Inbetweening with Occlusions for Non-Linear Rough 2D Animation

Melvin Even, Pierre Bénard, Pascal Barla

Inbetweening with Occlusions for Non-Linear Rough 2D Animation

Melvin Even, Pierre Bénard, Pascal Barla

RESEARCH REPORT
N° 9559
November 2024
Project-Teams Manao

ISSN 0249-6399 ISRN INRIA/RR--9559--FR+ENG

Inbetweening with Occlusions for Non-Linear Rough 2D Animation

Melvin Even*, Pierre Bénard*, Pascal Barla*

Project-Teams Manao

Research Report n° 9559 — November 2024 — 30 pages

Abstract: Representing 3D motion and depth through 2D animated drawings is a notoriously difficult task, requiring time and expertise when done by hand. Artists must pay particular attention to occlusions and how they evolve through time, a tedious process. Computer-assisted inbetweening methods such as cut-out animation tools allow for such occlusions to be handled beforehand using a 2D rig, at the expense of flexibility and artistic expression.
In this work, we extend the more flexible 2D animation framework of Even et al. [2023] to handle occlusions. We do so by retaining three key properties of their system that are crucial to speed-up the animation process: input rough drawings, real-time preview, and non-linear animation editing. Our contribution is two-fold: a fast method to compute 2D masks from rough drawings with a semi-automatic dynamic layout system for occlusions between drawing parts; and an artist-friendly method to both automatically and manually control the dynamic visibility of strokes for self-occlusions. Such controls are not available in any traditional 2D animation software especially with rough drawings. Our system helps artists produce convincing 3D-like 2D animations, including head turns, foreshortening effects, out-of-plane rotations, overlapping volumes and even transparency.

Key-words: 2D Animation, Occlusions

* Inria, Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800

RESEARCH CENTRE
Centre Inria de l'Université de Bordeaux

200 avenue de la Vieille Tour
33405 Talence Cedex

Interpolation en présence d'occultations pour l'animation 2D esquissée non-linéaire

Résumé : La représentation de mouvements 3D et de la profondeur par des dessins animés en 2D est une tâche notoirement difficile, qui demande du temps et de l'expertise lorsqu'elle est réalisée à la main. Les artistes doivent accorder une attention particulière aux occultations et à leur évolution dans le temps, un processus fastidieux. Les méthodes d'interpolation assistées par ordinateur, telles que les outils d'animation *cut-out*, permettent de traiter ces occultations à l'avance à l'aide d'un *rig* 2D, au détriment de la flexibilité et de l'expression artistique.

Dans ce travail, nous étendons le système d'animation 2D de Even et al. [2023] pour gérer les occultations. Nous le faisons en conservant trois propriétés clés de leur système qui sont cruciales pour accélérer le processus d'animation : les dessins esquissés en entrée, la prévisualisation en temps réel et l'édition non-linéaire de l'animation. Notre contribution est double : une méthode rapide pour calculer des masques 2D à partir de dessins esquissés avec un système de mise en profondeur dynamique semi-automatique pour les occultations entre les parties du dessin ; et une méthode conviviale pour l'artiste permettant de contrôler automatiquement et manuellement la visibilité dynamique des traits pour les auto-occultations. De tels contrôles ne sont pas disponibles dans les logiciels d'animation 2D traditionnels, en particulier pour les dessins esquissés. Notre système aide les artistes à produire des mouvements 3D convaincants à l'aide d'animations 2D, y compris des rotations de visages hors du plan, des effets de perspective, des volumes qui se superposent, et même de la transparence.

Mots-clés : Animation 2D, Occultations

Contents

1 Introduction **3**

2 Previous work **5**

3 System overview **6**

4 Animation structure **7**
4.1 Static layout . . . 7
4.2 Dynamic layout . . . 9
4.3 Stroke visibility . . . 10

5 Animation tools **11**
5.1 Layout editing tools . . . 11
5.2 Stroke visibility tools . . . 15

6 Results **17**
6.1 User interactions . . . 17
6.2 Case studies . . . 19
6.3 Performance . . . 22

7 Comparison with the system of Even et al. **24**
7.1 Non-linear editing with occlusions . . . 24
7.2 Usability study . . . 24

8 Discussion **25**

A Lattice expansion **27**

B Animation structures overview **27**

1 Introduction

Creating the illusion of 3D-like motion with animated drawings is one of the most difficult challenges of 2D animation, which as of today still relies extensively on the skills of 2D artists.

