Local Manipulation of Image Layers Using Standard Image Processing Primitives

Niranjan Mujumdar
Don Bosco Institute of Technology
niranjanpm@gmail.com

Sanju Maliakal
Don Bosco Institute of Technology
sanjumaliakal@gmail.com

Sweta Malankar
Don Bosco Institute of Technology
sweneera@gmail.com

Satish Kumar Chavan
Don Bosco Institute of Technology
satyachavan@yahoo.co.in

Parag Chaudhuri*
Indian Institute of Technology Bombay
paragc@cse.iitb.ac.in

ABSTRACT

In any modern, standard image manipulation program, images are made up of multiple layers ordered on top of each other in a user defined sequence. A layer is a collection of graphical objects (like masks and image patches). Each layer can be edited separately and then all the layers are composited together to get the final image. In some images, however, we need to change the order of stacking at multiple areas of overlap for the same set of layers without creation and manipulation of individual layers at each position of overlap. This problem is solved by the concept of Local Layering as given by McCann and Pollard [10]. Their technique was, however, presented as a standalone idea, making its use difficult in standard image processing pipelines.

In this paper, we present a novel implementation of local layering using a combination of standard image processing primitives. We show that it is possible to locally align layers while simultaneously continuing to also use the global alignment of layers via an efficient use of image masks. All our ideas are implemented as a plug-in addition to the GIMP, however, the algorithms we present are general and implementable in any standard image processing pipeline.

Categories and Subject Descriptors

I.3.3 [**Computer Graphics**]: Picture/Image Generation

Keywords

Local layering, Image Masks, Image Matting

*Corresponding author

ICVGIP '10, December 12-15, 2010, Chennai, India

1. INTRODUCTION

In a conventional image manipulation program, images can be made of a stack of layers. Each layer is independently editable. The final image is obtained by compositing and blending this stack of layers. The layer stacking and ordering has a global scope over the image, i.e., if layer M is stacked above layer N in one part of a image, then the ordering is true for all parts of the image. Any modern image editor like the GNU Image Manipulation Program or GIMP [8] allows the global reordering and manipulation of these layers.

In order to create complex images like that of a weave pattern (see Figure 2), where many threads overlap each other in many different places in different orders, duplicate layers have to made for each thread for each point of overlap and the global ordering has to be rearranged to get the weave pattern. It is also impossible to create for example, a cyclic ordering of layers using global ordering (see Figure 1). An elegant solution to this problem was presented in the work on local layering [10] that for the first time allowed the ordering of the layers to be done at a local level rather than at a global level. The idea was demonstrated via a standalone implementation that was not integrated into any standard image processing pipeline. This makes its use cumbersome and limited.

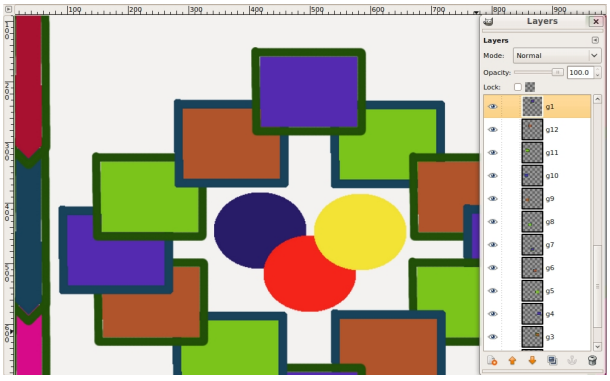


Figure 1: A standard global ordering of layers makes a cyclic ordering of layers impossible without duplication and splitting of layers.

Contributions: We present novel techniques and algorithms

that allow the use of image masks to efficiently implement local layering of images in any image processing pipeline. It also maintains the global ordering of images as defined by the user while simultaneously allowing local layer manipulation wherever needed. Our ideas have been implemented as a plug-in to the GIMP, a widely used open source image manipulation tool. This not only eases the learning curve of using local layering significantly, it also makes it easily available to users of standard image editors as a part of their toolkits. Our algorithms are able to handle large images efficiently providing interactive feedback as the layer ordering is changed locally. Our methods also allow the local layering information to be saved using standard image processing primitives that are saved as a part of the image, so that the editing can be resumed at a later time. Even though we have implemented our methods as a plug-in to the GIMP, all our algorithms are general and independent of the GIMP.

It should be noted that the basic algorithm we follow for maintaining and updating the local layering information at each local overlap region is the same as presented in the original version. Also, we do not claim that we are more efficient that the standalone prototype implementation provided as a part of the same work. Our implementation of local layering using standard image-processing primitives like image masks is novel and we demonstrate that it is efficient and correct using a variety of examples.

