EUROGRAPHICS 2012 / P. Cignoni, T. Ertl
(Guest Editors)

Volume 31 (2012), Number 2

Soft Stacking

J. McCann† [1,2] and N. S. Pollard[2]

[1]Adobe Systems, Inc.
[2]Carnegie Mellon Univeristy

Figure 1: *A composition created with our brush-based soft stacking method. From left to right: source layers; initial order; some smoke pushed behind box; colored and white smoke interleaved; and colored smoke replaced with edited version.*

Abstract

In this paper, we present a continuous approach to ordering 2D images when compositing. Previous methods for stacking image layers require them to appear in a single (though possibly different) order at every point in the image. Our soft stacking approach removes this restriction – allowing layers to stack as if they were volumes of fog, appearing partially in front of and partially in back of other layers within the same pixel*, and moving smoothly through other layers across the image. Our approach involves augmenting each pixel with* stacking coefficients *– a necessary and sufficient representation for sub-pixel stacking complexity. These stacking coefficients arise naturally when considering sub-pixel stacking complexity, much as continuous (alpha) transparency arises when considering sub-pixel coverage complexity.*

While the number of stacking coefficients required to represent all possible sub-pixel stacking arrangements is factorial in the number of layers in the stack, in many practical situations only a small subset of the stacking coefficients are nonzero. We use this sparsity as the basis of a prototype that allows artists to interactively paint stacking adjustments into composites. Additionally, we demonstrate how to generate optimally-stacked images under a generalized notion of stacking consistency.

Categories and Subject Descriptors (according to ACM CCS): Computing Methodologies [I.3.3]: Computer Graphics—Picture/Image Generation

1. Introduction

Conventional 2D compositing methods require that image elements ("layers") appear in the same order everywhere they overlap. Approaches have been developed that relax this restriction, allowing layers to be stacked in different orders in different places (e.g. [Duf85]). However, these approaches are still limited in that they are discrete: they select, for each pixel, exactly one stacking. Such discrete methods are unable to accurately handle image elements whose color comes from more than one depth value (like smoke or fog), as these elements may well appear both above and below other layers in a final composition (see Figure 2).

Our continuous approach, on the other hand, introduces per-pixel *stacking coefficients*, which are both necessary and sufficient to represent sub-pixel stacking complexity in the presence of arbitrary layer blending modes.

We demonstrate two prototype systems for editing stacking coefficients: a brush-based stacking prototype wherein artists can paint with various layer order adjustments; and an optimization-based stacking prototype which interpolates artist-specified stacking constraints across an image. The

† Chairman Eurographics Publications Board

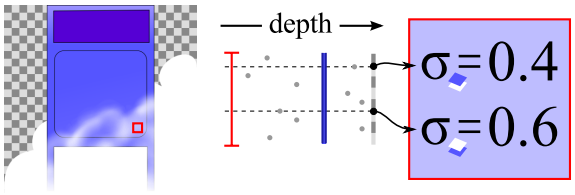

Figure 2: ***Left:*** *In the pixel marked in red, some fog is in front of the postbox and some is behind.* ***Center:*** *a side view with example box-atop-fog (top dashed line) and fog-atop-box (bottom dashed line) orders marked. Our stacking coefficients,* ***right****, track the frequency of these stacking orders.*

brush-based prototype is useful when creating artistic composites, like Figure 1, where iteration and experimentation are required to generate pleasing results. The optimization-based prototype is more suited to composites which have a simple stacking structure but are not amenable to discrete stacking methods because of glows or fog. In contrast to ad-hoc methods involving layer duplication and opacity adjustment, our prototypes allow layer contents and blending modes to be changed while retaining stacking edits.

Given L layers, $L!$ stacking coefficients are required to represent all possible stacking orders at each pixel. This limits our optimization prototype to working with three layers at 500x500 resolution, or two layers at 1 megapixel. However, our brush-based prototype enforces sparsity in stacking coefficients, so remains interactive even with 10 megapixel images and tens of layers and mappings.

2. Background

When creating two-dimensional compositions on a computer, artists often assemble final images from stacks of layers. Layers allow artists to separate semantically different portions of an input scene, adjust gross composition, and achieve occlusion effects.

The notion of creating such layered scenes on computers dates to the 1970s – though it duplicates practices in the film industry that go back much further. Computer compositing was formalized as a "compositing algebra" by Porter and Duff [PD84]. A short history of the subject is available from Alvy Ray Smith [Smi95], and the reader is encouraged to spend a few enjoyable minutes looking it over.

In certain scenarios, it is infeasible or artistically undesirable that layers should appear in the same order everywhere in an image. Methods have been proposed for 3D scenes [Duf85, SL98] and 2D images [Wil06, MP09] that allow local ordering of scene elements. Indeed, such methods have been adopted into commercial software [Med09, ASP07]. Recently, researchers have begun to apply mixed 2D/3D approaches, enlivening flat scenes with inferred [RID10] or interpolated depth [SSJ*10], and using 2D layering operations to manipulate 3D objects [IM10]. While the aforementioned methods are discrete (choosing one stacking order per pixel), other hybrid approaches allow continuous mixed stackings, blending depth-based and user-specified compositing orders [BRV*10], and interpolating temporal and depth information when drawing 3D paint strokes [SSGS11, BSS*11]. In contrast to these approaches, our method does not use any 3D information, and is sufficient to represent all possible sub-pixel stacking order variations.

3. Construction

Our soft stacking approach extends per-pixel stacking in much the same way that image alpha extends the notion of a 0/1 coverage mask ("bitmap transparency") (Figure 3). That is, both alpha values and our stacking coefficients are representations for sub-pixel structural information which are necessary and sufficient to perform compositing.

First, we define the problem setting. We wish to composite a stack of L layers, which we label with integers $1,\ldots,L$. Each layer, i, has an associated transfer function f_i, where $f_i(\mathbf{x},\mathbf{c})$ is the expected color when viewing color $\mathbf{c}$ through layer i at pixel $\mathbf{x}$. Notice that the transfer function f_i represents both layer color and blending mode. †

Notice that in this formulation, the material of a layer either interacts with colors below it or does not. However, multiple-interaction models like the opaque pigment/colored media Beta Color Model [OW91] are still supported thanks to our use of general f.

