Mixed-Order Compositing for 3D Paintings

Ilya Baran[1] Johannes Schmid[1,2] Thomas Siegrist[1,2] Markus Gross[1,2] Robert W. Sumner[1]
[1]Disney Research Zurich [2]ETH Zurich

Abstract

We present a method for rendering 3D paintings by compositing brush strokes embedded in space. The challenge in compositing 3D brush strokes is reconciling conflicts between their z-order in 3D and the order in which the strokes were painted, while maintaining temporal and spatial coherence. Our algorithm smoothly transitions between compositing closer strokes over those farther away and compositing strokes painted later over those painted earlier. It is efficient, running in $O(n \log n)$ time, and simple to implement. We demonstrate its effectiveness on a variety of 3D paintings.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: compositing, nonphotorealistic rendering

Links: DL PDF WEB VIDEO

1 Introduction

Traditional 2D painting techniques and digital 2D painting systems provide an artist with a lot of expressive freedom in the creation of a painting, allowing a wide range of styles. Translating this freedom into 3D so that paintings can be viewed from multiple angles has been a research challenge for many years. Meier [1996] first proposed generating brush strokes attached to 3D objects. Other systems rely more on the artist. For example, Disney's Deep Canvas [Katanics and Lappas 2003] and, most recently, OverCoat [Schmid et al. 2011], present the artist with a 3D canvas that allows him or her to place stylized paint strokes in space based on 3D proxy geometry. In this paper, we refer to a collection of brush strokes embedded in space as a *3D painting* (Figure 1).

A 3D painting system renders the scene by projecting each brush stroke onto the current view plane and rasterizing it as one or more fragments for every pixel that the stroke overlaps. This rendering method exposes a technical dilemma about the order in which the fragments should be composited. In the 2D painting metaphor, when the artist places a new paint stroke, it obscures all previous paint strokes that it overlaps. Such behavior is achieved by compositing in *stroke order*. From a 3D point of view, however, strokes that are closer to the viewer should obscure those that are farther away, which amounts to compositing in *depth order*. Compositing purely in stroke order negates much of the benefit of 3D painting, as the sense of tangible objects is lost when the view is changed. Compositing purely in depth order, on the other hand, leads to Z-fighting, precluding the artist from painting over existing strokes,

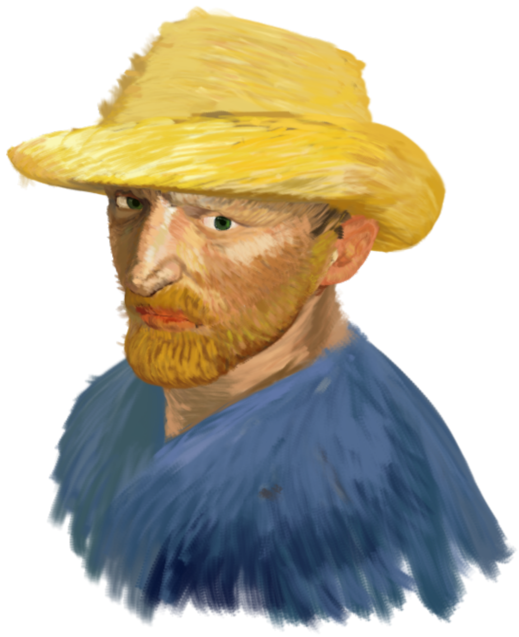

Figure 1: *A 3D painting, consisting of brush strokes embedded in space, composited using our method.* *© Disney Enterprises, Inc.*

and thus ignores an important part of the 2D painting metaphor. This problem dates back to Meier's work [1996] and is illustrated in Figure 2.

Katanics and Lappas [2003] articulated the desire for mixed-order compositing: fragments that, in the artist's mind, belong on the same surface should be composited in stroke order, while those that belong on different surfaces should be composited in depth order. Unfortunately, assigning each fragment to a specific surface is often impossible: the stroke that generated the fragment may span several surfaces, may self-occlude, or may not even conform to a surface at all. The guideline Deep Canvas adopts is therefore to choose the appropriate ordering based on a depth tolerance d: fragments whose depths are within d of each other are assumed to lie on the same surface and composited in stroke order, but fragments that are farther apart are composited in depth order.

In this work, we formalize this idea for the first time. We state the requirements for compositing order and temporal coherence as four properties, which a mixed-order compositing function must satisfy (Section 3.2). We discuss the ways in which existing solutions fail to satisfy one or more of these properties (Section 3.3) and describe the resulting artifacts. We design a function that does satisfy all four properties (Section 3.4) and show how to compute it efficiently (Section 3.5). We present renderings produced by this method and discuss limitations and future extensions (Section 4).

