Local Layering

James McCann*
Carnegie Mellon University

Nancy Pollard†
Carnegie Mellon University

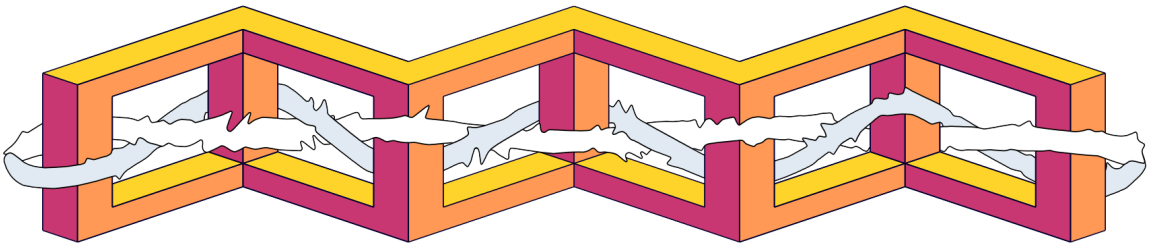

Figure 1: *Local layering allows complicated overlapping between graphical objects without manually splitting layers or painting masks.*

Abstract

In a conventional 2d painting or compositing program, graphical objects are stacked in a user-specified global order, as if each were printed on an image-sized sheet of transparent film. In this paper we show how to relax this restriction so that users can make stacking decisions on a per-overlap basis, as if the layers were pictures cut from a magazine. This allows for complex and visually exciting overlapping patterns, without painstaking layer-splitting, depth-value painting, region coloring, or mask-drawing. Instead, users are presented with a layers dialog which acts locally. Behind the scenes, we divide the image into overlap regions and track the ordering of layers in each region. We formalize this structure as a graph of stacking lists, define the set of orderings where layers do not interpenetrate as consistent, and prove that our local stacking operators are both correct and sufficient to reach any consistent stacking. We also provide a method for updating the local stacking when objects change shape or position due to user editing – this scheme prevents layer updates from producing undesired intersections. Our method extends trivially to both animation compositing and local visibility adjustment in depth-peeled 3d scenes; the latter of which allows for the creation of impossible figures which can be viewed and manipulated in real-time.

CR Categories: I.3.3 [Computing Methodologies]: Computer Graphics—Picture/Image Generation

Keywords: layers, visibility, compositing, image editing, stacking, animation

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

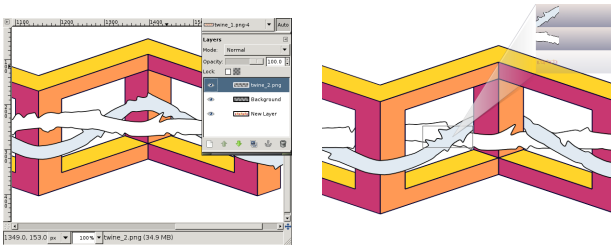

Figure 2: **Left:** *A standard paint program (GIMP pictured) provides only a global ordering of layers.* **Right:** *Our local layering prototype allows a different ordering in each region of overlap. Here the local layering dialog is being used to control the overlap of the white and gray ropes at one of their crossing points.*

1 Introduction

Standard digital image compositing, as used in modern paint and video editing packages, blends a stack of layers together to create a final image. The order and type of blending performed is uniform over the entire picture; thus, if part of layer A occludes part of layer B, then any other part of layer A must, perforce, occlude layer B. Image editors normally have some form of *layers dialog* to allow the ordering of objects to be changed.

In this paper we show how to allow object ordering edits to be made at a local, rather than global, level. Stacking control is performed through a local layers dialog (Figure 2) which controls stacking at a user-selected point in the image. With this local ordering control, users are able to create layers that weave over, under, and through one-another (Figures 1, 3), mimicking the behavior of objects in the real world (Figure 4).

The computational core of our approach is a method of representing local layering: the list-graph. Building on this structure, we define a consistent stacking, and prove that our local order-adjustment operators are both correct and sufficient to navigate the space of consistent stackings. We are also able to preserve and update this local ordering information in the presence of objects that change shape or position due to user edits. Though our prototype implementation is raster-based, the algorithms presented will work in any scenario where there are planar graphical primitives whose regions of intersection can be computed (e.g. any vector image built from the basic primitives outlined by the scalable vector graphics specification [2003]).