In traditional animation (either on paper or on screen), established workflows are used to produce such effects. For instance, in the solid drawing technique employed by Disney animators [2], rough drawings representing the main volumetric parts of a character are first animated, then relied upon to draw the outlines of the character, which helps convey depth and volume. Such an animation process not only requires intensive manual labor, which makes experimenting with different motions time-consuming as the artist needs to redraw every frame of the animation at each trial; but it also implies extensive training and expertise on the part of the animator.

An alternative approach is to rely on cut-out animation software, in which artists first build 2.5D rigs composed of 2D shapes organized in depth, then animate them directly through time. To convey depth, specific animations (e.g., out-of-plane rotations) must be predefined in the rig. This approach is often used in animated series as it greatly shortens production times, as well as in video games and live performance since rig controls may be easily mapped to various user inputs. However, cut-out animation suffers from a number of limitations: artists must work with a constrained workflow which is often restricted to character animation; and animation

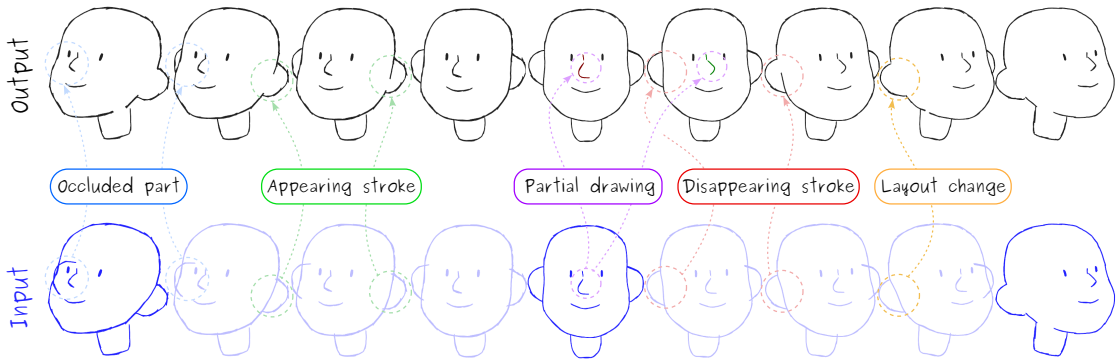


Figure 1: Starting from the head-turn animation in [1] (bottom), with input key drawings in dark blue and automatic inbetweens in light blue, we use different features of our system to produce an animation (top) with occlusions (e.g., ears). From left to right: masks are automatically computed for all drawing parts to create occlusions among them; a stroke is appearing through a temporal visibility threshold; a partial drawing of the nose is created at an intermediate frame (it is stored at a keyframe, but is made to appear later on via visibility); a stroke is made to disappear on the opposite ear, and it is suddenly interrupted by the "pop" of the right ear in front, via a layout change between keyframes. The final result conveys a strong 3D-like sense as shown in our video.

expressivity remains reduced unless a significant time is invested early on in the design of the rig, which once again requires a tremendous amount of manual work.

Our goal is to provide artists with a set of tools and algorithms that facilitate the creation of 3D-like motion, while retaining the flexibility of traditional 2D animation workflows. To guide the design of our method, we have conducted interviews with professional 2D animators, from which we have collected the following observations. First and foremost, the two main contributing factors to give the illusion of 3D shape are occlusions and perspective cues. In this paper we mostly focus on occlusions, which have been shown to play a central role in conveying motion in depth [3, 4]. Through our discussions with artists, we have identified three practical guidelines:

- **G1**: the handling of occlusions should not deviate significantly from animators workflow, with most of their time spent drawing or erasing lines.
- **G2**: since the illusion of 3D motion is conveyed at the very first stages of animation, occlusions should be conveyed given rough drawings as input.
- **G3**: occlusions should be made to happen at or between keyframes, and may be of two types: between object parts (e.g., a hand passing in front of another and hiding it), and self-occlusions (e.g., as in the folds of a piece of cloth).