2 Related Work

One of the first uses of paint strokes as rendering primitives was seen in Haeberli's "Paint By Numbers" [1990], which augments brush strokes indicated by the user with additional information, such as color or direction, from underlying photographs or rendered images. This concept was later extended to arrange and render brush strokes automatically based on photographs or video, and has lead to excellent results [Litwinowicz 1997; Hertzmann 1998; Lu et al. 2010]. Because such methods work entirely in image space, they do not suffer from visibility ambiguities during rendering.

Instead of images, 3D scenes can also be used as the basis to generate brush strokes. In Meier's painterly rendering system [1996], brush strokes are seeded by particles that are placed on 3D models. Color, direction, and size of the brush strokes are determined by separate reference images that are obtained by rendering the 3D models using specialized shaders. To obtain a painterly appearance, the brush strokes are rendered in image space after projection to the screen plane. This rendering concept was refined by Disney's Deep Canvas 3D painting system [Daniels et al. 2001; Katanics and Lappas 2003], in which brush strokes were manually applied in different views of a scene and immediately projected onto the underlying 3D geometry in a parametric form. Recently, the concept of stroke projection has been generalized to a more flexible method of embedding paint strokes in space that does not restrict the stroke placement to 3D surfaces [Schmid et al. 2011]. All of these methods share the need for mixed-order compositing, as described in the introduction, but they use different ad-hoc solutions, which we discuss in Section 3.3.

The "WYSIWYG NPR" system presented by Kalnins and colleagues [2002] also allows the user to project paint strokes onto 3D models, but they use discrete visibility testing instead of depth ordering, which can lead to popping and is restricted to strokes which adhere strictly to 3D surfaces. Keefe and colleagues [2001] used a virtual-reality environment to let the artist paint directly in space. In their system, paint strokes are represented by standard 3D primitives, which can be composited in strict depth order, but cannot easily accomodate a more painterly aesthetic.

Digital compositing with the alpha channel was invented by Catmull and Smith [Smith 1995]. Wallace [1981] wrote down the equation for the *over* operator, while Porter and Duff [1984] introduced premultiplied alpha and described the algebra of compositing. Because the over operator is not commutative, the order in which fragments are composited matters, leading to sorting algorithms for real-time rendering [Mammen 1989] and techniques for allowing the user explicit control over the order [McCann and Pollard 2009].

Conflicting orders of compositing have recently been considered by Bruckner et al. [2010] for illustrating 3D layers. Their goal is, in some sense, a transpose of ours: we aim to composite fragments close together in stroke order and fragments far apart in depth order, while they composite user-specified adjacent layers in depth order and the results in layer order. Their method works well for illustrative rendering and shares some technical similarity with ours. The problem they are solving is more restricted than ours: in their setting, the user specifies which layers are composited in which order and this order cannot change continuously, while our method needs to smoothly transition between compositing in depth order and in stroke order. Furthermore, their solution requires worst-case quadratic time in the number of layers (which roughly correspond to our fragments), while ours runs in $O(n \log n)$. For 3D paintings with hundreds or even thousands of fragments per pixel, a quadratic-time method is prohibitively expensive.

3 Method

We start by defining what properties a mixed-order compositing function needs to satisfy for pleasing animation results. We then describe such a function and an algorithm that computes it efficiently. In the supplemental material, we prove that our compositing function satisfies the desired properties.

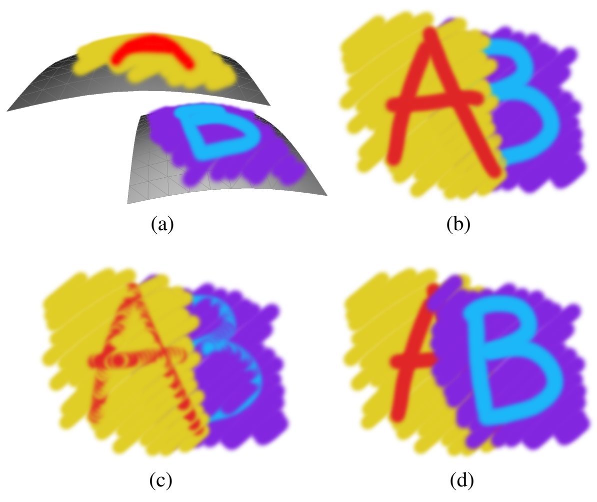

Figure 2: *Compositing order conflict: (a) shows the spatial arrangement of the paint strokes; (b) is the desired compositing result; (c) was composited in strict depth order; and, (d) was composited in strict stroke order. Each stroke is rendered as a collection of individual splats. The surfaces in (a) are only included for visualization—strokes are embedded in space.*