Figure 4: Left, *objects stacked in the real world.* **Middle,** *objects stacked with local layering.* **Right,** *layers used.*

Figure 3: *Complex overlapping creates visual interest; this composition uses only four layers – the snake, the rose, the staff, and the background.*

2 Background

In graphics, the notion of representing scenes as stacks of partially-transparent layers was largely borrowed from the film industry, and dates to the late 1970s. Along with premultiplied alpha and the compositing algebra of Porter and Duff [1984], the main contribution of graphics to compositing was the notion of intrinsic alpha, in which transparency is viewed as part of each pixel. It is this notion which encourages us to think of pixels in raster images as belonging to discrete objects instead of image-sized layers with associated image-sized mattes. More information about the history of compositing is available in a technical memo by Alvy Ray Smith [1995], and makes for interesting reading. Compositing remains largely unchanged in today's image- and video- editing programs (e.g. [Kimball and Mattis 1995–2009; Apple 1999–2009]).

The inverse problem, that of creating a stack of layers corresponding to an image, is of interest to vision researchers as a primitive for image understanding [Nitzberg and Mumford 1990]. In real scenes, layers may be segmented using any number of features (e.g. motion [Adelson and Wang 1994]). If all object contours are known, such a stack of layers – corresponding to front- and back- facing regions of the object – may be constructed using the paneling algorithm of Williams [1997]. This approach has been systematized for modeling [Karpenko and Hughes 2006; Cordier and Seo 2007].

When combining images of separate objects in a 3d scene, simple global layering is not enough, due to occlusion cycles. If available, depth information can be used to provide a local stacking [Duff 1985]. Alternatively, layers can be modified by removing occluded pieces [Snyder and Lengyel 1998]. The core of the approach presented by Snyder and Lengyel, which is to compile an occlusion graph into a series of compositing operations, may be useful to accelerate display of our list-graphs. However, because Snyder and Lengyel rely on whole-image operations, their technique cannot handle the situation where two layers A and B must be composited A over B in one place and B over A in another. Similar stacking results may be attained manually – for example by designing 3d geometry for a 2d scene (e.g. [Debevec et al. 1996]) or by masking out parts of objects that are occluded – though to do so is tedious.

Our construction works on the adjacency graph of regions of overlap of graphical objects. This is similar, in spirit, to a planar map [Baudelaire and Gangnet 1989] – a method of representing a vector drawing in which all curves are treated as lying in the same plane and separate fills and strokes may be assigned to any region or portion of an edge. Just as users can simulate local layering in a stacked representation by duplicating layers and painting masks, users can simulate local layering (and stacking) in a planar map by adjusting fills and colors of regions, potentially touching many adjacent regions to make one visibility edit. In contrast, our system maintains structural information about objects and stacking in each overlap region, and thus can automatically re-stack adjacent regions when visibility editing and compute proper pixel colors in overlaps of transparent objects.

Where our method relaxes the rigid structure of a layer-based approach, Asente, et al. [2007] propose a method that infers structure from a planar map in order to allow more intuitive editing operations. Their system estimates object colors for closed curves and extracts a potential stacking order by examining regions of overlap. This allows the system to propagate region fill colors in a reasonable way as curves are moved. As these stacking orders are estimates based on region coloring, they cannot reliably be made in drawings with, e.g. artist-simulated transparency. This is acceptable as it is not the goal of Asente, et al. to calculate the stacking of objects; rather, the inferred stacking is one of many cues used to determine reasonable behavior during edits. In contrast, our system explicitly maintains a consistent stacking order and allows it to be edited directly.

Wiley [2006] 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. This system is able to elegantly handle self-overlapping surfaces, and (in contrast to planar maps) does produce a full visibility-order for each region of overlap. While Druid presently only supports solid fill colors, one could imagine extending it to produce results similar to our own by constructing closed curves around layer boundaries of raster objects.

Though our proposed algorithm does not handle self-overlaps, it does have several technical advantages over that of Wiley: Our core re-stacking operator is polynomial time, whereas the core relabeling method of Druid involves a worst-case exponential-time search (though heuristics and construction of equivalence classes

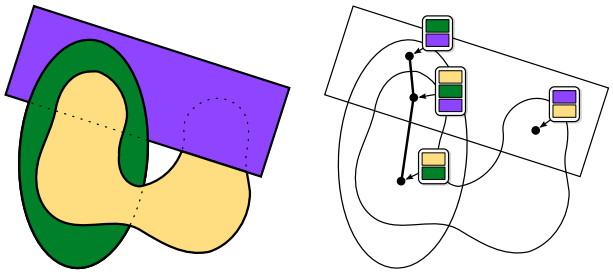

Figure 5: Left, *an example figure containing three objects. Dotted lines indicate occluded contours.* **Right,** *the corresponding regions of overlap and list-graph. Lists containing one or fewer elements are not shown.*