We have decided to build our method on the 2D animation system of Even et al. [1] (presented in Section 3), as it retains the flexibility of traditional 2D rough animation while easing both the matching of drawings at key frames (key drawings) and their interpolation at intermediate frames (inbetweens). In addition, it allows artists to perform non-linear edits and provides real-time previsualization of rough animations.

However, this system does not handle any kind of occlusions. Our first contribution (Section 4) is to extend its structure to support the semi-automatic inbetweening of key drawings, using masks and dynamic layouts to handle occlusions among object parts and stroke visibility

attributes for self-occlusions. Our second set of contributions (Section 5) is to provide (semi-)automatic algorithms and tools to control this animation structure, while retaining non-linear editing functionalities and interactivity. In particular, we introduce (1) automatic layout propagation and layout change algorithms to produce dynamic occlusions among drawing parts (Section 5.1); and (2) automatic computation of stroke visibility attributes to let lines appear or disappear between keyframes (Section 5.2).

In Section 6, we describe a typical working session with our system, and demonstrate the effectiveness of our approach by reproducing complex 2D animations with occlusions using a minimal number of key drawings or user interaction (see Figure 1 and the supplemental video). This is confirmed in Section 7 through a usability study comparing our novel tools with the system of Even et al. [1]. We discuss the limitations of our method in Section 8.

2 Previous work

Previous work on computer-assisted 2D animation with occlusions can be classified into two main families of approaches: rig-based and stroke-based methods. In traditional 2D animation, occluded lines are either erased manually or masked through *matting*, a process in which drawings are broken down into layers ordered in depth, and each drawing element is represented as a silhouette to create solid areas that occlude underlying elements [5]. This method can be found in all modern 2D animation software [6, 7]

Rig-based methods. In rig-based systems, artists first create a rig over the drawing by defining controllers and deformations. Rigged models are then used to accelerate the animation process at the cost of less flexibility compared to traditional animation workflows, which require drawing every single frame. The "skeletal strokes" of Hsu et al. [8, 9] are probably the first rig of that sort demonstrating the ability to manually control and animate vector drawing deformations, including 3D-like effects with occlusions (using a surface removal trick). Following 2.5D animation systems allow artists to decompose 2D drawings into parts according to depth ordering, using 3D handles (e.g., bones, anchors) to guide their motion and deformation in 3D space.

Yeh et al. [10] use a simple two-sided drawing representation to produce 2.5D effects, including rolling, twisting, and folding. The method of Rivers et al. [11] estimates 3D anchor points to drive the motion of animated billboards. In their system, the user draws various viewpoints of the same object and place them in a predefined parameterized space which is then used to both blend the key drawings as well as to interpolate the 3D anchors. Coutinho et al. [12] use a 2.5D skeleton on top of the previous method to ease the creation of new poses. The work of Fukusato and Maejima [13] is in the same vein but it combines view-dependent deformations [14] with As-Rigid-As-Possible (ARAP) interpolation techniques [15].

Other methods use different types of proxies such as 3D skeletons [16]. E et al. [17] leverage Blender [18] "grease pencil" to provide artists with a 2.5D animation interface allowing the generation of simulated animations in a 3D environment that adhere to hand-drawn keyframes. Some commercial software use 2.5D methods to animate characters for cartoons [19, 20] or interactive performance [21].

Such 2.5D methods suffer from rigid workflows due to a strict decoupling between the rigging and the animation phases, making them closer to 3D animation (with flat 2D shapes) than traditional hand-drawn 2D animation (**G1**). In addition, despite depth information and occlusions, the resulting animations may still look overly flat, revealing their cut-out nature. Our method precisely addresses these two limitations.

Stroke-based methods. Stroke-based methods take as input clean-line drawings and either 1) use the strokes themselves to define occluding regions, or 2) use the stroke-graph representation of each drawing to handle the visibility of (sub)strokes.

In the first category, Jiang et al. [22] let the user define one-sided *boundary strokes* from which masks are automatically computed at keyframes and during the inbetweening process. In the system of Carvalho et al. [23], the user manually selects a sequence of strokes to define the contour of a region. These regions can then be colored to mask each other; region depth is manually specified at keyframes and interpolated for inbetween frames.