3.1 Compositing Background

In the presence of a well-defined ordering, fragments or layers are typically composited using the over operator. The formula for the over operator depends on how color and transparency are represented. Given two fragments whose colors and opacities are (h_1, α_1) and (h_2, α_2), the over operator (denoted by $\oplus$) is:

$$(h_1, \alpha_1) \oplus (h_2, \alpha_2) = \\ = \left(\frac{h_1\alpha_1 + h_2(1-\alpha_1)\alpha_2}{\alpha_1 + (1-\alpha_1)\alpha_2}, \alpha_1 + (1-\alpha_1)\alpha_2 \right).$$

It is common to use the premultiplied-alpha representation, storing $c = h\alpha$ instead of h. This simplifies the over operator to

$$(c_1, \alpha_1) \oplus (c_2, \alpha_2) = (c_1 + (1-\alpha_1)c_2, \alpha_1 + (1-\alpha_1)\alpha_2).$$

In this exposition, we will work with the premultiplied-alpha representation, treating c as a 3-vector. In our implementation, we actually store $\beta = 1 - \alpha$ instead of α, which further simplifies the over operator to:

$$(c_1, \beta_1) \oplus (c_2, \beta_2) = (c_1 + \beta_1 c_2, \beta_1\beta_2).$$

From the above expression, it is easy to see that $\oplus$ is associative, but not commutative.

3.2 Desired Properties

Just like regular compositing, mixed-order compositing should work independently on each pixel. In our exposition, we will therefore consider a single pixel. Suppose that n fragments were rasterized at the pixel. Each fragment has a color c_i, opacity α_i, depth z_i, and stroke number s_i, and we write $\mathbf{f}_i$ to denote the entire fragment $(c_i, \alpha_i, z_i, s_i)$. The fragments are given in depth order, starting with the closest to the viewer, so $z_i \leq z_{i+1}$. We also assume that the stroke numbers s_i range between 1 and n and that no two fragments in a pixel have the same stroke number.

We now describe the properties a function $C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n)$ needs to satisfy to be a good compositing function for our application. These properties express our high-level goals: fragments close in depth should be composited in stroke order while fragments further apart should be composited in depth order, and spatial and temporal coherence should be maintained. In expressing them mathematically, we strove to balance generality and the ease with which we can reason about them, but we did not attempt to formulate an exhaustive set of properties for the problem.

Because we treat fragments at the same depth as being on the same surface, we would like to composite fragments at the same depth in stroke order:
Property 1. *If two fragments i and $i+1$ have the same depth and are adjacent in stroke order, i.e., $z_i = z_{i+1}$ and $s_i = s_{i+1} + 1$, replacing them with their composite in stroke order should not change the final output for the pixel:*

$$C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n) = C(\mathbf{f}_1, \ldots, \mathbf{f}_{i-1}, \mathbf{f}_i \oplus \mathbf{f}_{i+1}, \mathbf{f}_{i+2}, \ldots, \mathbf{f}_n)$$

We treat fragments separated in depth as being on different surfaces and would like to composite them in depth order. Suppose that the user specifies a distance d such that fragments farther than d apart in depth are considered to be on different surfaces. Then, compositing them in depth order should not change the result:
Property 2. *If for some i, $z_{i+1} \geq z_i + d$, then:*

$$C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n) = C(\mathbf{f}_1, \ldots, \mathbf{f}_i) \oplus C(\mathbf{f}_{i+1}, \ldots, \mathbf{f}_n)$$

A stroke whose alpha smoothly fades to zero towards its borders can nevertheless cause sharp visible edges when composited with other strokes (see e.g., Figure 6, bottom left). To avoid these edges, we require that a fully transparent fragment have no effect:
Property 3. *If for some i, $\alpha_i = 0$, then:*

$$C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n) = C(\mathbf{f}_1, \ldots, \mathbf{f}_{i-1}, \mathbf{f}_{i+1}, \ldots, \mathbf{f}_n)$$

So far, we can construct a function that satisfies all of the above properties simply by sorting the fragments in lexicographical order by depth and then by stroke number. However, during animation, α's, depths, and colors may change, and popping should be avoided to obtain a nice rendering:
Property 4. *The mixed-order compositing function C must be continuous in all of the c_i's, α_i's, and z_i's.*

3.3 Existing Techniques

Not every function C that satisfies these properties is necessarily a good compositing function. For instance, C may exhibit undesirable behavior when all $z_i - z_{i-1}$ approach $d/2$ because this configuration is sufficiently far from the premises of Properties 1 and 2. Nevertheless, we have found in our experiments that in natural candidates for C, artifacts can be explained in terms of violations of these properties.