The goal of any compositing algorithm is to determine the expected color inside every pixel, given layer ordering and opacity information.

$$\mathbf{c}_{\text{final}}(\mathbf{x}) \equiv \mathbb{E}(\text{color at random point in } \mathbf{x}) \qquad (1)$$

In case of per-pixel ordering and per-pixel coverage information, this is a simple matter of applying the transfer functions corresponding to opaque layers in bottom-to-top order (that is, applying the bottom-most transfer function first and working up):

$$\mathbf{c}_{\text{final}} \equiv f_{o_1}\left(\cdots f_{o_N}(\mathbf{c}_{\text{background}})\right) \quad \text{where } o_1,\ldots,o_N \text{ are the opaque layers at } \mathbf{x}, \text{ in top-to-bottom order} \qquad (2)$$

(Note that we have dropped the pixel location, $\mathbf{x}$, for compactness; a conceit we will continue.)

We now briefly re-iterate a standard derivation of alpha, then follow with our extension to sub-pixel stacking complexity.

† Having such a flexible notion of compositing is important for real-world image editing applications; for example, the GNU Image Manipulation Program – a common layer-based painting program – supports 21 different blending modes, ranging from the standard `over` operator ("Normal") to complicated non-linear functions (e.g., "Hue") [GIM10].

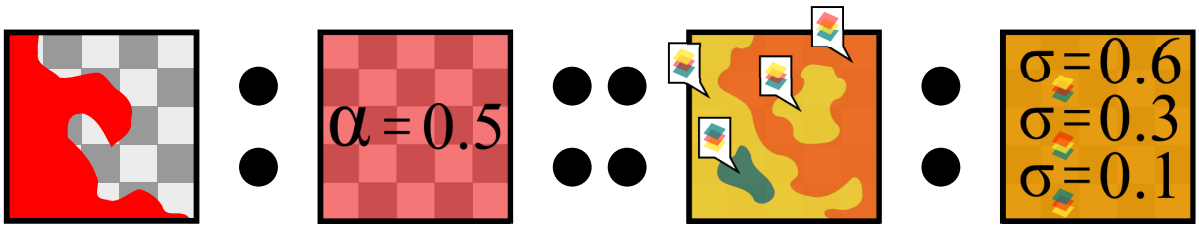

Figure 3: *Stacking coefficients by analogy. Inside a single pixel, the area covered by a layer can be summarized by opacity (α). Similarly, inside a single pixel, the areas where a set of layers assume different stacking orders can be summarized by a list of stacking coefficients (σ). In both cases the representation is necessary and sufficient for compositing tasks.*

3.1. From Coverage to Alpha

Notice that in (2) every layer either completely covers a pixel or does not cover the pixel at all. To extend to the case of partially covered pixels, one must know the probability of a random sample inside each pixel interacting with each layer. Indeed, this is a definition of conventional opacity, α:

$$\alpha_i(\mathbf{x}) \equiv \mathbb{P}(\text{random point in } \mathbf{x} \text{ interacts with layer } i) \quad (3)$$

With these probabilities in hand – and assuming uncorrelated sub-pixel coverage between layers – the expected value in (1) can be re-written in conventional image compositing format:

$$\begin{aligned} \mathbf{c}_{\text{final}} &\equiv \mathbf{c}_{l_1/\cdots/l_L} \\ &\quad \text{where } l_1,\ldots,l_L \text{ are the layers at } \mathbf{x}, \\ &\quad \text{in top-to-bottom order, and:} \\ \mathbf{c}_{i/j/\ldots} &\equiv (1-\alpha_i)\mathbf{c}_{j/\ldots} + \alpha_i f_i\left(\mathbf{c}_{j/\ldots}\right) \\ \mathbf{c}_i &\equiv (1-\alpha_i)\mathbf{c}_{\text{background}} + \alpha_i f_i\left(\mathbf{c}_{\text{background}}\right) \end{aligned} \quad (4)$$

In the above equation – and the remainder of this paper – we write stacking orders as lists of indices separated by forward slashes; this notation is compact and gives a visual hint as to the order. Thus, $1/3/2$ denotes a stack with layer 1 over layer 3 over layer 2.

Notice that the per-pixel alpha values are both sufficient – no other information is needed to write down the expected color of the stack – and necessary – given any black-box method of computing the final composited color, one can read back the α values for each layer by varying the layers' transfer functions (we give a construction in Appendix A).

3.2. From Local Layering to Soft Stacking

The construction of our stacking coefficients follows closely the construction of continuous alpha. In a setting where stackings may vary even inside a pixel, the expected color expression now must take into account the order of layers that is sampled. (Notice the similarity to (3).)

$$\sigma_{i/\cdots/k}(\mathbf{x}) \equiv \mathbb{P}(\text{random sample in } \mathbf{x} \text{ encounters layers stacked in order } i/\cdots/k) \quad (5)$$

These stacking coefficients, σ, are not a per-layer variable; rather, given L layers, there are $L!$ stacking coefficients, one for each stacking. Also, as we've defined these stacking coefficients in terms of probabilities, any sub-pixel stacking structure will result in stacking coefficients that are non-negative and sum to one. (The converse is also true[‡].)

With these stacking coefficients, σ, in hand – and making an assumption that stacking information and alpha information are independent – we can re-write the expected color of the composite by expanding over possible stacking orders:

$$c_{\text{final}}(\mathbf{x}) \equiv \sum_{p \in \text{perm}(L)} \sigma_p \mathbb{E}(\text{color at } \mathbf{x} | \text{layers stacked in order } p) \quad (6)$$

Note that here we use $\text{perm}(L)$ ("permutations of $1,\ldots,L$") to denote the set of all stacking orders.

Of course, we've already written down the expected color for a given stacking in (4), so we substitute to get a more compact expression:

$$\mathbf{c}_{\text{final}} \equiv \sum_{p \in \text{perm}(L)} \sigma_p \mathbf{c}_p \quad \text{with} \quad \begin{array}{l} \forall p : \sigma_p \geq 0 \\ \sum \sigma_p = 1 \end{array} \quad (7)$$

In other words, the expected color of a pixel with sub-pixel stacking variation is a convex combination of the expected colors of each stacking, with coefficients proportional to the area of the pixel appearing in each order.