In the second category, Noris et al. [24] and Yang [25] handle occlusions by drawing occluded strokes or manually tagging their visibility at the substroke level, which is then propagated through time using the stroke graph. Dalstein et al. [26] extend such a representation with a space-time data-structure that stores time-continuous topological events (merge, split, appearance, or disappearance) throughout the animation. The drawing interpolation problem is thus solved by construction, but the structure itself must be built manually, which is little intuitive and time-consuming. Similarly to our system, the layout of the drawings may be specified at keyframes with a discrete depth-ordering. However, unlike ours, the layout cannot change inbetween keys during the interpolation, and no tools are provided for its automatic propagation to adjacent keyframes.

When starting from existing cell animations, dedicated region tracking algorithms [27, 28] have been proposed for editing propagation or temporal super-sampling. They manage to automatically find spatio-temporal correspondences between regions across a full animation sequence with an existing but implicit layout in the presence of partial and even complete occlusions. We tackle an almost inverse problem: starting from already registered key drawings, we aim at propagating the layout of one of these keys to the others.

Stroke-based methods are more in line with the traditional 2D animation pipeline than rig-based approaches, but they can only be applied after the cleaning step, on the final animation, which prevents their use at earlier, more creative stages of the animation process (**G2**). In contrast, we would like to handle rough drawings that are typical of the motion design stage, and we want to avoid creating a global structure (e.g., a stroke-graph) of the drawings or animation. These goals guided our choice for the non-linear animation system of Even et al. [1] based on transient embeddings that we describe in Section 3. It seems to us a more appropriate starting point than the system of Xing et al. [29] that purely works at the stroke level and thus makes drawing layout hard to define.

3 System overview

Previous system. The core of our system (matching and interpolation of the key drawings) is based on the 2D animation system of Even et al. [1], whose source code is available online [30].

In this system, the user draws sparse and rough key drawings that are manually decomposed into a set of transient embeddings. Each transient embedding corresponds to a drawing part, i.e., a group of strokes (polylines) that usually share a common semantic and similar motion, and are embedded in the same 2D square lattice. The lattice is built axis-aligned, and each quad cell stores the stroke vertex indices that it contains.

Drawing parts of consecutive keyframes are semi-automatically registered using ARAP matching [31] and inbetweened with ARAP interpolation [32]. The matching process can adapt to various traditional animation workflows such as pose-to-pose (keyframes are drawn then matched together) and shift-and-trace (keyframes are drawn sequentially, with each one being traced over a deformation of the previous keyframe). The system is non-linear in the sense that the matching

and the interpolation dynamics (i.e., the *timing* and *spacing*) can be modified in a non-destructive fashion in any order or workflow. The *spacing* (i.e., the distribution of the inbetween frames in time) of the animation can be controlled via a chart-like interface. Spatial and temporal continuity of the animation may be locally enforced using trajectory constraints connecting multiple transient embeddings, allowing for continuity preservation across topological changes between key drawings.

Nevertheless, the major limitation of the system of Even et al. is the fact that drawing parts cannot occlude each other. Furthermore, topological changes (e.g., stroke appearance or disappearance) can only occur at keyframes which is problematic for 3D-like motion since many elements of the drawing should potentially appear or disappear during interpolation (**G3**).

Our additions and necessary structural changes. To lift these restrictions, we extend their animation structure in several ways. First, at keyframes, we compute masks for rough drawings parts and let the user order them in depth, forming the *static layout* of key drawings (**G2**). Second, at any inbetween frame, we support adding *dynamic layout changes* that provide artists with controls over occlusions among drawing parts (**G3**) without sacrificing non-linear editing. Finally, we introduce *temporal visibility thresholds* to control the gradual appearance and disappearance of stroke vertices within embeddings during the course of the interpolation. These three structural extensions are described in Section 4. Figure 22 presents an overview of our new structures and how they interplay with the structures of Even et al. [1].

The ways we instantiate and manipulate this new animation structure are described in Section 5. Our general approach is to suggest changes to the animation structure, which if required may be non-linearly edited through tools compatible with traditional animation workflows (**G1**). The main advantage of our approach is that animators do not have to learn how to manipulate a complex structure, as opposed to methods based on rigs or stroke graphs.