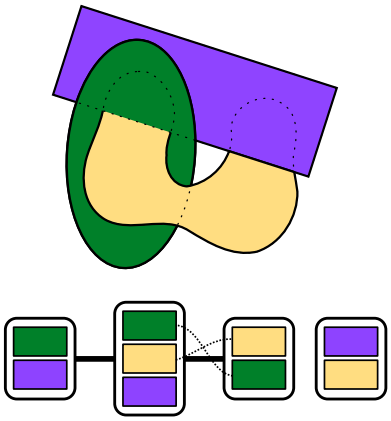

Figure 6: Above, *a version of the stacking given in Figure 5 containing an undesirable intersection between the green circle and tan blob. The list-graph,* **below***, is inconsistent (inversion indicated with dotted lines). Lists containing one or fewer elements are not shown.*

make the common case scale reasonably well). Rendering from our representation is fast, because the stacking order of surfaces inside regions is known; Druid must infer relative surface depths when rendering, which is time-consuming. Our editing operations are sufficient to reach all valid orderings; such a result would be difficult to prove given Wiley's editing operations (though it is likely true). Finally, it is not clear how to generalize the edge-crossing representation in Druid to animation, whereas our adjacent-region method was easy to generalize by giving regions temporal extent.

From a practical standpoint, our system is simple to describe, prove properties of, and implement. Also, because we extend the notion of a conventional layers dialog, our stacking adjustment method may be more familiar to users.

Igarashi et al. give a heuristic method, based on continuously monitored and updated depth values, to generate a temporally-coherent stacking order for a deformable 2d object [2005]. Their method is designed for self-occlusions of a single object and does not make guarantees about an intersection-free result, while our method is designed for complex layering of multiple objects and is provably correct.

Limited local layer re-ordering support is provided by the Push-Back tool in Real-Draw PRO [Mediachance 2001–2009]. This tool moves the top layer of a stack to some other point in the stack in a user-selected region. This automates the traditional mask-based method of achieving local re-ordering, but retains the drawback of requiring the user to manually define the region of effect. In contrast, our method can locally re-order all layers and automatically extends layer edits as far as is needed to prevent layer interpenetration.

3 Method

With local layering, layers can be ordered just as one would order paper cut-outs, weaving and overlapping but never passing through one-another. In this section, we examine the techniques we use to achieve this goal. Specifically, we discuss how our local stacking is represented; the operators invoked when the user instructs the program to change the order of layers in a region of overlap; and the ability of these operators to navigate all possible stackings. Additionally, we talk about how we handle the case where layers change content (e.g. if the user transforms or paints on a layer).

3.1 Data structure

Where a global layering approach needs only one list to track the relative ordering of objects, our approach requires lists for each region of overlap. To store these local orderings we use a structure we call a list-graph. A drawing and the corresponding list-graph appear in Figure 5.

Definition 3.1 (List-graph)**.** A list-graph G is an undirected graph with a list stored at each vertex. We write $L \in G$ to indicate that list L appears at some vertex of G, and $x < y \in L$ to indicate object x appears under – though not necessarily directly under – object y in the stacking given by list L. It is not the case that all objects must appear in all lists, or that they must appear in the same order in those lists which they inhabit.

To construct a list-graph from a set of overlapping objects, one first partitions the image into regions of overlap – that is, connected regions of the plane covered by the same set of objects. For each region of overlap, a list-graph vertex is created that contains a stacking order for the layers that appear in that region. Edges are created between any two vertices whose associated regions of overlap share an edge. The stacking order stored at each vertex may either be initialized to a global order or set based on an existing order (e.g. when moving a layer), which we will demonstrate later.

Since our operators are defined in terms of areas-of-overlap and the list-graph abstraction, they apply equally well in a raster or vector setting (as long as the primitive objects contain no self-overlap). However, our prototype uses raster graphics exclusively, and we consider a layer to exist anywhere it has non-zero alpha. More implementation details are given in Section 5.

It is important that layers do not pass through each-other in our final image. We formalize this notion as consistency:

Definition 3.2 (Consistency)**.** A list-graph G is *consistent* if for all adjacent lists $A, B \in G$ and all pairs of layers $x, y \in A \cap B$, $x < y \in A \iff x < y \in B$.

An example of the sort of artifact produced by an inconsistent list-graph is given in Figure 6.

3.2 Local Order Adjustment

Of course, being able to represent different orderings is of no use if users have no way of specifying them. In order to make local ordering adjustments, we introduce the `Flip-Up` and `Flip-Down` operators (pseudo-code for `Flip-Up` appears in Figure 7). These

```
0: Flip-Up(L, x, t):
1:     if (x ∉ L ∨ t ∉ L) return
2:     while (x < t ∈ L):
3:         Let y be the element directly above x in L.
4:         Swap x and y in L.
5:         for (L′ adjacent to L):
6:             Flip-Up(L′, x, y)
```

Figure 7: *Pseudo-code implementing the* `Flip-Up` *operator, which places layer x above layer t in list L while maintaining consistency.* `Flip-Down` *is defined similarly, replacing the condition in line 2 with $x > t \in L$ and the word "above" with "below".*