Like α, the constructed stacking coefficients (σ values) are both sufficient to determine the final pixel color, and necessary. We give a construction to read back σ values from a black-box compositing function in Appendix B.

‡ Given stacking coefficients summing to one, one can create a sub-pixel structure which would yield these coefficients by, e.g., slicing the pixel into $L!$ vertical bands with different stackings orders and setting their widths properly

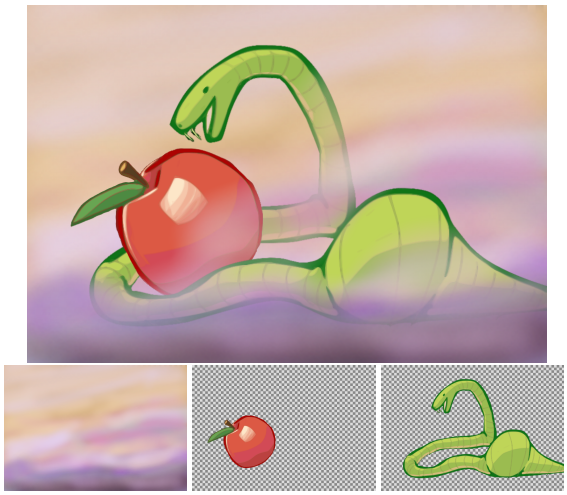

Figure 4: *A brush-based stacking result with three layers. Notice the local ordering of the snake and apple as well as the snake, apple, and mist.*

4. Painting Continuous Stackings

Having put forward the notion of using stacking coefficients to represent sub-pixel stacking complexity, we now describe a brush-based stacking editor that avoids storing $L!$ stacking coefficients at each pixel. Our editor allows artists to manipulate sub-pixel layer orderings in order to create composites with layer interleaving and volumetric effects (e.g. Figure 4).

In our brush-based prototype, artists begin with layers stacked in a global order, and locally refine this stacking to achieve their desired result. This refinement is accomplished by defining a mapping (an operation on stacking orders, like "put fog layer behind the person layer"), then painting per-pixel weights that control the influence of this mapping. Artists can paint these weights with semi-transparent brushes, leading to mixed stackings (e.g., 50% fog-over-person, 50% person-over-fog). Over time, a list of such mappings and their weights accumulate, with the final stacking coefficients at each pixel defined as the result of applying each mapping in sequence.

We allow artists to paint (or erase) weights for any mapping at any time; this is useful for making small adjustments without having to add additional mappings.

Our brush-based stacking editor doesn't explicitly store stacking coefficients; rather, it computes them as needed from the artist-defined mappings. A trimming step during this computation guarantees linear time and memory complexity. Further, the system uses the fact that pixel color is linear in mapping weights to provide real-time feedback when painting, even in arbitrarily complex compositions.

4.1. Mapping Specification

In our system, users paint with stacking mappings, $M : \text{perm}(L) \rightarrow \text{perm}(L)$, functions from stacking orders to stacking orders. While our rendering system supports arbitrary mappings, our interface's stacking coefficient mapping vocabulary is somewhat more limited (though, in practice, it seems to contain all the useful operators).

In our interface, users specify mappings with shorthand phrases like "$a > b \& e > c$". These phrases tell the system which layers to adjust (a, b, c, and e), what direction to move them (up if the condition is written with ">", down if it is written with "<"), and when to stop (when a is above b, c, and e; b is above c; and e is above c).

Under the hood, our prototype applies these phrases by using a slightly modified topological sort.

We have also developed an interface for specifying such phrases by graphically arranging boxes representing layers. This is used in our optimization-based editor, but we have not yet included it in our brush-based stacking prototype.

4.2. Mapping Weights

With each mapping M, our system stores per-pixel weights, β, that specify the strength of the mapping. This strength, $\beta(\mathbf{x}) \in [0,1]$ controls how much of the stacking coefficient $\sigma_q(\mathbf{x})$ for stacking q is moved to the stacking coefficient for stacking $M(q)$. We write this succinctly by introducing $\text{Pre}(p)$, a function that takes stacking order p to the set of all stacking orders q such that q maps to p under M:

$$\sigma'_p \leftarrow (1-\beta)\sigma_p + \sum_{q \in \text{Pre}(p)} \beta\sigma_q \qquad \text{where } \text{Pre}(p) \equiv \{q \mid M(q) = p\} \tag{8}$$

In our system, the user specifies these β values by painting.

4.3. Display

To produce a final image, our program must transform a series of mappings and weights to a final pixel color. As we'd like our system to remain general, this method should not rely on simplifications afforded by specific mapping types or layer transfer functions. Further, it is important to maintain interactive rendering performance and provide real-time feedback when the user is painting stacking order mappings.

0: `Render:`
1: $\sigma_p \leftarrow \begin{cases} 1 & \text{if } p = p_{\text{init}} \\ 0 & \text{otherwise} \end{cases}$;start with initial order
2: **foreach** mapping M with opacity β:
3: $\sigma \leftarrow \text{applyMapping}(M, \beta, \sigma)$;use Eqn 8
4: **while** $\text{countNonzero}(\sigma) > limit$:
5: Set smallest nonzero element of σ to zero
6: $\sigma \leftarrow \sigma / ||\sigma||_1$;normalize
7: **return** $\sum_{p \in \text{perm}(L)} \sigma_p \mathbf{c}_p$;stackings to color (Eqn 7)

Figure 5: *Code implementing our rendering algorithm, which computes the final color of a pixel given the list of stacking order mappings M applied by the artist at that pixel. For efficiency, only nonzero coefficients of σ are stored, and only a limited number are kept (lines 4,5).*