Figure 2 shows a complex weave pattern with alphabets woven in-between the threads. Notice that all the horizontal threads form one layer, the vertical threads form the second layer while the letters are in a third layer. We are able to locally edit their stacking order to create the pattern shown in the image. Our methods are also validated by the fact that we can reproduce all the results produced by [10] using our techniques. Figure 3 shows one such example.

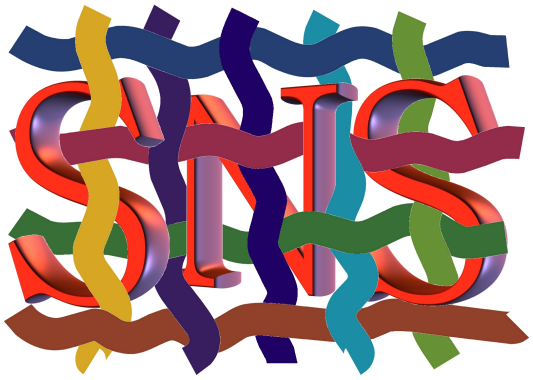


Figure 2: A complex weave pattern involving cloth strands and letters created with our plug-in.

The rest of the paper is organized as follows. We start with a brief overview of previous work in Section 2. We continue with a brief explanation of the original local layering technique in Section 3. Section 4 gives the details about our implementation of local layering. In Section 5 we present our algorithm for reconstructing local layer ordering from layer masks when the image is reloaded after saving and closing the image. Section 6 presents a discussion of the results we have produced using our method. Finally, we conclude in section 7.

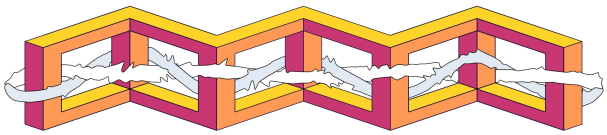


Figure 3: Intertwined threads result recreated with our plug-in. (Original layers are courtesy [10])

2. BACKGROUND

The idea of compositing partially transparent *cels* on which objects are drawn or painted is a well known technique in traditional hand-drawn animation. The use of alpha values in compositing whereby transparency is viewed as part of each $RGBA$ pixel was introduced to graphics by [12]. Pixels thus, can be thought of as representative of separate objects on raster images instead of belonging to global image-sized layers. This and other ideas behind compositing of images and videos have matured over the years and are used in many standard image processing programs (for e.g., [8], [3], [2]).

On the other hand, vision researchers have tried to decipher the ordering of layers that make up a scene, given an image of the scene [11]. Scenes depicting the real world have layers that can be segmented using different features depending on relevance and the purpose of performing the segmentation. Adelson and Wang [1], for e.g., describe how to decompose an image into layer using motion analysis. Separation of image regions into foreground-background layers forms the starting point for of many tracking and surveillance algorithms [9]. Given all object contours, a stack of layers corresponding to front and back-facing regions of the object can be constructed and used for modeling [7].

Layers corresponding to separate objects in a 3D scene may give rise to occlusion cycles. This makes global ordering of these layers impossible unless additional depth information is used to provide a local stacking [5]. Alternatively, layers can be modified by editing out occluded pieces, as presented by Snyder and Lengyel [13]. Their approach is to compile an occlusion graph into a series of compositing operations, may be useful to accelerate display of our list-graphs. However, since they rely on operations that effect the entire image, their technique cannot handle the situation where two layers M and N must be composited, with M over N in one place and N over M in another.

Local layering [10] presents a way to address this problem by maintaining the layer stack ordering in a *list-graph* that stores the adjacency graph of regions of overlap of graphical objects. Building on this structure, they define a consistent stacking, and prove that the local order-adjustment operators are both correct and sufficient to navigate the space of consistent stackings. This is similar, in spirit, to a planar map as used in [4]. Wiley [14] presents a vector-graphics drawing system, Druid, that represents regions and their stacking as valid over/under labellings of the intersections of a set of boundary curves. However, the Druid program uses a worst case exponential-time search whereas the local layering approach of [10] is polynomial-time. IN more recent work, Igarashi and Mitani [6] layer operations for single-view 3D deformable object manipulation, in which the user can

control the depth order of layered 3D objects resting on a flat ground with simple clicks and drags, as in 2D drawing systems.