Meier [1996] simply composites the strokes in depth order, which violates Properties 1 and 4. The rendering in OverCoat [Schmid et al. 2011] offsets the fragments' z_i by a function of stroke order before compositing them in depth order. This method does not satisfy Properties 2 and 4, as offset surfaces may poke through closer surfaces, and popping can occur when the offset fragments switch depths. It also fails to satisfy Property 1 because two strokes at the same depth with adjacent stroke numbers may "sandwich" a third stroke after the offsets are computed.

Luft and Deussen [2006] propose a blurred depth test for smooth compositing. Their goals differ from ours in that they only aim for improving temporal coherence but do not need to deal with conflicting compositing orders. Property 1 therefore does not apply. They also do not support user-specified alpha transparency, so Property 2 is trivially satisfied and Property 3 does not apply. The use of depth-dependent compositing in that method leads to a violation of Property 4, resulting in popping artifacts in their animations. For depth order, the method of Bruckner et al. [2010] satisfies Properties 2–4, but it also is not designed to take stroke order into account.

Deep Canvas [Daniels et al. 2001] clusters the fragments by z and composites each cluster separately using a combination of depth and stroke order. As we understand it, this method satisfies Property 2, but the clustering is sensitive to z and can be changed by a zero-α fragment, violating Properties 3 and 4. We experimented with other methods that use clustering (including soft clustering to maintain continuity) to determine distinct surfaces, but we could not simultaneously satisfy Properties 1, 3, and 4.

3.4 Compositing Function

The main idea of our method is to replace the color of each of the fragments with the result of compositing nearby fragments in stroke order, and then composite the fragments with replaced colors in depth order. While this idea is conceptually simple, its implementation requires careful attention to ensure continuity and good performance.

The user specifies a global constant, d, so that fragments farther than d apart only composite in depth order. We therefore define the function $S(z) = (S_c(z), S_\alpha(z))$ that is the result of compositing all fragments with depths strictly between $z - d/2$ and $z + d/2$ in stroke order. When there are no fragments between $z - d/2$ and $z + d/2$, we define S to be the identity color, $(\mathbf{0}, 0)$. S is a piecewise-constant function with discontinuities at $z_i + d/2$ and $z_i - d/2$. If we assign a new color to each fragment using $S(z_i)$, we would not have continuity with respect to z_i's. Instead, we smooth $S(z)$ in depth by convolving it with a box filter of width γd, where γ, with $0 < \gamma \leq 1$, specifies how much smoothing is perfomed. We compute the colors and alphas as:

$$(c_i', \alpha_i') = \frac{1}{\gamma d} \int_{z_i - \gamma d/2}^{z_i + \gamma d/2} S(z)\, dz.$$

Note that because the colors are premultiplied with alphas, this integral is correctly weighted by alpha. We replace the fragment colors, while keeping their original alpha values, setting $c_i'' = c_i' \alpha_i / \alpha_i'$. Furthermore, because $S_\alpha(z) \geq \alpha_i$ over the range of integration, we have $\alpha_i \leq \alpha_i'$, and the division is well-behaved for nearly-transparent fragments. The final output is $C(\mathbf{f_1}, \ldots, \mathbf{f_n}) = (c_1'', \alpha_1) \oplus \cdots \oplus (c_n'', \alpha_n)$. Although the final output is composited in depth order, C does not exhibit discontinuities when the depth order changes because two fragments at the same depth will have the same replacement color.

In the supplemental material, we prove that this function satisfies all of our desired properties. Together with our experiments (Section 4.1), the proofs confirm the intuitive behavior of mixed-order compositing.