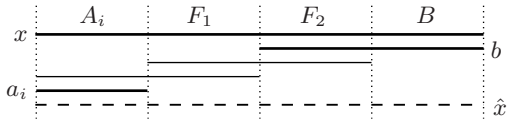

Figure 8: *Side view of scenario in Lemma 3.5; if it was the case that $x < a_i \in A_i$ before a call to* `Flip-Up`*, then it must have been the case that $x < b \in B$ as well. (e.g. x was at $\hat{x}$.)*

operators are local versions of the "move above" and "move below" operators in a global layering setting. When `Flip-Up`(L, x, t) is called, the procedure first makes sure that $x, t \in L$, then proceeds to slowly inch x up in the local stacking – calling `Flip-Up` on all the adjacent lists after each increase to make the list-graph consistent. (If it were, instead, to jump x immediately above t then the information about x's new order with respect any element $x < b < t$ would not be propagated.) `Flip-Down` proceeds similarly, lowering x until it is below t.

However, while the rearrangement operators in a global setting are trivial, these local operators are not, and thus need to be proven correct.

Theorem 3.3 (Termination of `Flip-Up`)**.** *On list-graph G,* `Flip-Up`*(L, x, t) terminates in $O(\#\text{edges} \cdot \#\text{layers})$ time.*

Proof: If `Flip-Up` is called on a list L with no neighbors, then $O(\#\text{layers})$ work is needed. Otherwise, with the proper data structures, $O(1)$ work is done per execution of the recursive call on line 6. We charge this work to the edge traversed by the recursive call. Consider an edge between lists A and B. Since x must be moved up one step in either A or B (line 4) to traverse the edge between them, and x is never lowered, at most $|A| + |B| = O(\#\text{layers})$ calls are made. Multiplying, at most $O(\#\text{edges} \cdot \#\text{layers})$ work was done. ■

Theorem 3.4 (Soundness of `Flip-Up`)**.** *After* `Flip-Up`*(L, x, t) runs on a consistent list-graph G, the list-graph remains consistent and x appears above t in list L.*

Proof: Notice that `Flip-Up` will not terminate without $x > t \in L$, so all we need to show is that G remains consistent. Proceed by contradiction. Assume that there is a pair of elements, shared by adjacent lists A, B, which appear in different orders in each list. Since `Flip-Up` never changes the relative order of non-x elements, one of these must be x. Call the other y. Without loss of generality, let $x < y \in A$ and $x > y \in B$. Since `Flip-Up` never moves x down, it must be the case that $x > y \in B$ because of the action of `Flip-Up`. But this leads to a contradiction, because when `Flip-Up` placed x above y in B it would have recursed to A and placed x above y there as well. ■

The termination and soundness of `Flip-Down` proceed similarly.

Now we need to address a more subtle point. While we have shown that our operators always take consistent list-graphs to consistent list-graphs, it could be the case that there are some consistent configurations that cannot be reached with `Flip-Up` and `Flip-Down`; this would – no doubt – be infuriating to users. First, though, we need a lemma:

Lemma 3.5 (Invertability)**.** *The action of* `Flip-Up` *may be inverted by a sufficient number of calls to* `Flip-Down`*.*

Proof: We show that the results of a call to `Flip-Up`(L, x, t) may be undone by the following procedure: At step i, find a remaining inversion – that is, choose layer a_i in list A_i such that it was the case that $x < a_i \in A_i$ before the call to `Flip-Up`, and it is currently the case that $x > a_i \in A_i$. Call `Flip-Down`(A_i, x, a_i).

To show this approach is correct, we demonstrate that each call to `Flip-Down` strictly decreases the set of remaining inversions. Specifically, consider $b \in B$ such that $x > b \in B$ before the ith call to `Flip-Down` and $x < b \in B$ after the call. (One possible scenario is illustrated in Figure 8.)

Before the call to `Flip-Down`(A_i, x, a_i), there must have been adjacent lists $A_i, F_1, \ldots, B$ such that

$$a_i \leq f_1 < x \in A_i, f_1 \leq f_2 < x \in F_1, \ldots, f_k \leq b < x \in B$$

(This is simply writing down the condition for `Flip-Down` to have recursed to $b \in B$.) These inequalities must have also been true just after `Flip-Up` returned, since the intervening $i-1$ calls to `Flip-Down` can't have raised x.