Our implementation of the list-graph data structure is based on an efficient manipulation of layer mask pixel values associated with every image layer. Since layer masks are commonly available in all common image editors like Adobe Photoshop [2] and the GIMP [8], our algorithms are generic and implementable in any standard image processing pipeline. We further show that image masks can be used to store and retrieve local layer ordering if the image is saved in the native format of the editor (for e.g., *PSD* for Adobe Photoshop, *XCF* for the GIMP). For this purpose, we present a novel algorithm to reconstruct a consistent list-graph from saved image mask data.

3. LOCAL LAYERING

With local layering, layers can be ordered just as one would order paper cut-outs, weaving and overlapping but never passing through one-another. Local stacking is represented using a list-graph and the layer flipping operators used to change the layer ordering in local regions, as presented in [10]. We present a brief explanation of this process. For further details the reader is encouraged to refer to [10].

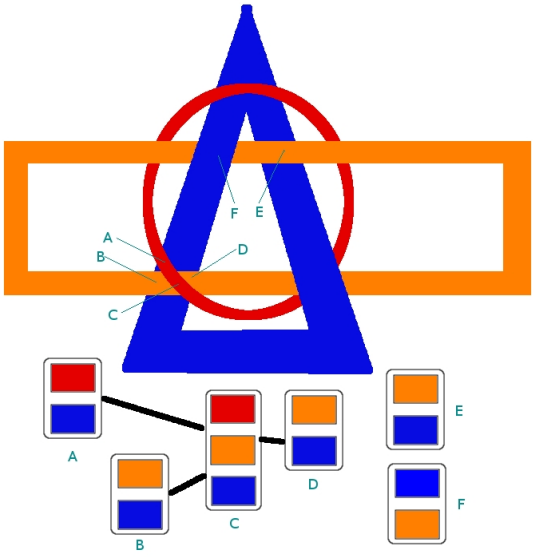


Figure 4: Illustration showing the list graph: 6 overlapping regions are shown. 4 are adjacent and connected, while 2 regions are separate.

In order to construct the list-graph G, the image is first partitioned into regions where the layers overlap each other. Each of these regions of the image has the same stacking of objects. Now, for each such region, a list-graph vertex is created which contains the ordering of the layers in that region, in a list L. The vertices between two neighbouring regions of overlap (i.e., regions which share an edge in the image) are linked together by an edge in the graph. The list L at each vertex of the list-graph G may be initialized to a predefined global order. Consistency of list-graphs is maintained to ensure that the objects do not pass through each other in the resulting image. An example of a list graph is shown in Figure 4 with 6 overlapping regions marked. The stacking of layers in each region is shown below. 4 of these regions are adjacent and hence, connected. This data structure is created and maintained in memory in order to allow the local layer manipulations to be done efficiently.

Editing of layer stacking order is done via a pair of operators, flip-up and flip-down. These operators are similar to the move-up and move-down operators of a global layering environment.

At any region of overlap and its corresponding list, L, a call to the flip-down operator (for e.g., $Flip{-}Down(L, R, B)$) for a selected pair of layers R and B, moves the layer R below the layer B while maintaining the consistency of the list for the region in question and all adjacent regions where the same layers may overlap. The flip-up and flip-down operators run in polynomial time and their termination, soundness, invertability and sufficiency properties are proven in [10]. Figure 5 illustrates the working of the flip-up and flip down operators. Note that a single flip-up or flip-down call may result in multiple layer swaps, as shown in the figure.

4. LOCAL LAYERING USING MASKS

In this section we present our method of implementing local layering using masks. We represent objects as independent variable sized layers. A layer is said to exist at pixel locations where it has non-zero alpha values. Regions of overlap are calculated by first creating an image sized Boolean bit-field array. A pixel position which has layers 2, 5 and 7 present will have the 2nd, 5th and 7th Boolean array location set. Based on that, a layer code is calculated for each pixel. Connected pixels having the same layer code are grouped to form a region of overlap by using a region-growing algorithm. For region growing, we consider 4-connectivity of pixels. Each region of overlap is also given a different numerical tag and is stored in an image sized tag array. This is similar to what is done in the prototype implementation given by [10].

A list is created to store the layers present for a particular region and also their order with respect to other layers. Each region is associated with such a list. The lists are connected using adjacency of regions to form a list-graph.