3.5 Algorithm

We now describe an algorithm to compute the function C efficiently in $O(n \log n)$ time and using $O(n)$ memory. The high-level procedure is to explicitly compute $S(z)$ in $O(n \log n)$ time and then define the replacement colors in linear time. Naïve algorithms for

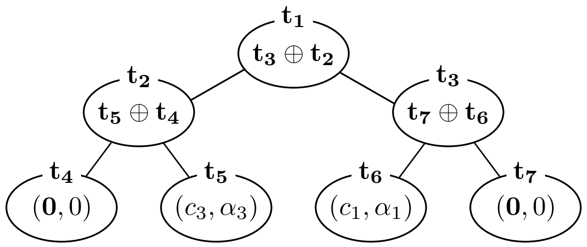

Figure 3: *The binary tree used to compute $S(z)$. The leaves correspond to four fragments in this pixel. At the time instant shown, fragments 1 and 3 (with $s_1 = 3$ and $s_3 = 2$) are in the window. Suppose fragment 2 enters the window and $s_2 = 4$. This change requires an update of the nodes $\mathbf{t_7}$, $\mathbf{t_3}$, and $\mathbf{t_1}$ for the root to have the correct new $S(z)$.*

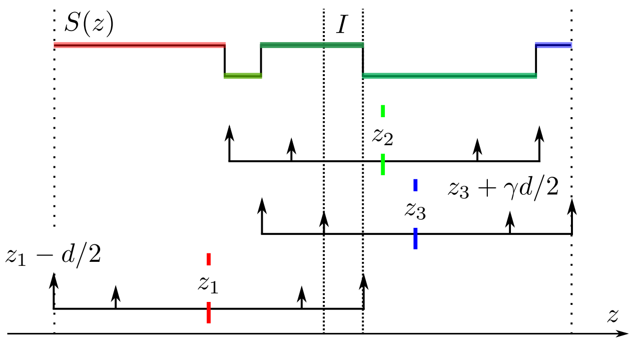

Figure 4: *This figure illustrates the algorithm for determining the replacement colors. Three fragments are shown with their compositing windows ($\pm d$) and box filter windows ($\pm \gamma d$). The stroke order is z_1, z_3, z_2. During the integration, the contribution of the interval I is added to the replacement colors of fragments z_2 and z_3. The vertical separation is used only to make the illustration less cluttered.*

both of these tasks run in quadratic time because each distinct value of S can depend on all n fragments and each replacement color can depend on all $\Omega(n)$ distinct values of S. Our algorithm instead sweeps across depth and exploits the problem structure to compute values of S and replacement colors (c', α') incrementally.

We start by using a sort to assign fragments distinct stroke numbers from 1 to n. As previously noted, $S(z)$ only changes at $z_i + d/2$ and $z_i - d/2$, and therefore it only needs to be computed at these locations. At $z_i - d/2$, S is modified to include the new fragment z_i, and at $z_i + d/2$, S changes to no longer include z_i. To accomodate these events, we need a data structure that maintains a subset of the fragments and can add or remove a fragment from the subset efficiently. It also needs to be able to report the composite of this subset in stroke order. We use a complete binary tree with n leaves, each node of which stores a color and alpha, initially $(\mathbf{0}, 0)$. Leaf s_i of the tree stores either $(\mathbf{0}, 0)$ or (c_i, α_i), and each internal node stores the composite of its children in reverse order (because later strokes go on top). The root therefore stores the composite of all of the leaves of the tree in stroke order. Inserting or deleting a fragment can be achieved by changing the appropriate leaf from $(\mathbf{0}, 0)$ to (c_i, α_i) or vice versa and updating all of the nodes on the path to the root (Figure 3). These updates therefore run in $O(\log n)$ time.

Now that $S(z)$ is known, we compute its integral over a window of size γd around each fragment (Figure 4). Consider the union of the set of discontinuities of S and the points $z_i \pm \gamma d/2$. This union partitions the interval $[z_1 - \gamma d/2, z_n + \gamma d/2]$ into at most $4n$ subintervals. Within such a subinterval I, $S(z)$ is constant by construction. The set of fragments within $\gamma d/2$ of z is also constant and contiguous, consisting of all fragments from z_j to z_{j+k}, for some j and k. The contribution of I to each fragment in this set is $S(I)/\gamma d$ times the length of I. Because k may be as large as n, adding this contribution to all fragments is too expensive. However, if we maintain the integrals as differences between adjacent fragments, $(\Delta c'_i, \Delta\alpha'_i) = (c'_i - c'_{i-1}, \alpha'_i - \alpha'_{i-1})$, we can add the contribution to $(\Delta c'_j, \Delta\alpha'_j)$ and subtract it from $(\Delta c'_{j+k+1}, \Delta\alpha'_{j+k+1})$ in $O(1)$ time. We process all $4n$ subintervals by sweeping over the discontinuities of S and the points $z_i \pm \gamma d/2$ and incrementally maintaining j and k. Before doing the final composite, we compute (c'_i, α'_i) from the deltas by computing the prefix sum.