Consider the same adjacent lists before the call to `Flip-Up`. By hypothesis, $x < a_i \in A_i$; so, by consistency:

$$x < a_i \leq f_1 \in A_i \Rightarrow x < f_1 \leq f_2 \in F_1 \Rightarrow \ldots \Rightarrow x < f_k \leq b \in B$$

Thus, the ith call to `Flip-Down` has returned b and x to their original order; the set of inversions strictly decreases; and we are done. ■

With that lemma in hand, we can dive into the main theorem:

Theorem 3.6 (Sufficiency)**.** *The* `Flip-Up` *and* `Flip-Down` *operators allow any consistent configuration of given list-graph G to be reached from any consistent starting position.*

Proof: First, we show how to take any consistent configuration of G to a canonical form where the layers $1, \ldots, n$ appear bottom-to-top in numerical order:

Proceed by induction on current layer i. Assume layers $i+1, \ldots, n$ appear in numerical order at the top of any $L \in G$ in which they occur. Now, call `Flip-Up`(L, i, y) for all $L \in G$ and $1 \leq y < i$. This moves i to the top of every $L \in G$ in which it appears, just before elements $i+1, \ldots, n$. `Flip-Up`(L, i, k) will never be called with $k > i$, because that would imply that $k < i \in L$, which contradicts the assumption that all $k > i$ are at the top of L. Therefore we can place the layers in a canonical order with calls to `Flip-Up`.

Of course, this means that – by Lemma 3.5 – we can take the canonical form to any consistent form G' using calls to `Flip-Down`. ■

In practice, using `Flip-Up` and `Flip-Down` to navigate through the space of consistent stackings is quite intuitive, despite the cumbersome construction used in the proof above.

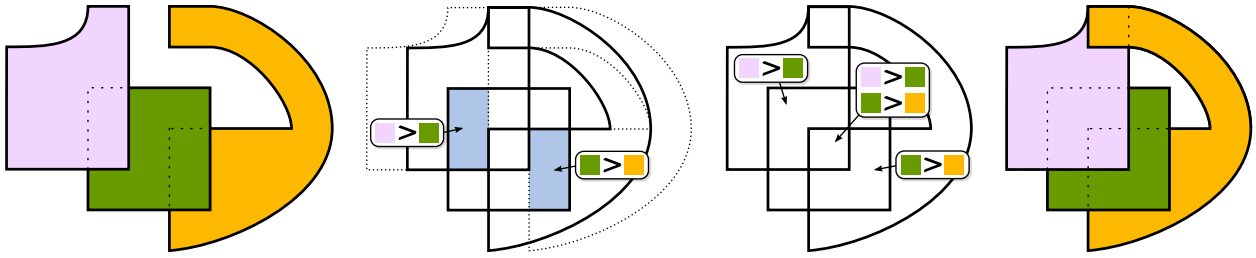

Figure 9: *Maintaining a temporally coherent stacking.* **(a)** *The original stacking.* **(b)** *The updated layer positions, with local votes cast in the image-space overlap between the old stacking regions (dotted) and the new (solid).* **(c)** *After vote propagation.* **(d)** *Final stacking; all votes satisfied. The overlap at the top was not constrained, so was chosen based on a given tie-breaking order (here prioritizing the orange crescent).*

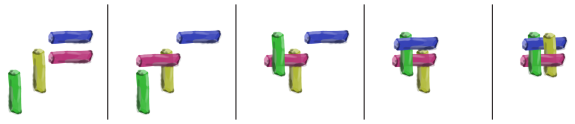

Figure 10: *The user weaves together four layers by dragging them. This example relies on temporal coherence to maintain stacking information. Our prototype places the active layer on top in new overlaps.*

3.3 Temporal Coherence

There are many situations when one wishes to initialize the local orders in a new list-graph to match those in an existing one – for instance, when moving (Figure 10) or editing layers, or when moving the viewpoint around a visibility-edited 3d model. This is non-trivial, as there may not be a one-to-one correspondence between the old regions of overlap and the new ones, and their adjacency may have changed. To overcome this obstacle, we propose an area-weighted voting scheme (illustrated in Figure 9). First, regions in the old list-graph send votes for their current stacking order to regions in the new list-graph. These votes are weighted by the area of image-space overlap between the regions. (One can imagine using some sort of pixel-pixel correspondence or motion estimation to warp these regions; however, since we want to handle arbitrary changes, we do not.) Next, these local votes are spread between regions based on consistency. Finally, the new list-graph is ordered by greedily choosing a consistent set of votes.

At each list L with associated area $A(L)$ in the image, we define the reward for a given order $a < b \in L$ as the sum of areas of overlap with old lists L_{old} which have $a < b \in L_{\text{old}}$:

$$R_{\text{local}}(a < b \in L) = \sum_{L_{\text{old}} \text{ s.t. } a<b\in L_{\text{old}}} \text{Area}(A(L) \cap A(L_{\text{old}})) \quad (1)$$

Since choosing a given order $a < b \in L$ will force adjacent lists containing a, b to also adopt that order, we sum the local rewards over each set of adjacent lists:

$$R_{\text{global}}(a < b \in L) = \sum_{L_{\text{adj}} \in C} R_{\text{local}}(a < b \in L_{\text{adj}}) \quad (2)$$

(Where C is all lists L_{adj} for which, by consistency, $a < b \in L \Rightarrow a < b \in L_{\text{adj}}$; i.e. those L_{adj} such that a path exists from L to L_{adj} with every list in that path containing a, b.)