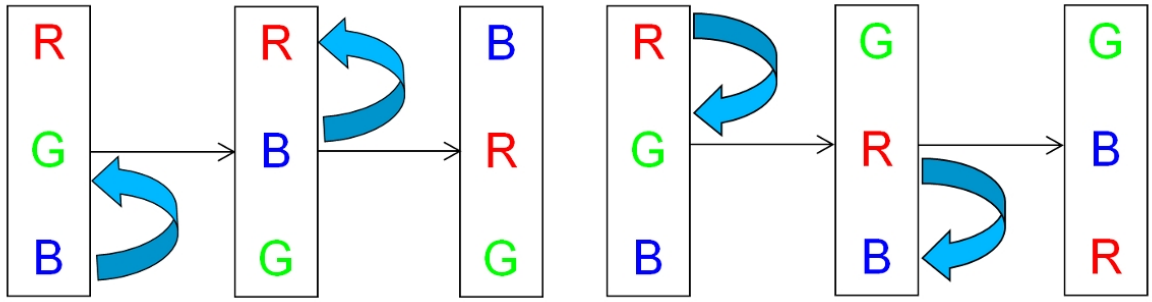
The flip-up and flip-down operations are used to change the local stacking order of layers. For each flip operation on a region, corresponding flip operations are carried out on adjacent regions to maintain consistency. The list-graph is updated with each flip operation.

We also provide an Undo operator which is used to undo the last flip operation. The action of the flip-Up operator may be inverted by a sufficient number of calls to flip-Down, and vice-versa.

Efficiency considerations include the effective use of the GIMP Tile structure that caches local image pixel regions in the memory and allows them to be processed faster.

4.1 Masks

The local order composition is done by using masks. A layer mask is added for each layer present. The layer mask has the same size and same pixel number as the layer to which it is attached. The mask is a set of pixels in grayscale on a value scale from 0 to 255. Here 0 implies complete



(a) $Flip - Up(L, B, R)$ (b) $Flip - Down(L, R, B)$

Figure 5: Illustration showing the flip operation. (a) Flip-Up(L,B,R) : B is flipped with G first, and then with R to get the flipped output. (b) Flip-Down(L,R,B) : R is flipped with G first, and then with B to get the flipped output.

transparency and a value of 255 implies complete opacity. We consider that at any given pixel position *only one* layer has its mask painted white and the rest of the layers have its masks painted black. Thus, only the *local* top-most layer is made visible in a region and the rest are hidden. An example of the masks that get created when the local-layering is initialized can be seen in Figure 6.

On calls to flip-up and flip-down the changes with respect to the ordering of layers for the regions are represented in the list graph. The changes are reflected back in the layer masks by repainting them based on the above condition and list graph contents.

5. RECONSTRUCTING THE LIST-GRAPH FROM MASKS

If the image editing process is interrupted for some reason, the editing has to be saved and resumed from the saved state at a later time. The original local-layering implementation given by [10] provides easy way of doing this that can be integrated into existing image processing pipelines. In this section, we present a novel algorithm which reconstructs the list-graph from the masks and layer data, thus, allowing us to save and retrieve the local layering edits at a later time.

We assume that the layers are not moved between sessions. Thus, we can calculate the layers present in a list and the edges of the list-graph as explained in the previous section. This data, therefore, does not change and hence is 'known'. The local layer stacking for a list is not known since it can change across sessions.

5.1 Additional Data Structures

We use additional data structures in the forms of pair-lists to keep track of layer stacking information for each region for reconstructing the list-graph. A global pair queue is used to queue pairs that are to be processed. These are explained below.

Definition 1. Pair-list - All known layer stacking information is stored in the form of pairs (a, b) indicating that layer a is above layer b. For each region with corresponding list L, we have a list of known pairs, which is called as a pair-list. The maximum number of pairs in a pair-list for a region is $\binom{n}{2}$, where n is the number of layers in that region.

A pair-list P is said to be adjacent to another pair-list P' if their corresponding lists, L and L' are adjacent in the list-graph G.

5.2 Algorithm

1. **Initialize List-Graph**: List-graph edges are known. The layers present at each region (layers of a list) are also known. The list is initially empty (i.e., the ordering is not known). Global ordering of layers is also known.

2. **Find Topmost Layer**: We know the topmost layer for each region. The topmost layer will have its mask value set as opaque (or 255), while all other layers will have the mask value transparent (or 0). This is ensured by our image mask based implementation of the list graph (see Section 4.1). For each region, we find out the top most layer and put it in the corresponding list, at the top.

3. **Find Next Layer**: For all regions, find the first layer with a transparent mask value and put the pair formed by the first and second layer in the pair-list. We put all such pairs into the pair queue and the pair-list corresponding to the region.