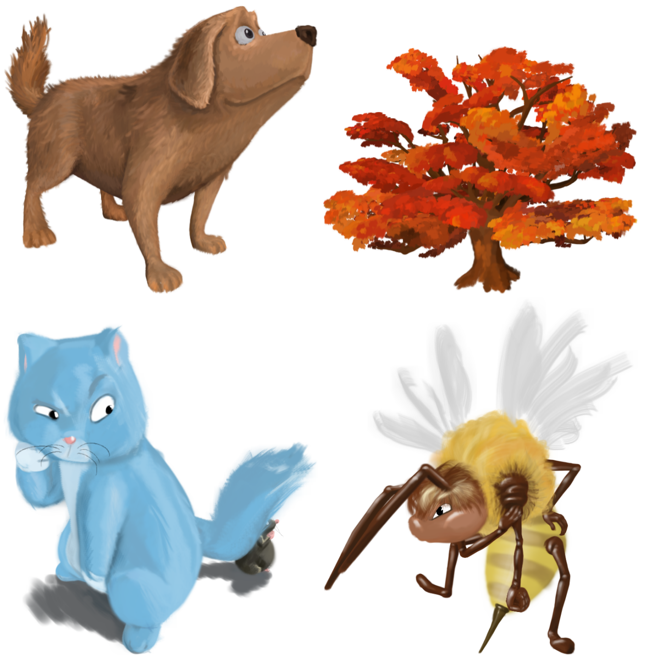

Figure 5: *Several 3D paintings rendered using mixed-order compositing. © Disney Enterprises, Inc.*

4 Discussion

4.1 Implementation and Results

We have tested mixed-order compositing on a variety of 3D paintings drawn in OverCoat [Schmid et al. 2011]. OverCoat generates splats by projecting stroke centerlines onto the 2D display and uniformly sampling them in 2D, placing a splat centered at each sample. These splats are then rasterized into fragments that are used as input by mixed-order compositing. The depth of each fragment is simply the depth of the center point of its splat.

Figure 5 shows some of our results. As demonstrated in our video, there are no popping or other noticeable artifacts in any of the scenes. Even scenes designed for the original OverCoat renderer look improved with mixed-order compositing. Figure 6 illustrates some of the artifacts that can be seen when depth offsetting or clustering is used to resolve the ordering conflict.

The scene statistics and timings for our implementations are given in Table 1. We have written two implementations of mixed-order compositing: a single-threaded C++ implementation (running on

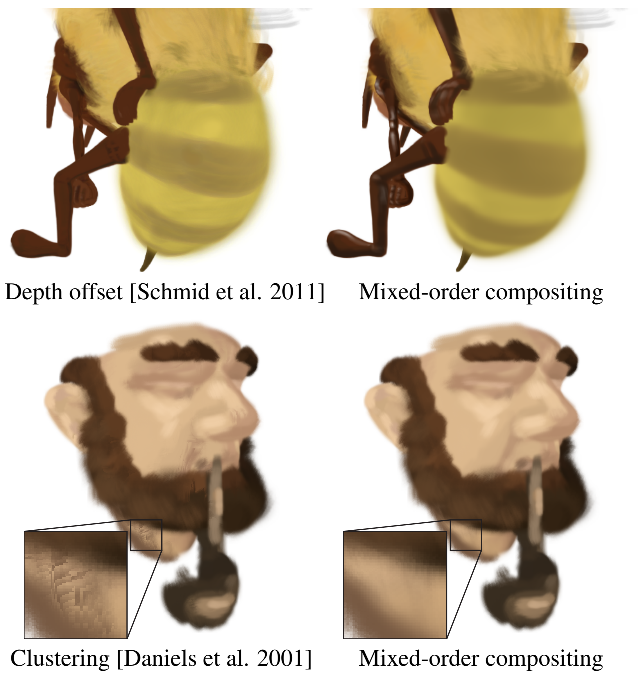
Depth offset [Schmid et al. 2011]
Mixed-order compositing
Clustering [Daniels et al. 2001]
Mixed-order compositing

Figure 6: *A comparison between existing methods and mixed-order compositing. The optimal parameters have been manually chosen for each algorithm. The left column shows artifacts on the bee's limbs and abdomen and on the captain's neck, mouth, and nose regions. These images are best viewed zoomed in. As our video demonstrates, all of these artifacts are time-varying.*