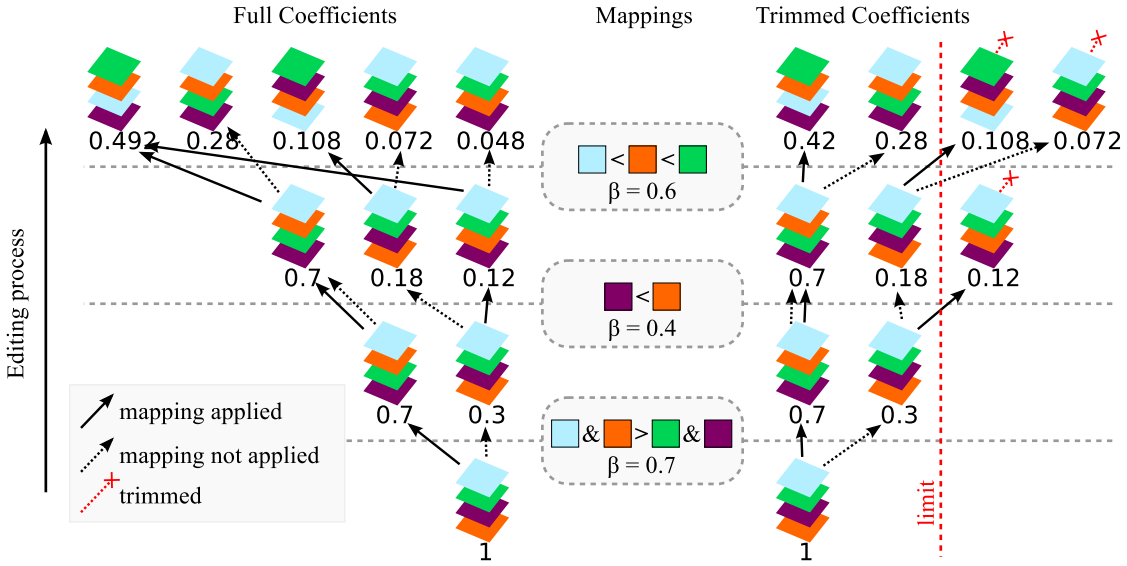

Figure 6: *The computation of stacking coefficients, given a series of mappings. The full set of coefficients (**left**) can grow exponentially, which is why our renderer trims the set of coefficients after applying each mapping (**right**). The mappings applied (**center**) are: move blue and orange up until they are over green and purple; move purple down until it is under orange; and move blue and orange down until blue is under orange is under green. The* β *values give the opacity, at the current pixel, of the brush used to apply each mapping.*

Our prototype uses a trimmed direct solution to determine stacking coefficients (and, thus, pixel color). This method is visualized in Figure 6; pseudo-code is provided in Figure 5. This method works by maintaining a sparse vector of stacking coefficients, starting with a single non-zero coefficient (the artist's initial stacking) and applying each artist-specified mapping in turn (using (8)). If, after applying a mapping, the number of nonzero coefficients has grown too large, the smallest stacking coefficients are dropped. After all mappings are applied, the remaining coefficients are divided by their sum to compensate for energy lost during the trimming process. These coefficients are then used to compute the color, as per (7).

Though our rendering scheme is already quite fast, we can provide even faster feedback to artists while they are painting by using the linearity of stacking coefficients. Notice that, by (8), the final stacking coefficients at a pixel are linear in the weight, β, of any given mapping. Further, by (7), the final pixel color is linear in the stacking coefficients. Thus, for any mapping M, anywhere in the stack of applied mappings, the final pixel color is a linear interpolation of the color when that mapping isn't applied and when it is applied at full weight:

$$\begin{aligned} \mathbf{c}_{\text{final}} &\equiv (1-\beta)\mathbf{c}_{\text{no-M}} + \beta\mathbf{c}_{\text{M}} \\ \text{where } \mathbf{c}_{\text{no-M}} &\equiv \text{final color without } M \\ \mathbf{c}_{\text{M}} &\equiv \text{final color with opaque } M \end{aligned} \tag{9}$$

As soon as the artist selects an existing mapping (or creates a new mapping), our prototype begins to render versions of the image without that mapping applied at all and with that mapping applied with 100% weight. As portions of the image finish rendering, they are uploaded to the GPU as textures. Wherever the user draws, a simple shader computes the interpolation for real-time feedback.

If the user begins painting before this precomputation has finished, a placeholder checkerboard texture is used in the blending. Our system preferentially renders parts of the image close to the user's cursor so that when a stroke is started, it is likely we can provide feedback immediately.

4.4. Implementation

Our prototype is written in C++. We use the Qt toolkit for UI, and OpenGL for display and for real-time painting. All images are treated in 128x128 tiles, with stacks of layer and mapping tiles handed out to background threads for rendering. This allows our prototype to make good use of multiple cores.

Our renderer's theoretical time complexity is $O(\#\text{layers} \cdot \#\text{mappings} \cdot \#\text{retained coefficients})$ and memory complexity is $O(\min(\#\text{retained coefficients}, 2^{\#\text{mappings}}, 2^{\#\text{layers}}))$. Our implementation has been tuned for common workloads, so performs better with coherent mapping weights than with random ones, and with layer and mapping counts that allow one stack of tiles to fit in cache.

The examples in this paper all rendered very quickly, even without trimming. On a single core of an aging workstation (Core 2 Quad Q6600) running Linux, Figure 8, a 10 megapixel composite with six layers and eleven mappings,

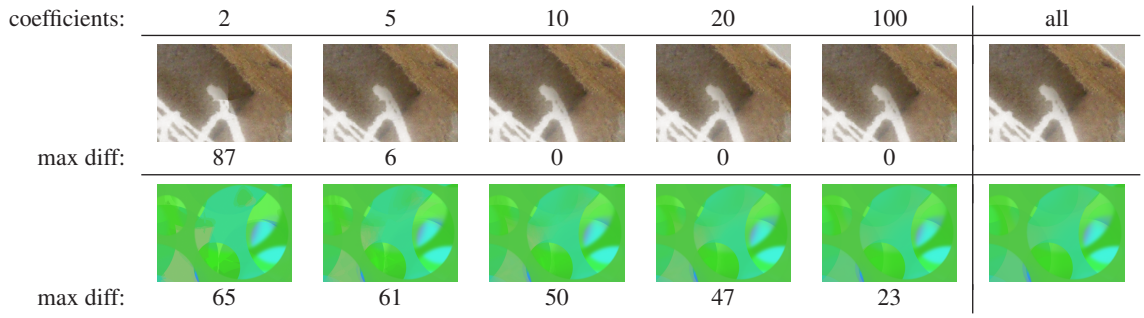

Figure 7: *Banding artifacts appear if too few stacking coefficeints are retained by our rendering algorithm. "Max diff" is the maximum absolute difference between the trimmed and full solution, relative to color values in the range* $[0,255]$. *The top row is a crop from Figure 8, while the bottom row is a crop from a stress test with 20 layers and 20 random mappings.*

takes 1.7 seconds to render, while Figure 1, a five megapixel composite with four layers and five mappings, takes 0.5 seconds. Note that, when painting, any tiles under the current stroke need to be rendered twice (due to (9)), at which point any further modification is real-time. Thus the worst-case[§] lag between brush touch-down and rendered feedback is about 25 milliseconds for this example.