4. **Build Pair Queue**: For all other regions, we find the next subsequent layers after the first and second layers, in any order. Since we know the top most layer for each region, we have pairs $(a, b_1), (a, b_2)...(a, b_n)$, where a is the top layer for that region and $b_1, b_2...b_n$, are all the layers below a. We put these pairs into the global pair queue and the pair-list for the region. At the end of this step, we have some pairs in the pair queue and we know to which pair-list (and region) do these pairs belong.

5. **Process the Pair Queue**: We take one pair from the queue at a time and inspect the region, R, to which it belongs. If the two layers forming the pair exist in region neighbouring R, then we check the pair-lists of the neighbouring regions. If the pair is not present in these pair-lists, we add the pair to the pair-lists of the neighbouring regions as well as to the pair queue. The algorithm for the procedure is given below.

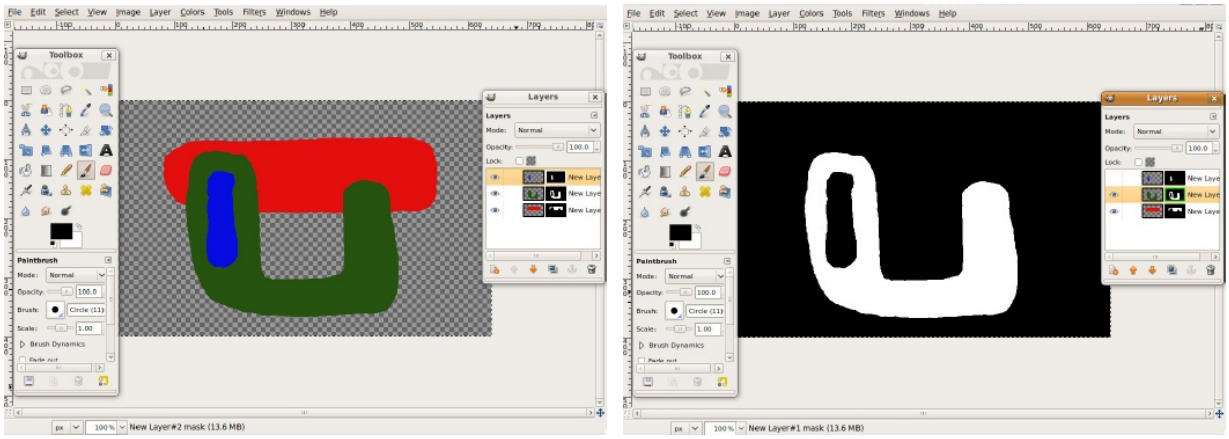


Figure 6: The left image shows the original image loaded with three global layers. The right image shows the mask created for the layer with the green U stroke. Other masks can be seen in the Layers dialogue box on the right of each image.

Algorithm 1 $Process - pair(L, P, x, y)$

```
for (P', L' adjacent to P, L) do
  if (x ∈ L' and y ∈ L') then
    if ((x, y) ∉ P') then
      Put (x, y) in P'
      Push (x, y) into Pair-Queue
    end if
  end if
end for
```

6. **Get a near-complete pair-list** - Once the pair queue is processed completely, we get either a *near-complete* pair-list or a *complete* pair-list for each region. This is illustrated in Figure 7

   Definition 2. Near-complete pair-list - A pair-list generated from the masks or layers which are not sufficient to generate the complete layer stacking order for that list is called a near-complete pair-list. Conversely, when the pairs generated are sufficient to build the list-graph, it is called a complete pair-list. A complete pair-list has $\binom{n}{2}$ pairs, where n is the number of layers in that list.

   All near-complete lists need to be made complete before ordering can be finalized. We complete the near-complete lists by using the global layer stacking available. The pairs which are not part of the pair-list are found and the order of the two layers is decided on the basis of the global layer stacking. Since the global order applies to the whole image, consistency is maintained when completing the pair-lists.

7. **Build the list-graph** A complete pair-list is necessary and sufficient to build the list-graph. For a pair-list P corresponding to list L, the position of any layer x in the stacking is decided by the number of pairs $(x, y) \in P$. The position of layer x is equal to *Total number of layers in* L $-$ *No. of pairs* $(x, y) \in P$. For example, a layer placed 3rd in a region having 7 layers will have 4 pairs associated with it in the pair-list.

This completes our algorithm for reconstructing the list-graphs from the mask/layer data. We now prove that a complete pair-list is necessary and sufficient to reconstruct the list-graph and that the reconstructed list-graph is always consistent, given the notion of consistency is the same as defined in [10].