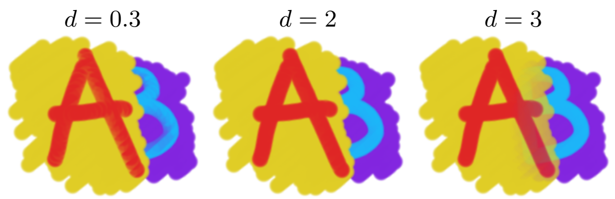

Figure 7: *Comparison of different values for the stroke order window size d. If the window is too small (left), the compositing algorithm is unable to resolve the stroke order on a surface properly. If the windows is too large (right), surfaces which are close in depth start to become blended in stroke order.*

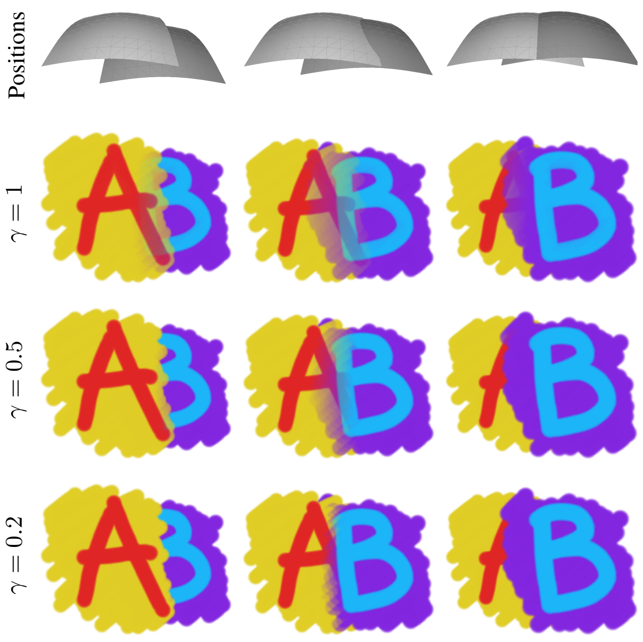

Figure 8: *Comparison of different values for the smoothing width parameter γ. A smaller value decreases the amount of blending in the image by increasing the speed of transitions between different visibility configurations.*

one core of a Core i7 2.8 GHz with 12GB of RAM) and a GPU implementation in GLSL (running on an NVIDIA GTX 480). The CPU implementation relies on a software rasterizer to produce the fragments. On the GPU, fragment generation takes less than 10% of the compositing time. Neither of the timing columns in Table 1 includes the fragment generation time.

4.2 Parameters and Temporal Coherence

We found it easy to choose the two parameters, d and γ. We used the following criterion to choose d: it should be large enough to ensure that strokes on a single surface are composited in stroke order, but no larger. We used $\gamma = 0.5$ for most of our examples. We show the effect of d in Figure 7 and of γ in Figure 8.

Mixed-order compositing achieves temporal coherence by blending different compositing orders. While this blending eliminates popping artifacts, it is not always artistically desirable. Our examples are painted in a style that works well with blending, but one can imagine 3D paintings whose artistic style would be compromised by it. The parameter γ allows the user to trade temporal coherence for reduced intermediate blending: a lower γ can minimize the frames with undesirable blending, while preventing hard "pops."

In some cases it may be useful to vary d with depth. For example, wide brush strokes may need to be composited in stroke order over a larger window than thin brush strokes. Our method could be extended to support a variable window size $d(z)$ as long as it satisfies the Lipschitz condition $|d(z_1) - d(z_2)| \leq |z_1 - z_2|$. This condition guarantees that no window completely contains another and allows the efficient computation of replacement colors.

4.3 Extensions, Limitations, and Future Work

For large scenes, where d is much smaller than the depth range, the running time can be improved to $O(n \log m)$ (assuming fragments are given in depth order), where m is the maximum number of fragments in an interval of length d. The bottleneck is the computation of $S(z)$, which can be sped up by maintaining the fragments to be composited in stroke order in a dynamic binary tree, such as a red-black tree or a splay tree instead of our static binary tree. We did not implement this version of the algorithm because we expect that, for our scenes, the higher hidden constants of the dynamic binary tree would eclipse the potential improvement. In terms of memory, all of our steps stream over depth, so by interleaving the stages of our algorithm, memory usage can be improved to $O(m)$. Practical avenues for further optimization include compositing nearby fragments in stroke order (with bounds on the maximum incurred error) and discarding fragments obscured by other fragments closer to the viewer.