We compare the appearance of our rendered results with different numbers of retained coefficients in Figure 7. When coefficients are trimmed too vigorously, banding artifacts can appear. However, with 10 coefficients, the results appear smooth, even if they may not exactly match the untrimmed solution.

4.5. Results

We find that our brush-based stacking system is useful for dealing with transparent and translucent smoke (Figure 1), and adding fog- or mist-like effects to a scene (Figure 4). It can also be used in situations where previous discrete layering approaches would be useful, but are stymied by the lack of a clean alpha channel (the rings in Figure 8).

One benefit of using a tool that knows about stacking coefficients is that layers can be changed after the stacking has been determined. For instance, when creating Figure 8, the initial composite looked somewhat flat. It was easy to add defocus blur to the rings by modifying (and re-loading) the source layers while keeping the same mappings and weights.

4.6. User Comments

To informally evaluate usability of our brush-based stacking system, we introduced two technically-literate Photoshop users to our system, then interviewed them as they duplicated the stacking in Figure 4. Both users found the painting process intuitive, though they generally found that specifying mappings by typing was cumbersome, and pointed out that this could be a stumbling block for novice users.

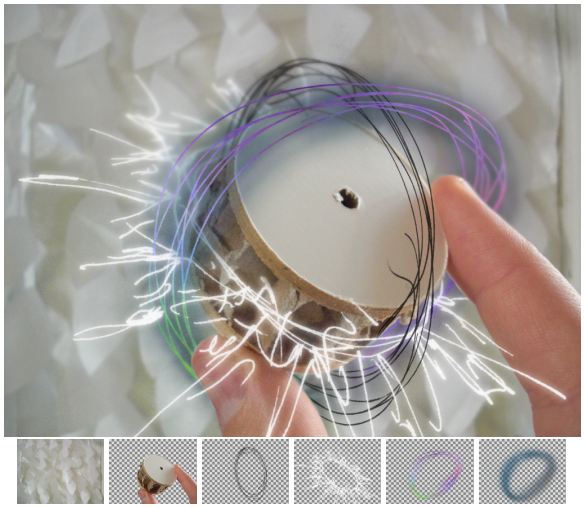

Figure 8: *Brush-based soft stacking was used to interleave the rings and their glows in this 10 megapixel image. Our approach handles arbitrary transfer functions, like the translucent ("multiply") glow around the colored ring.*

One of the users performed further tasks, including replacing the snake layer in their version of Figure 4 with a worm layer (duplicating a task which appears in the video), copying the stacking in Figure 1, and attempting each of these tasks in Photoshop. This user observed that with our brush-based soft stacking prototype, he could immediately begin restacking, whereas in Photoshop he needed to figure out a plan first. He also commented that restacking transparent layers was more difficult in Photoshop because layer duplication changed the look of the layer (specifically, the white smoke in Figure 1 got oversaturated, while the colored smoke became too dark).

[§] This is a pessimistic estimate because the system pre-renders tiles as soon as a mapping is selected, favoring those tiles closer to the mouse cursor.

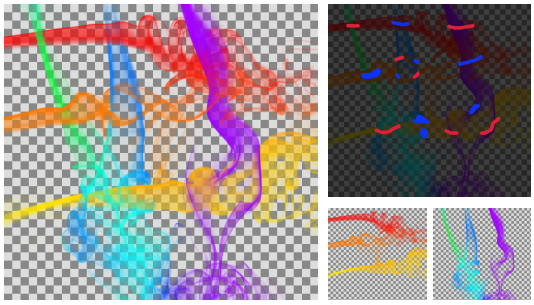

Figure 9: *Two photos of smoke are woven using our optimization-based method. Constraints are horizontal smoke over vertical (blue), and vertical smoke over horizontal (red). Source image from flickr user aubergene.*

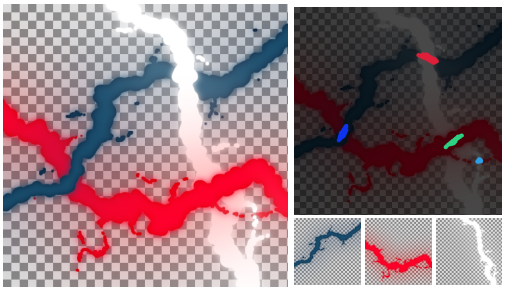

Figure 10: *Three lightning bolts interleaved using optimization-based soft stacking.*

5. Optimization-Based Editing

Though remapping stacking coefficients with a brush is direct and intuitive, sometimes an artist may wish to provide a more sparse set of inputs (e.g. "Layer 1 should be on top on the left of the image and layer 2 should be on top on the right"). In this section, we present a method for determining an optimal selection of stacking coefficients subject to such user-specified constraints. Our optimal stacking coefficients change smoothly and allow less-opaque layers to pass through each-other more readily. This results in smooth transitions between stacking orders (e.g., Figure 9).

5.1. Objective Function

As we now must begin to talk about whole groups of stacking coefficients, we introduce the notation $\sigma(\mathbf{x})$ – notice the lack of subscript – to denote the column vector of all the stacking coefficients at pixel $\mathbf{x}$:

$$\sigma(\mathbf{x}) \equiv \left[\begin{array}{ccc} \sigma_{1/\cdots/L}(\mathbf{x}) & \cdots & \sigma_{L/\cdots/1}(\mathbf{x}) \end{array} \right]^T \tag{10}$$

We build our optimization around a generalization of the discrete notion of *stacking consistency* [MP09]; specifically, we introduce $\bar{c}(\sigma, \sigma')$, which measures the inconsistency between two vectors of stacking coefficients σ and σ'. Intuitively, $\bar{c}$ penalizes for layers that pass through each-other, with the magnitude of the penalty being proportional to the product of the layers' opacities (in some sense, the probability that the layers collide). We give an exact definition of $\bar{c}$ in Appendix C.