THEOREM 1. $\binom{n}{2}$ *pairs for each list* L *are necessary and sufficient to give stacking of all layers in that region with respect to each other, i.e., list-graph.*

PROOF. First, we show that for any list-graph G, $\binom{n}{2}$ pairs will be generated for each list $L \in G$. Consider list $L \in G$, where G is the list-graph. Let layer $x \in L$ be in the p^{th} position in the list L. Let the total number of layers in L be n. Then $\forall x \in L$, the number of pairs generated are $(n - p)$. Thus the total number of pairs for list L is

$$\sum_{p=1}^{n}(n-p) = \frac{n(n-1)}{2} = \binom{n}{2}$$

Conversely, we show that $\binom{n}{2}$ pairs for each list $L \in G$ gives the relative stacking order of all layers in a region with respect to each other. Let us consider we have $\binom{n}{2}$ pairs for each pair-list P corresponding to a list $L \in G$. $\forall x, y \in L$, $(x, y) \in P$ gives the layers y below x in the layer stacking. Let the number of such pairs be p. p indicates the number of layers y below x. Then,

$$\sum_{x=x_1}^{x_n} p = (n-1) + (n-2) + ... + 0 = \frac{n(n-1)}{2} = \binom{n}{2}$$

Thus, $\binom{n}{2}$ pairs gives us the relative stacking of all layers with respect to each other for all lists $L \in G$. □

THEOREM 2. *A List-graph re-constructed using the algorithm will always be consistent.*

PROOF. From definitions 1 and 2, a list-graph G is consistent if pair $(x, y) \in P \Leftrightarrow (x, y) \in P'$ and P, P' are adjacent. Proceed by contradiction. Let us assume a pair $(x, y) \in P$ and pair $(y, x) \in P'$ and P, P' are adjacent. When $Process - pair(P, L, x, y)$ is called, it checks in all

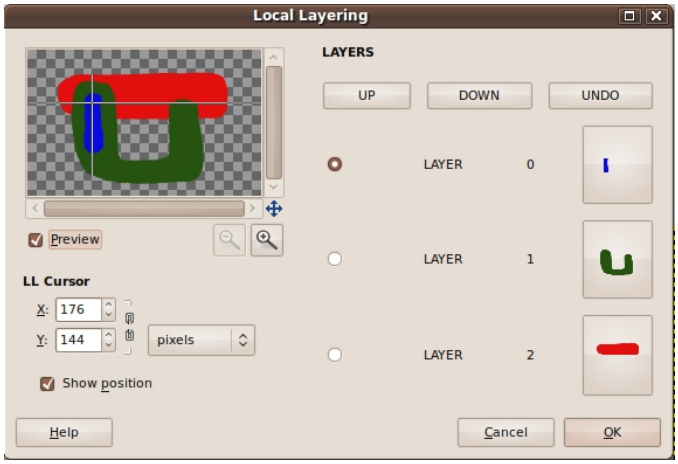


Figure 8: Our intuitive graphical user interface for local-layering.

neighbouring pair-lists if the pair is present. If not, (x, y) is added to P'. In this case, $(x, y) \notin P'$, i.e., it is added to P'. Thus, $(x, y) \in P'$ and $(y, x) \in P'$, which is not possible from the definition of a pair.

Now, consider a near-complete pair-list which is completed using the global layer ordering. Since the global ordering applies to all regions in the image, $(x, y) \in P \Leftrightarrow (x, y) \in P'$, where P' is any other pair-list and (x, y) is a pair decided using the global ordering.

Thus, a re-constructed list-graph G will always be consistent. □

It should be noted that even after the reconstruction there is no *visible* change in the appearance of the image - it is only that the layers below the top layer may have a different order than what was saved. This does not change the look of the image as the topmost layer in every overlapping region is known and reconstructed correctly. This also does not hinder editing of the image in any way, and the user can continue his/her editing right from where they left.

For example, Figure 7 shows an image with 5 layers and its corresponding list graph. In list C, the order originally is {Red, Orange, Blue, Green, Yellow}. The global layer order is {Blue, Green, Red, Orange, Yellow}. When the list-graph is re-constructed using our algorithm, the order is {Red, Blue, Green, Orange, Yellow} in list C, which is not the original order but has no effect on the visible image.

This is only possible, however, when the image is stored in the native format of the image editor being used. In case of GIMP it is the *XCF* format, while in case of Photoshop, it would be the *PSD* format. This is so because standard image formats like JPEG, PNG, GIF, etc. do not support image masks and layers. Our methods and algorithms are, however, independent of the internal details of the actual file format used and hence can be implemented on *any* image processing pipeline that supports layer masks.