Finally, we choose facts (i.e. statements of the form $a < b \in L$) about the ordering in a greedy manner, choosing, at each round, the largest fact that is allowed, given those already chosen:

$$fact_i \equiv \text{argmax}_{\text{allowed } a<b\in L} R_{\text{global}}(a < b \in L) \quad (3)$$

(We call an ordering fact "allowed" if it does not contradict a fact that can be derived from those we have already chosen.)

If any fact with nonzero reward is not allowed, this indicates that we are unable to perfectly satisfy temporal coherence and some layer will "pop" through another (as in Figure 11). Since this may not be desired, we provide the option to halt at this point and roll back the change; this has the effect of stopping layers from being moved through each-other and impossible figures from being rotated improperly. Otherwise, we proceed as follows.

Once all facts with nonzero rewards are either chosen or not allowed, we order each list L based on those facts that can be inferred from the chosen facts. In cases when neither $a < b \in L$ nor $b < a \in L$ can be derived – such as when user actions introduce a new overlap – we choose the relative order of a and b to be consistent with a global tie-breaking ordering (if we were to chose locally, we might introduce inconsistencies). This ordering depends on the application. When moving layers, we place the layer being moved at the top of the ordering, so it slides 'on top' of anything it is dragged into. When working with depth-peeled 3d models we use the depth ordering, so any new depth complexity winds up stacked properly.

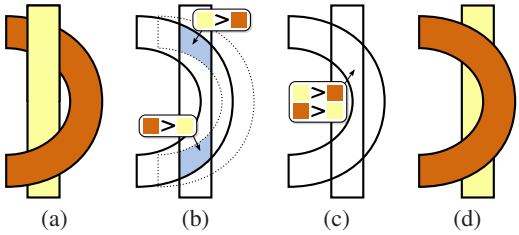

Figure 11: *Starting with configuration* **(a)** *the user moves the crescent to the left. The votes,* **(b)***, are inconsistent in the central region,* **(c)***. Picking a consistent subset will result in popping (one possibility shown in* **(d)***). Depending on a user toggle, we can instead roll back the edit.*

4 Extensions

We can extend the notion of local layering beyond simple stacks of static images. In this section, we demonstrate how to adjust layering in animations (viewing each layer as a spatio-temporal volume), and how to use local layering to adjust depth-order in 3d models.

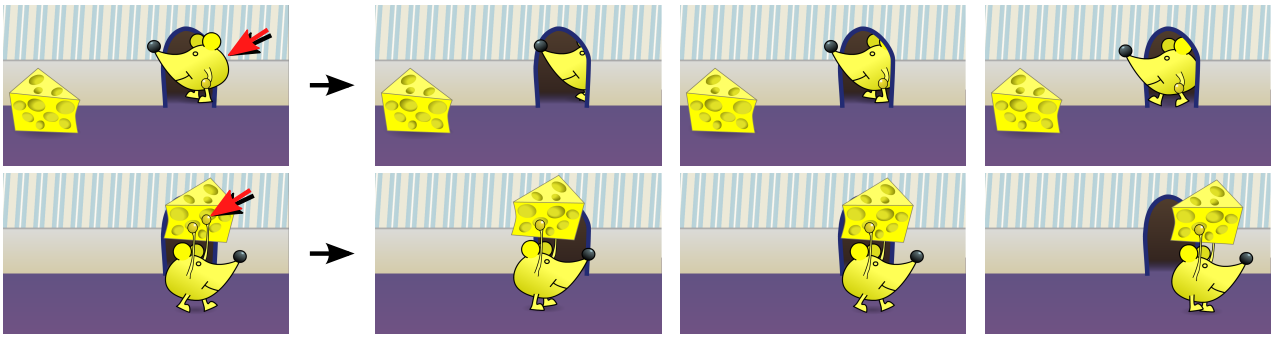

Figure 13: *Animation stacking with spatio-temporal overlap regions.* **Top:** *Re-ordering of the mouse and wall at the red arrow in frame 22,* **left**, *results in a stacking change in the spatio-temporally local overlap region (***right**, *frames 17, 22, and 27).* **Bottom:** *Later in the same animation, placing one of the mouse's hands behind the cheese in frame 94,* **left**, *changes an overlap region extending to,* **right**, *frames 89, 94, and 99. The mouse passes in front of the wall in these frames because the edit at frame 22 is temporally local.*