More specifically, we present two automatic algorithms to propagate the layout of a given key drawing to adjacent keyframes, but also to determine whether a layout must change between two key drawings to minimize visibility changes and, if so, at what interpolation time. We additionally propose stroke visibility tools to automatically estimate temporal visibility thresholds based on the key drawing content, and to manually specify or refine them with a painting metaphor. We show in the supplemental video and in Section 6 how these tools are used in practice and work in conjunction with those of Even et al., and how they allow to convey occlusions in typical animation cases.

4 Animation structure

In this section we describe the extensions we made to the animation structure of Even et al. [1] to handle occlusions. In Section 4.1, we explain how occlusions are handled from rough drawing parts, using masks and layouts. Occlusion between keyframes are permitted through dynamic layout changes for occlusions among parts, and through stroke visibility attributes for self-occlusions, as described in Sections 4.2 and 4.3 respectively.

4.1 Static layout

Mask creation. To create occlusions among parts, we first need to identify their interiors. We thus pre-compute a *mask* per drawing part, which consists of a 2D polygon that approximately encloses all of its strokes. Previous methods (e.g., [23]) use clean lines as support for the mask outline, and thus cannot handle rough drawings. "Alpha Contours" [33] could be used to compute

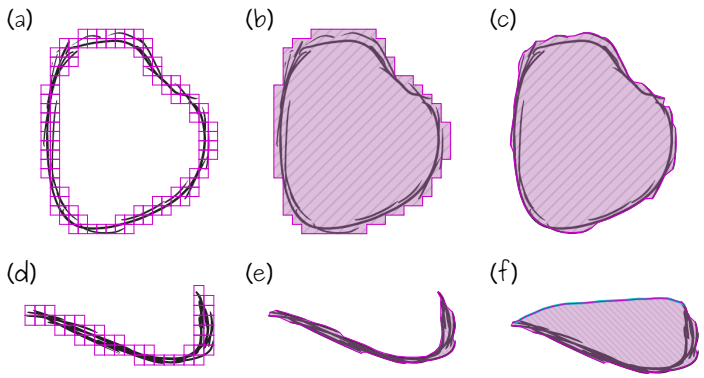


Figure 2: Mask creation. (a,d): input drawing parts overlaid by their lattice (in magenta). The coarse mask in (b) is computed from the exterior boundary of the lattice, and automatically refined by projecting the lattice boundary vertices onto the strokes to yield the tight mask in (c). The tight mask in (e) only encloses the strokes in the drawing part. In (f), an invisible line has been sketched, yielding a new mask.

a tight enclosing of the positive space of a sketchy drawing, but it is computationally expensive and unnecessarily complex in our case since we do not need masks with holes. We thus propose a simpler algorithm that leverages the lattice to compute masks in real time.

We assume that the lattice has a sufficient resolution to capture the general silhouette of the drawing part, otherwise its resolution may be increased independently for each part. The exterior boundary of the lattice already forms the outline of a coarse but conservative mask (Figure 2(b)). It is efficiently extracted by walking along boundary edges starting from one of the quad corners with minimum abscissa. We then refine this initial mask by projecting its outline onto the enclosed strokes (Figure 2(c)). More precisely, we move every quad corner that is part of the exterior boundary to the position of its closest stroke vertex inside the quad. The interior of the resulting polygon is then tessellated using the algorithm of the OpenGL Utility Library (GLU) as described by Shreiner [34]. Note that the polygon might be self-intersecting, but this is not an issue for mask rendering.

Optionally, the artist may manually refine the mask by drawing its silhouette, which is especially useful to close open parts (Figure 2(e,f)). In practice, silhouette strokes are simply added to the lattice, potentially expanding it, before recomputing the mask. These strokes are never drawn, except when editing the mask.

Drawing layout. With all the parts of a key drawing equipped with their mask, we next need to deal with the way they overlap, which we call the drawing layout. To this end, parts are partially ordered with *discrete* depth levels. Parts at the same depth do not mask each others, which is useful in the earliest stages of the animation design process where occlusions are absent (i.e., all parts lie at the same discrete depth). Parts at a given depth occlude the strokes of parts with a strictly lower depth level as well as parts from lower layers. The major advantage of such a representation over continuous depth values is that we do not need to consider the depth of a part relative to the whole drawing, but solely to the parts it intersects, requiring less user intervention.