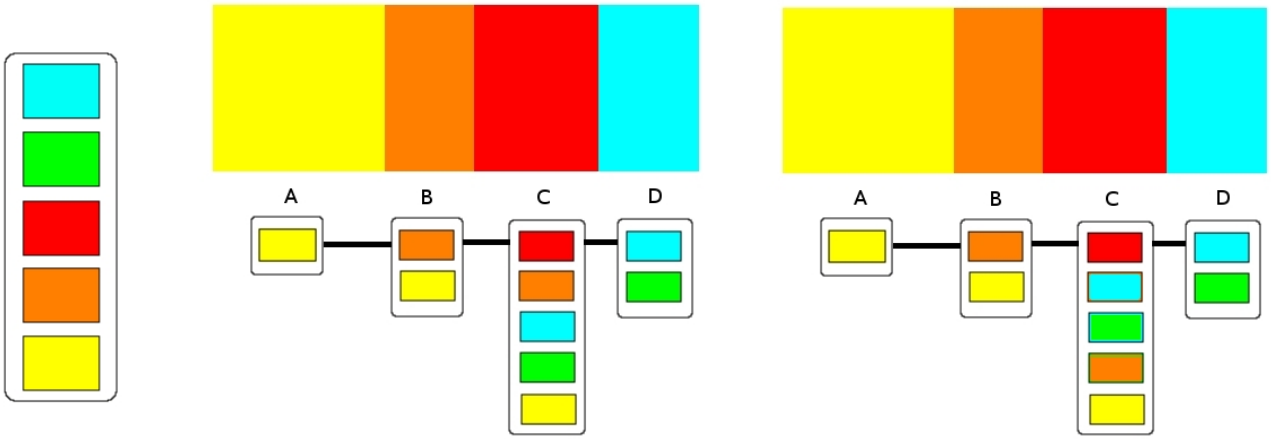
6. RESULTS

Our local layering implementation can be used as a general purpose tool to create images which contain object which are inter-twined around each other. Figure 3 shows an example involving a rectangular frame, a straight twine and a curved twine. We provide a convenient interface that allows the user to obtain the intertwining effect with just a few mouse clicks.

In particular the plug-in to the GIMP we have implemented, can handle up to 32 (global) layers (maximum number of layers allowed for a GIMP plug-in) through the GIMP procedural database consistently and efficiently. The flips are done at interactive rates even for images with a number of substantially sized layers. Figure 9 shows an image with 32 layers, each of size 1 megapixel. The entire direction of overlap is changed very conveniently to give a pattern which points in the opposite direction with respect to the original image. Also the circular interleaved stacking in the output forms an impossible figure. This cannot be obtained through global layering but is generated easily using Local Layering.

Figure 10: Top: The image prior to manipulation of local image layers. Bottom: Image after the layers have been edited locally.

Figure 10 shows another example of an image created eas-



(a) Global Order (b) Originally (c) After Reconstruction

Figure 7: Illustration showing reconstructed list-graph of image with a near-complete pair-list. The local stacking order for region C is different after reconstruction.

ily by local manipulation of layers. The image on the left shows the original global layers in the image, while the image after editing is shown on the right. Notice how the human figures intertwine through the letters.

A supplementary video submitted with the paper shows the working of our plug-in inside the GIMP through various examples. An example of saving an editing session and resuming the session later from local layer stacking reconstructed from layer masks is also shown.

The source code for our plug-in can be downloaded from `http://llgimp.sourceforge.net/`.

7. CONCLUSION

We have presented a novel implementation of local layering that is based entirely on standard image processing primitives. This makes local layering immediately available to every popular image processing pipeline that work with image masks. To illustrate our ideas, we have also presented a working open-source plug-in for the GNU Image manipulation Program (GIMP). Our algorithms are able to handle large images efficiently providing interactive feedback as the layer ordering is changed locally. Our methods also allow the local layering information to be saved using standard image processing primitives that are saved as a part of the image, so that the editing can be resumed at a later time.

Future work would focus on allowing the layers to move and use local matting to facilitate animation, as shown in [10] and extending the local layering concept to video.

8. ACKNOWLEDGMENTS

We would like to thank Jim McCann and Nancy Pollard for making the original source layers for all the images used in their work available. This greatly helped in the testing of our implementation. We would also like to thank the anonymous reviewers for their helpful and encouraging comments.

9. REFERENCES

[1] E. H. Adelson and J. Y. A. Wang. Representing moving images with layers. *IEEE Transactions on Image Processing*, 3:625–638, 1994.