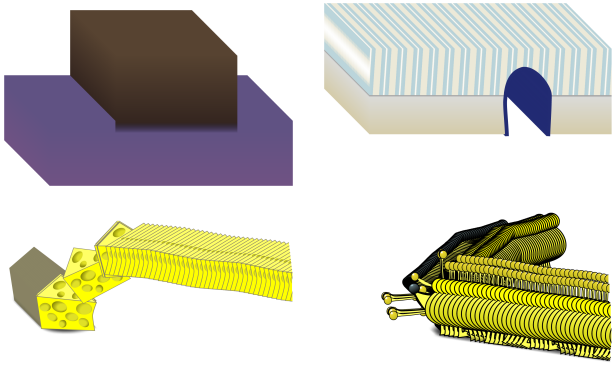

Figure 12: *Spatio-temporal volumes corresponding to the four layers used in the animation editing example in Figure 13.*

4.1 Animation

We can adjust the spatio-temporally local stacking of animated layers by extending the notion of adjacency across time. That is, we view the entire animation as a stack of overlapping volumes (e.g. Figure 12) and build a list-graph over space-time regions of similar overlap within that volume. Working with space-time volumes means that consistency guarantees layers do not 'pop' through each-other over time. Temporally local stacking is useful in animations when one element starts behind another then passes in front of it; for instance, a character walking through a door (Figure 13).

4.2 Impossible Figures

Our method of local stacking can also be used to create impossible figures, or correct unwanted interpenetration of 3d objects. In this case, a 3d model is decomposed into layers using GPU-based depth peeling [Everitt 2001], and the user can change the stacking order of these layers using our system. This process is illustrated in Figure 14; another example is given in Figure 15. Frame-to-frame stacking coherency is maintained using the method of Section 3.3.

5 Implementation Details

Our prototype represents each layer as an image-sized set of pixels with alpha values. The layer is considered to exist wherever it has non-zero alpha. Regions of overlap are calculated by first splatting all the layers into a bitfield image (e.g. a pixel overlapped by layers 2, 5, and 7 would have bits 2, 5, and 7 set), then by extracting the connected components of this bitfield. We consider pixels to be adjacent in the up, down, left, and right directions (that is, diagonal adjacency is not considered). When working with animations, pixels immediately before and after a given pixel are considered adjacent as well. A tag image is created that stores the index – in the list-graph – of the stacking order for each pixel. Final compositing proceeds by looking up the stacking order at each pixel (using the tag image) and combining the layers in this order. When working with 3d models, depth-peeling is performed on the GPU and the layers read back into main memory; all other operations are performed on the CPU. While we expect that pushing compositing onto the GPU would accelerate the process significantly, our present CPU version runs fast enough to allow interactive editing of 1-5 megapixel images.

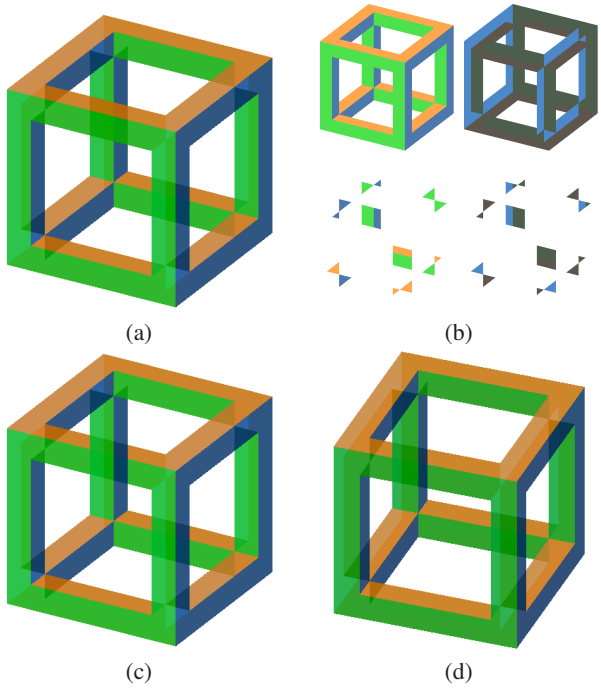

Figure 14: *Constructing an impossible cube.* **(a)** *Before manipulation.* **(b)** *Depth-peeled layers.* **(c)** *After local layer rearrangement.* **(d)** *Rotated (stacking preserved using our temporal coherence method).*

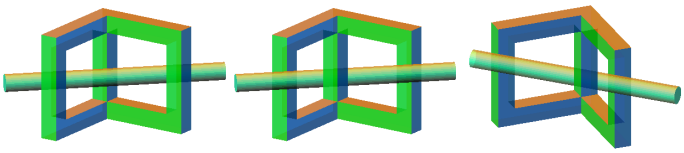

Figure 15: *Another impossible figure.* **Left,** *before editing.* **Middle,** *after editing.* **Right,** *another view.*

6 Future Work

At present, we cannot handle layers that overlap themselves. To do so we would need to change the list-graph structure to allow a layer to appear multiple times in a given area of overlap. We would also need to add connectivity information to tell us which instances of layers connect over each edge. However, such list-graphs may not, in general, have any consistent stacking, which makes designing algorithms and the associated proofs challenging future work.