As shown in Figure 3, the depth ordering is stored at keyframes as an ordered list of sets of parts. The size of this list is the number of discrete depth levels. When a key drawing is first drawn, the list is made of a single set holding all the drawing parts; hence no occlusion

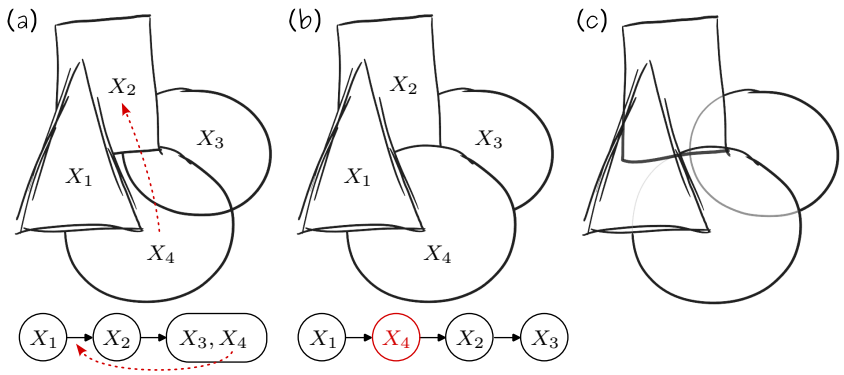


Figure 3: Drawing layout. We start in (a) with a layout where part X_1 is in front of X_2, and parts X_3 and X_4 are both in the background. This is represented by the list at the bottom. Part X_4 is then moved in front of X_2 (red arrow). The result is shown in (b) with the default occluded line style. We change the style per part in (c), varying thickness and opacity.

occurs. The layout may be modified through a tool inspired by the interaction metaphor of Sykorà et al. [35]. This is shown in Figure 3(a,b) and the supplemental video: the user may move the currently selected part in front of, behind, or at the same depth level of a second part by clicking on it and using keyboard modifiers.

Stroke masking. When rendering a key drawing, its parts are drawn from farthest to closest according to the drawing layout. To account for the cumulative effect of the masks, we progressively store them in an offscreen buffer that acts as a stencil. We proceed in two back-to-front passes. We first render all masks in the offscreen buffer storing their discrete depths and optional intensity values. In a second pass, we render all strokes part-by-part in the main framebuffer, sampling the stencil to check whether it is occluded and modulating its attributes (e.g., opacity, line thickness) accordingly. This rendering pipeline is fully implemented on the GPU using Geometry and Fragment shaders with two passes per discrete depth.

To support displaying hidden lines in various styles, the masks and the offscreen buffer can optionally store intensity values in the $[0, 1]$ range. The offscreen buffer is then updated using the `maximum` OpenGL blending mode. The choice of style for (partially) occluded strokes is made per drawing part, as shown in Figure 3(c). This is particularly useful during the first stages of animation, where hidden lines may be shown in a different style even when fully occluded to better read motion.

4.2 Dynamic layout

Layout inbetweening. Once the parts of each key drawing have been ordered with occlusions at keyframes, we need to define how the drawing layout behaves through interpolation.

Similarly to the strokes they enclose, masks are embedded in their corresponding lattice. They thus automatically follow the lattice deformation at intermediate frames, as shown in Figure 4. The system of Even et al. [1] optionally uses a thickness-based cross-fading of strokes between keyframes, whereby strokes of the next key drawing are propagated backward in time. We provide a similar option by propagating the mask of the next keyframe backward and cross-fading the opacities of forward and backward masks. Since we use the `maximum` blending mode for masks, fully opaque parts at a key drawing yield fully opaque inbetweened parts where they intersect.