Scene	Strokes	Splats	Fragments	Max Fragments/Pixel	Time (CPU)	Time (GPU)
Portrait	14k	256k	20M	783	5.5s	0.87s
Dog	29k	345k	65M	1301	20s	3.2s
Tree	21k	166k	33M	715	10s	1.4s
Cat	6.5k	158k	61M	1473	19s	3.2s
Captain	1.8k	23k	31M	645	9s	1.3s
Bee	20k	362k	121M	5077	46s	9.5s

Table 1: *Scene statistics and timings for our CPU and GPU implementations of mixed-order compositing with an output of 960x720 pixels.*

Our method assumes that fragments close together in depth are on the same surface and should thus be composited in stroke order. This works well the vast majority of the time, but it may lead to unintuitive results in cases where the artist has interleaved drawing on different surfaces and if the surfaces pass through each other. Although surfaces cannot be reliably identified in general, an interesting extension of our work would be to smoothly incorporate information about distinct surfaces when it is available.

For some applications, the stroke order is irrelevant and only a temporally-coherent depth-order compositing that satisfies Properties 2–4 is needed. For such a case, we can redefine

$$S(z) = \sum_{\{i|z-d/2<z_i<z+d/2\}} (c_i, \alpha_i)$$

and leave the rest of the algorithm unchanged. Together with the box filter, the effect is that the replacement color is the average of the original fragment colors weighted by a trapezoid. This method is similar to the soft depth compositing of Bruckner et al [2010], but can be computed in $O(n)$ time (because the tree is not necessary for sums) if the fragments are given in depth order.

5 Conclusion

3D paintings hold great potential as an expressive medium that allows an artist to make use of 3D structure without being bound by limitations of the traditional 3D pipeline. A sound rendering and compositing method is necessary for 3D paintings, especially animated scenes, to gain wider use and realize this potential. We have presented such a method and proved that it has several desirable properties. We have also presented an efficient algorithm to compute it, and demonstrated its effectiveness on both synthetic and real examples.

Acknowledgements

We thank Peter Kaufmann for feedback on the proofs, and Maurizio Nitti, Martin Sebastian Senn, and Alessia Marra for the 3D paintings.

References

BRUCKNER, S., RAUTEK, P., VIOLA, I., ROBERTS, M., SOUSA, M. C., AND GRLLER, M. E. 2010. Hybrid visibility compositing and masking for illustrative rendering. *Computers and Graphics 34*, 4, 361 – 369.

DANIELS, E., LAPPAS, A., AND KATANICS, G. T. 2001. *Method and apparatus for three-dimensional painting*. US Patent 6268865.

HAEBERLI, P. E. 1990. Paint by numbers: Abstract image representations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, 207–214.

HERTZMANN, A. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 453–460.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: Drawing strokes directly on 3D models. *ACM Transactions on Graphics 21*, 3 (July), 755–762.

KATANICS, G. T., AND LAPPAS, A. 2003. Deep Canvas: Integrating 3D Painting and Painterly Rendering. In *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems*, ACM SIGGRAPH 2003 Course Notes.

KEEFE, D. F., FELIZ, D. A., MOSCOVICH, T., LAIDLAW, D. H., AND LAVIOLA, JR., J. J. 2001. CavePainting: a fully immersive 3D artistic medium and interactive experience. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, ACM, 85–93.

LITWINOWICZ, P. 1997. Processing images and video for an impressionist effect. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 407–414.

LU, J., SANDER, P. V., AND FINKELSTEIN, A. 2010. Interactive painterly stylization of images, videos and 3D animations. In *Proceedings of I3D 2010*.

LUFT, T., AND DEUSSEN, O. 2006. Real-time watercolor illustrations of plants using a blurred depth test. In *NPAR 2006: Fourth International Symposium on Non Photorealistic Animation and Rendering*, 11–20.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics & Applications 9*, 4 (July), 43–55.

MCCANN, J., AND POLLARD, N. 2009. Local layering. *ACM Transactions on Graphics 28*, 3 (July), 84:1–84:7.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 477–484.

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

SCHMID, J., SENN, M. S., GROSS, M., AND SUMNER, R. 2011. Overcoat: An implicit canvas for 3D painting. *ACM Transactions on Graphics 30*, 4 (July), 28:1–28:10.

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

WALLACE, B. A. 1981. Merging and transformation of raster images for cartoon animation. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, 253–262.