Given a set of user-specified constraints, we compute the optimal selection of stacking coefficients by minimizing the inconsistency between the stacking coefficients at each pixel $\mathbf{x}$ and the stacking coefficients at each four-way neighbor $\mathbf{n}$:

$$\begin{aligned} \sigma \leftarrow \underset{\sigma}{\operatorname{argmin}} & \sum_{\substack{\mathbf{x} \\ \mathbf{n} \in \mathrm{Nbr}(\mathbf{x})}} \bar{c}(\sigma(\mathbf{x}), \sigma(\mathbf{n})) \\ & \text{subject to user-defined constraints on } \sigma \\ & \text{subject to } \sum \sigma = 1, \sigma \geq 0 \end{aligned} \tag{11}$$

5.2. Implementation

Our prototype is written in C++, and performs all computation on the CPU. It is not multi-threaded, though it does use an SVD routine provided by the Intel Math Kernel Library, which may elect to use multiple threads behind the scenes.

Our user interface is rudimentary. The user begins with the layers to be stacked arranged in a default order. (This order is required to select between multiple optimal answers – e.g., when there is a region of the image disconnected from any constraints.) The user can specify constraints by painting into the image with subsets of allowed stacking coefficients. These subsets are defined using a vocabulary similar to that used in our brush-based prototype to specify mappings – we constrain pixels only to take stackings which are not changed by the specified mapping. The user may then press a key to start the optimizer and – if she doesn't like the partial results displayed during the process – can opt to cancel the optimization in order to paint more constraints.

Our objective function (11) is a quadratic optimization with inequality constraints. Unfortunately, the scale and sparsity of the system make it infeasible in the off-the-shelf quadratic program solvers we have tried.

Instead, we use a form of gradient descent; alternating steps of successive over-relaxation (which bring us closer to the global, unconstrained minimum) and constraint enforcement. Steps are taken until a fixed limit is reached or the maxiumum change drops below a specified threshold. For examples in this paper we use a limit of 500 iterations and a threshold of $\frac{1}{512}$.

To aid in convergence, we start by scaling down our layers and constraints and iterating on this coarse version of the problem, then project this solution to finer and finer levels – refining after each projection (Figure 11). This is similar to a multigrid approach, except that we never return to the coarse level for further iteration on residual error.

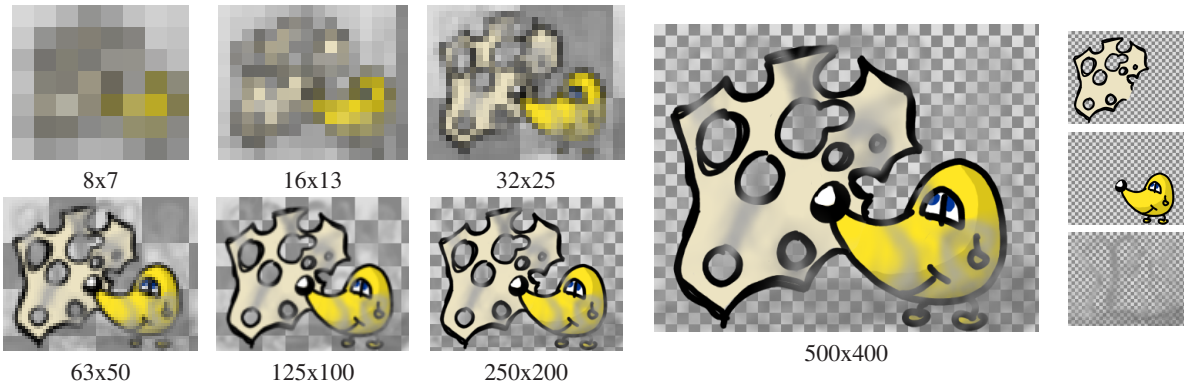

Figure 11: ***Left:*** *Our optimizer solves at progressively higher resolutions.* ***Right:*** *The final image and source layers.*

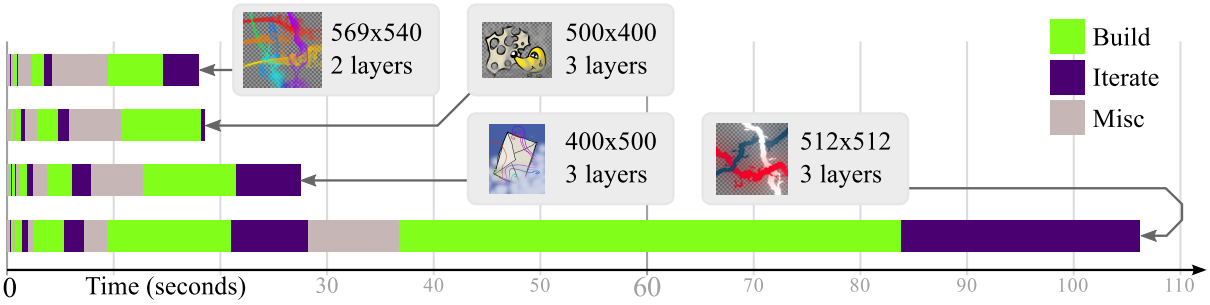

Figure 12: *Optimizer runtime for the examples in this paper. Runs are color-coded by type of work. "Build" is building the system, "iterate" is solving, and "misc" includes upsampling, downsampling, and rendering intermediate results. Each build/iterate pair corresponds to a level of our multi-resolution solver – so, for example, even though the lightning example takes 106 seconds to run, a 1/4 resolution version of the result is available in under 10 seconds.*

An additional advantage of the coarse-to-fine approach is that users can attain low-resolution feedback quickly, even if full-resolution feedback is a long time off.

We show the run-time for our optimizer on a several-year-old workstation-class computer (Core2 Quad Q6600) in Figure 12. The optimization examples in this paper all took under 30 seconds to run fully, except for the lightning result shown in Figure 10, which took 106 seconds. This dramatically longer runtime is likely due to the many nearly transparent pixels, which make constructing $\bar{c}$ less numerically stable.

