN, A., BOUSSEAU, A., WINNEMOELLER, H., BARLA, P., JOËLLE, AND SALESIN, D. 2008. Diffusion curves: A vector representation for smooth shaded images. *ACM Transactions on Graphics 27*, 3.

PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. *ACM Transactions on Graphics 22*, 3, 313–318.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, ch. 19.6, 871–888.

ROBERTS, A. J. 2001. Simple and fast multigrid solution of Poisson's equation using diagonally oriented grids. *ANZIAM J. 43*, E (July), E1–E36.

SCHÖDL, A., SZELISKI, R., SALESIN, D. H., AND ESSA, I. 2000. Video textures. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 489–498.

SZELISKI, R. 2006. Locally adapted hierarchical basis preconditioning. *ACM Transactions on Graphics 25*, 3, 1135–1143.

TUMBLIN, J., AGRAWAL, A., AND RASKAR, R. 2005. Why I want a gradient camera. In *Proceedings of IEEE CVPR 2005*, vol. 1, 103–110.

WERNER, H. 1935. Studies on contour: I. qualitative analyses. *The American Journal of Psychology 47*, 1 (Jan.), 40–64.