[2] Adobe. Photoshop CS 5. *http://www.adobe.com/products/photoshop/*, 2010.

[3] Apple. Final Cut Pro. *http://www.apple.com/finalcutstudio/finalcutpro/*, 1999-2010.

[4] P. Baudelaire and M. Gangnet. Planar maps: an interaction paradigm for graphic design. In *Proceedings of CHI '89*, pages 313–318. ACM, 1989.

[5] T. Duff. Compositing 3-d rendered images. In *Proceedings of SIGGRAPH '85*, pages 41–44. ACM, 1985.

[6] T. Igarashi and J. Mitani. Apparent layer operations for the manipulation of deformable objects. *ACM Transactions on Graphics*, 29(3), 2010.

[7] O. A. Karpenko and J. F. Hughes. Smoothsketch: 3d free-form shapes from complex sketches. *ACM Transactions on Graphics*, 25(3):589–598, 2006.

[8] S. Kimball and P. Mattis. GIMP - the GNU image manipulation tool. *http://www.gimp.org/*, 1995-2010.

[9] S.-N. Lim, A. Mittal, L. S. Davis, and N. Paragios. Fast illumination-invariant background subtraction using two views: Error analysis, sensor placement and applications. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages 1071–1078, 2005.

[10] J. McCann and N. Pollard. Local layering. *ACM Transactions on Graphics*, 28(3):1–7, 2009.

[11] M. Nitzberg and D. Mumford. The 2.1-d sketch. In *International Conference on Computer Vision (ICCV)*, pages 138–144, 1990.

[12] T. Porter and T. Duff. Compositing digital images. *SIGGRAPH Computer Graphics*, 18(3):253–259, 1984.

[13] J. Snyder and J. Lengyel. Visibility sorting and compositing without splitting for image layer decompositions. In *Proceedings of SIGGRAPH '98*, pages 219–230. ACM, 1998.

[14] K. Wiley. Druid: Representation of interwoven surfaces in 2 1/2 d drawing. *PhD Thesis, University of New Mexico*, 2006.

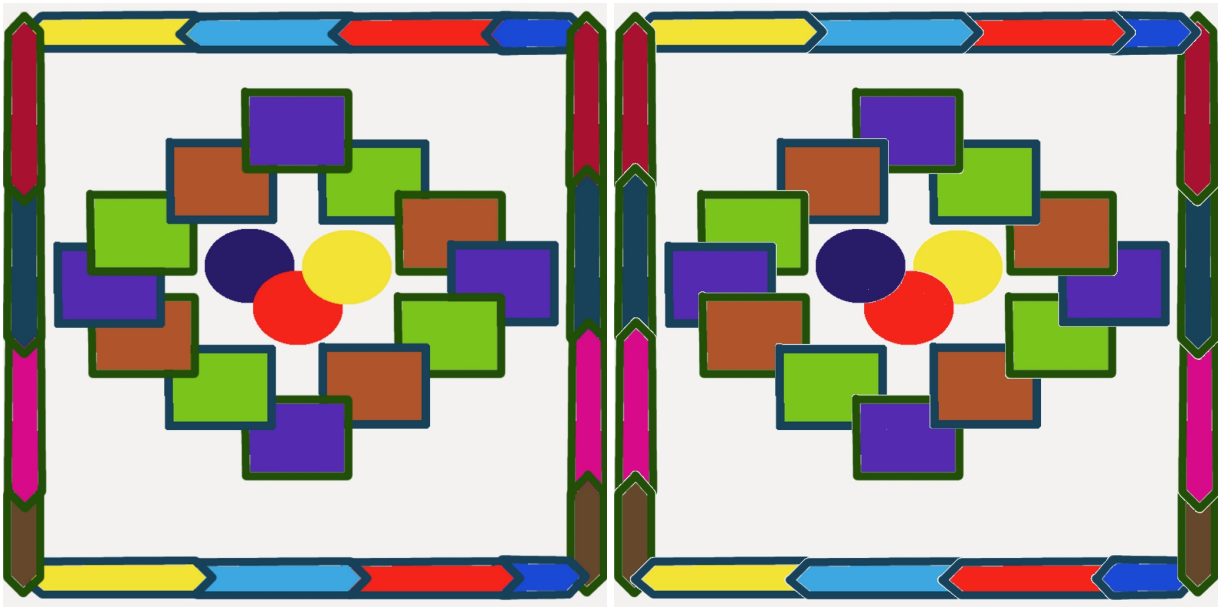


Figure 9: Left: An image with 32 layers. Right: The layers have been locally reordered to form a impossible stacking of layers.