In all cases, the solution time appears to be dominated by bookkeeping and system construction rather than by the iterative solution. This suggests that caching or approximation of $\bar{c}$ (which is currently constructed per-pixel-pair) may speed the optimization process. Additionally, a more sophisticated solution technique (e.g. algebraic multigrid) could result in better convergence and thus reduce solution time.

5.3. Results

Our optimization-based approach provides reasonable stacking order interpolation in a number of situations. It is useful when interleaving layers with volumetric glows (Figure 10, right), or smoke-like layers – both photographic (Figure 9) and hand-drawn (Figure 11).

One downside of our multi-resolution approach is that thin features merge at lower resolutions – making it hard to have alternating stacking orders on closely-spaced parallel lines. Such a difficulty arose when stacking Figure 13, with the consequence that more constraints were required to achieve the desired solution.

6. Comparison to Existing Methods

Our continuous stacking formulation is able to represent all possible mixtures of sub-pixel stackings. This makes our method more flexible than existing one-stacking-per-pixel approaches. However, in instances where the desired stacking order is discrete, systems like Local Layering [MP09] allow users to specify stackings quickly, without having to paint over large areas with constraints, manually chase implications of order adjustments, or – in the case of our optimization-based stacking prototype – wait minutes for a result. In the future, a hybrid approach could layer our brush-based stacking adjustments atop a base discrete stacking editable with Local Layering to provide the benefits of both.

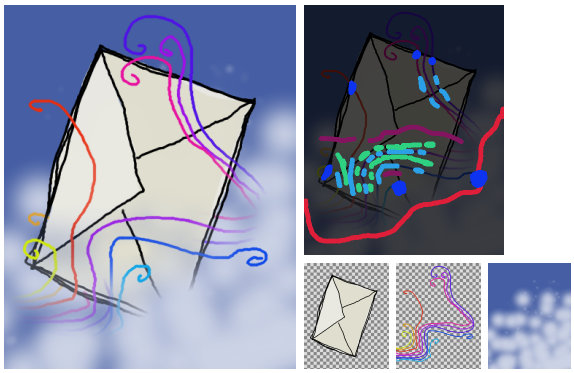

Figure 13: *A difficult case for our optimizer. The thin features in the rainbow merge at finer solution levels, so the user was forced to paint many constraints to achieve the interleaving he was interested in.*

Continuous stackings can be created in conventional image editing programs by cloning from various stacking orders into a final image.[¶] This is equivalent to painting in our brush-based prototype with mappings that select exactly one order; our system allows significantly more flexible constraints (e.g., "put *a* in front of *b* and *c*, but preserve their relative order"). Additionally, our system allows the transfer functions and contents of source layers to be changed freely – something that requires, at best, a large amount of manual copying and pasting in a conventional image editor.

7. Closing Remarks

We started this paper by generalizing per-pixel stacking orders to stacking coefficients. These stacking coefficients are both necessary and sufficient to represent sub-pixel stacking complexity with arbitrary layer transfer functions.

We then demonstrated two approaches for the manipulation of stacking coefficients. The brush-based stacking method has the advantage of allowing direct control, providing rapid feedback, and scaling well. On the other hand, the optimization-based stacking approach can provide a globally smooth result given sparse constraints, though it isn't practical for large numbers of layers.

One can perform an alternate derivation of stacking coefficients in order to represent correlated coverage information. In this case, stacking coefficients are introduced for all orders of all subsets of layers to account for portions of each pixel which are covered by each layer. For example, with three layers, additional coefficients $\sigma_\emptyset, \sigma_1, \sigma_2, \sigma_3, \sigma_{1/2}, \sigma_{2/1}, \sigma_{1/3}, \sigma_{3/1}, \sigma_{2/3}$, and $\sigma_{3/2}$ would be required at each pixel, and additional equality constraints on the stacking coefficients would be required (the sum of all σ mentioning layer i must equal the opacity of i). Such a derivation might make an interesting basis for future work in this area.

Overall, our notion of soft stacking extends the current state-of-the-art in layer stacking by providing a continuous-domain extension of current discrete techniques. In addition to providing artists all the flexibility of existing approaches, stacking coefficients allow artists to manipulate layers – like hair or fog – as if they had sub-pixel stacking complexity.

[¶] Additionally, compositions that look like (but are not) continuous stackings can be created by manipulating layer alpha channels.

References

[ASP07] ASENTE P., SCHUSTER M., PETTIT T.: Dynamic planar map illustration. *ACM Transactions on Graphics 26*, 3 (2007), 30. 2

[BRV*10] BRUCKNER S., RAUTEK P., VIOLA I., ROBERTS M., SOUSA M. C., GRÖLLER M. E.: Hybrid visibility compositing and masking for illustrative rendering. *Computers & Graphics*, 34 (2010), 361–369. 2

[BSS*11] BARAN I., SCHMID J., SIEGRIST T., GROSS M., SUMNER R. W.: Mixed-order compositing for 3d paintings. *ACM Transactions on Graphics 30*, 6 (2011). 2

[Duf85] DUFF T.: Compositing 3-d rendered images. *Computer Graphics (Proceedings of SIGGRAPH 85) 19*, 3 (1985), 41–44. 1, 2

[GIM10] GIMP DOCUMENTATION TEAM, THE: Gimp user manual, Section II.8.2: Layer modes. http://docs.gimp.org/en/gimp-concepts-layer-modes.html, 2002–2010. 2

[IM10] IGARASHI T., MITANI J.: Apparent layer operations for the manipulation of deformable objects. *ACM Trans. Graph. 29* (July 2010), 110:1–110:7. 2

[Med09] MEDIACHANCE: Real-Draw PRO push-back tool. http://www.mediachance.com/realdraw/help/pushback.htm, 2001–2009. 2

[MP09] MCCANN J., POLLARD N. S.: Local layering. *ACM Transactions on Graphics (SIGGRAPH 2009) 28*, 3 (Aug. 2009). 2, 7, 8, 10

[OW91] ODDY R. J., WILLIS P. J.: A physically based colour model. *Computer Graphics Forum 10*, 2 (1991). 2

[PD84] PORTER T., DUFF T.: Compositing digital images. *Computer Graphics (Proceedings of SIGGRAPH 84) 18*, 3 (1984), 253–259. 2

[RID10] RIVERS A., IGARASHI T., DURAND F.: 2.5d cartoon models. *ACM Trans. Graph. 29* (July 2010), 59:1–59:7. 2