While our method of creating impossible figures does allow for a neat demo, generating layers with depth peeling is not particularly intuitive. Consider the difficulty a user would encounter in trying to insert a 2d illustration of a curl of smoke rising through the center of the cube in Figure 14. Some connected polygons are spread across multiple layers, while other disconnected polygons end up adjacent in the same peel of depth complexity. Splitting the depth-peeled layers at occluding contours and adding connectivity information, as proposed above for self-overlaps, could help address these difficulties.

At present, our notion of consistency is founded on layers either being present ($\alpha > 0$) in a overlap or absent ($\alpha = 0$); this works well for layers that represent solid objects, but is not terribly satisfying for objects that are partially transparent everywhere (e.g. fog). To address this, one could envision relaxing our notion of consistency to allow layers to pass through each-other smoothly and at a rate proportional to their transparency.

While the strength of our approach lies in its locality, this locality can sometimes also be troublesome for users. Consider, for instance, a layer consisting of a cloud of fine particles. To move it in front of another layer requires the user to select each particle individually – a tedious task. To abjure this tedium we could allow multiple regions of overlap to be selected at once (e.g. by painting a stroke or dragging a box). Local layers dialog operations would then result in invocation of `Flip-Up` or `Flip-Down` at all selected regions. Some care, however, is required if the user selects regions with inconsistent stacking orders – both to convey this inconsistency to the user, and to figure out the "right thing" to do with stacking manipulations.

7 Conclusion

We have proposed a local ordering method for graphical objects that can replicate the complex stacking possible with real art materials (e.g. cut paper), instead of being fixed to the global-ordering paradigm inherited from film compositing. In our prototype, users are given a layers dialog which operates locally. Our method stores a local ordering for each region of overlap in a list-graph structure. We defined, as *consistent*, the subset of these graphs with the desirable property that layers do not pass through each-other, and demonstrated two operators `Flip-Up` and `Flip-Down`, which we proved sufficient to navigate this subset. Finally, we showed how to extend this notion beyond static images by demonstrating its applicability to animation editing and the creation of impossible figures. We hope that makers of image editing software find the case for local stacking compelling enough to consider including it in their applications.

References

ADELSON, E. H., AND WANG, J. Y. A. 1994. Representing moving images with layers. *IEEE Transactions on Image Processing 3*, 625–638.

APPLE, 1999–2009. Final Cut Pro. http://www.apple.com/finalcutstudio/finalcutpro/.

ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM Transactions on Graphics 26*, 3, 30.

BAUDELAIRE, P., AND GANGNET, M. 1989. Planar maps: an interaction paradigm for graphic design. *SIGCHI Bull. 20*, SI, 313–318.

CORDIER, F., AND SEO, H. 2007. Free-form sketching of self-occluding objects. *IEEE Comput. Graph. Appl. 27*, 1, 50–59.

DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. 1996. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *Proceedings of SIGGRAPH 96*, ACM, New York, NY, USA, 11–20.

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

EVERITT, C. 2001. Introduction to interactive order-independent transparency. Tech. rep., NVIDIA.

IGARASHI, T., MOSCOVICH, T., AND HUGHES, J. F. 2005. As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics 24*, 3, 1134–1141.

KARPENKO, O. A., AND HUGHES, J. F. 2006. Smoothsketch: 3d free-form shapes from complex sketches. *ACM Transactions on Graphics 25*, 3 (July), 589–598.

KIMBALL, S., AND MATTIS, P., 1995–2009. The GNU Image Manipulation Program. http://www.gimp.org.

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

NITZBERG, M., AND MUMFORD, D. 1990. The 2.1-d sketch. *Computer Vision, 1990. Proceedings, Third International Conference on* (Dec), 138–144.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *Computer Graphics (Proceedings of SIGGRAPH 84) 18*, 3, 253–259.

SMITH, A. R. 1995. Alpha and the history of digital compositing. In *Microsoft Technical Memo #7*.

SNYDER, J., AND LENGYEL, J. 1998. Visibility sorting and compositing without splitting for image layer decompositions. In *Proceedings of SIGGRAPH 98*, ACM, New York, NY, USA, 219–230.

SVG WORKING GROUP, 2003. Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG11/.

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

WILLIAMS, L. R. 1997. Topological reconstruction of a smooth manifold-solid from its occludingcontour. *Int. J. Comput. Vision 23*, 1, 93–108.