[SL98] SNYDER J., LENGYEL J.: Visibility sorting and compositing without splitting for image layer decompositions. In *Proceedings of SIGGRAPH 98* (New York, NY, USA, 1998), ACM, pp. 219–230. 2

[Smi95] SMITH A. R.: Alpha and the history of digital compositing. In *Microsoft Technical Memo #7* (1995). 2

[SSGS11] SCHMID J., SENN M. S., GROSS M., SUMNER R. W.: Overcoat: an implicit canvas for 3d painting. *ACM Trans. Graph. 30* (August 2011), 28:1–28:10. 2

[SSJ*10] SÝKORA D., SEDLÁČEK D., JINCHAO S., DINGLIANA J., COLLINS S.: Adding depth to cartoons using sparse depth (in)equalities. *Computer Graphics Forum 29*, 2 (2010), 615–623. 2

[Wil06] WILEY K.: *Druid: Representation of Interwoven Surfaces in 2 1/2 D Drawing*. PhD thesis, University of New Mexico, 2006. 2

Appendix A: The Necessity of Alpha

To read back α_i from a black box compositing function, set the background color to 0 and layer transfer functions as follows:

$$f_j(\mathbf{c}) \equiv \begin{cases} 1 & \text{if } j = i \\ \mathbf{c} & \text{otherwise} \end{cases} \tag{12}$$

This makes all other layers besides i effectively transparent, so the final composited color is just the probability of interaction with layer i – that is, α_i.

Appendix B: The Necessity of Stacking Coefficients

To read back $\sigma_{l_1/\cdots/l_L}$ from a black box compositing function, set all layer opacities to 1, the background color to L, and the transfer functions as follows:

$$f_j(\mathbf{c}) \equiv \begin{cases} i-1 & \text{if } j = l_i \text{ and } \mathbf{c} = i \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

This setting counts from L to 1 if a sample interacts with layers in order $l_1/\cdots/l_L$, and yields 0 if the sample interacts in any other order. As every sample must interact with every layer, the final composited color will be $\sigma_{l_1/\cdot/l_L}$.

Appendix C: Measuring Continuous Consistency

In this appendix, we describe how to extend the discrete notion of *consistency* introduced by McCann and Pollard [MP09] to the continuous domain. Briefly, in the discrete domain, a consistent stacking is one in which layers in adjacent pixels appear in the same order unless they are fully transparent in one or both of the pixels.

In the continuous domain, we introduce a function $\bar{c}$ (for "not consistent"), which measures the inconsistency between vectors of stacking coefficients in a continuous way.

Intuitively, $\bar{c}(\sigma,\sigma')$ measures the difficulty one would encounter in re-stacking a pixel with the mix of stacking orders given by σ into one with stacking orders given by σ'.

Re-stacking. Consider the task of transforming a pixel with the mix of stackings σ into one with the mix of stackings σ'. One might perform this re-arrangement by selecting some portion of the pixel stacked in a given order, say $1/2/3$, and exchanging two adjacent layers, say 1 and 2, so that that portion of the pixel is now stacked $2/1/3$; continuing in this way, swapping various layers, one could eventually transform the stacking coefficients of the pixel to σ'.

In the general case, we'll denote the portion of the pixel where we swap p_i and p_{i+1} in the stacking $p = p_1/\cdots/p_L$ by $\delta_{i,p}$. Notice that when performing a set of swaps, stacking coefficient σ_p will lose value to neighboring stacking $q(i) = p_1/\cdots/p_{i+1}/p_i/\cdots/p_L$ via swap $\delta_{i,p}$ and gain value from it through swap $\delta_{i,q(i)}$ (Figure 14). With this in mind, we

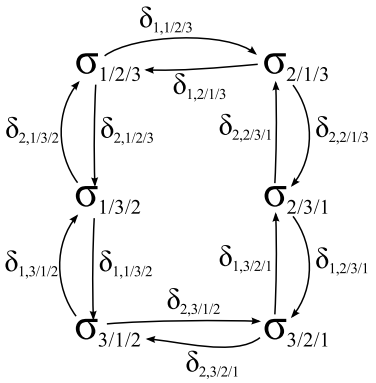

Figure 14: *Using swaps to transform stacking coefficients* (14). *To move from* $\sigma_{1/2/3} = 1$ *to* $\sigma'_{3/2/1} = 1$, *one could set* $\delta_{1,1/2/3} = 1$, $\delta_{2,2/1/3} = 1$, *and* $\delta_{1,2/3/1} = 1$. *(Though, depending on* α, *this may not be minimal* (16).*)*

can write down what it would mean for a set of swaps to transform stacking coefficients σ into σ':

$$\forall p: \quad \sigma'_p = \sigma_p + \sum_{i=1}^{L-1} \delta_{i,q(i)} - \delta_{i,p}$$
$$\text{where } q(i) \equiv p_1/\cdots/p_{i-1}/p_{i+1}/p_i/p_{i+2}/\cdots/p_L \tag{14}$$

Difficulty. Of course, with all this passing of layers through other layers, some collisions are bound to occur. Appealing to our earlier definition of opacity (3) as the probability of a random sample inside a layer being solid, we can write the expected colliding area when passing layer i through layer j in a region of area a:

$$\alpha_i\alpha_j a = \mathbb{E}(\text{colliding area passing } i \text{ through } j \text{ in region of area } a) \tag{15}$$

Putting it together. When re-arranging, it makes sense to pick a set of swaps that does not have much colliding area. Thus, we take minimum squared expected colliding area as our measure of inconsistency:

$$\bar{c}(\sigma,\sigma') \equiv \min_{\delta} \sum_{p,i} \left(\alpha_{p_i}\alpha_{p_{i+1}}\left(\delta_{i,q(i)} - \delta_{i,p}\right)\right)^2 \quad \text{subject to (14)} \tag{16}$$

As $\bar{c}$ is evaluated "between pixels" we use the average α values of pixels to which σ and σ' belong.

Notice that the sum of squared collision areas is a convex quadratic function with equality constraints, and is, therefore, minimized by a linear transformation of σ and σ'. Thus, $\bar{c}$ itself is a convex quadratic function of the concatenation of σ and σ', which is convenient for our optimization.