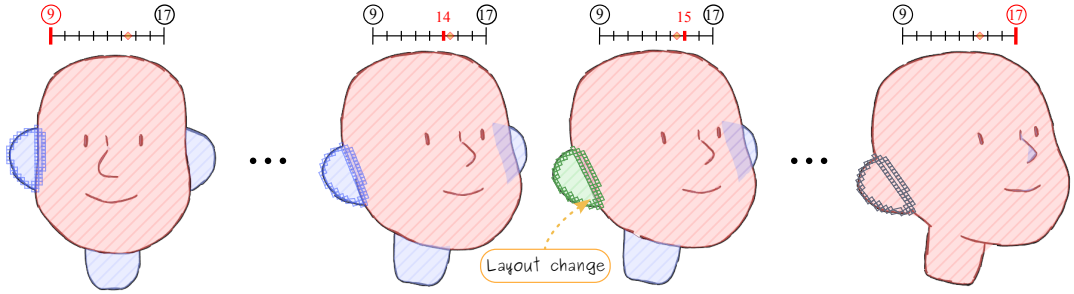


Figure 4: Layout inbetweening. Breakdown of the interval between the second and third keyframe of the animation in Figure 1, focusing on the right ear. The spacing charts (top row) indicate the frame number of each drawing (highlighted in red, keyframes are circled). The first keyframe is decomposed into 8 parts at two depth levels, whose masks are respectively colored in red and blue. The second keyframe is made of 5 parts, all at the same depth (in red). The axis-aligned lattice at the first keyframe (in blue) is deformed up to the second keyframe, and shown in gray since this drawing part is then not visible anymore. The associated mask is carried out by the deformation at intermediate frames, and brought to the front by a layout change located at $t = 0.6$, which occurs between frames 14 and 15 (orange diamond on the spacing charts), and thus becomes effective at frame 15 (lattice and mask colored in green).

Layout change. By default, the drawing layout remains constant throughout interpolation, and only changes at keyframes. However, it is hardly ever desirable to define a new keyframe every time the depth level of a given part changes. We thus support layout changes that may occur at any intermediate frame between two keyframes. These layout changes are not recorded at absolute frame numbers but relative to the linear interpolation time $t \in [0, 1]$ between two contiguous keyframes. This has the advantage of precisely locating the layout change at a point in space (e.g., along a motion trajectory), while remaining unaffected by timing or spacing. The layout change occurs at the frame f that follows the relative time t (frame 15 in Figure 4). When an intermediate frame is rendered, we assign the current layout to that of the previous layout change if there is one, or else to the layout stored at the previous keyframe. Our dynamic layout approach provides an explicit control in time, in contrast to previous approaches (e.g., [11] and [23] for rig- and stroke-based methods respectively) where ordering emerges from depth interpolation of drawing parts.

4.3 Stroke visibility

In addition to global layout changes, contours may grow or shrink to convey self-occlusions due to camera motion or object deformation. To capture this effect with sketchy drawings, we need a way to make a group of strokes (representing one or multiple contours) gradually appear or disappear during interpolation.

Our solution is to assign a *temporal visibility threshold* $\nu_i \in [-1, 1]$ to each vertex of a stroke, with $\text{sign}(\nu_i)$ encoding whether the i^{th} vertex should appear or disappear, and $|\nu_i|$ storing the time when this event should occur. The visibility $V_i(t)$ of a vertex i at interpolation time $t \in [0, 1]$ between keyframes is a Boolean given by $t \geqslant \nu_i$ for appearing vertices and $t < -\nu_i$ for disappearing vertices.

By default, $\nu_i = 0$ which means the vertex is always visible. When all the vertices of a stroke have the same threshold $\nu_i \neq 0$, it will "pop" in or out at frame $|\nu_i|$. For a progressive appear-

Notation	Description
X/Y	Two successive key drawings, i.e., set of stroke vertices
X_m^t	m^{th} drawing part of X deformed at time t (omitted if $t=0$)
L_X	Layout of X, i.e., ordered list of its drawing parts
$\mathbf{x}_i$	2D position of the i^{th} stroke vertex
τ_i	Radius of the i^{th} stroke vertex
ν_i	Visibility threshold of the i^{th} stroke vertex (in $[-1,1]$)
$N_Y(\mathbf{x})$	Set of vertices in Y that are within a threshold distance of $\mathbf{x}$
$\text{vis}_i(L_X)$	Indicates whether the i^{th} stroke vertex is visible with L_X applied (in $\{0,1\}$)
$\mathcal{V}_{X,Y}(L_X, L_Y)$	Stroke visibility changes metric between layouts L_X and L_Y
X_{vis}^t	Set of visible vertices in X a time t
$X_{\text{out}}(Y)$	Set of vertices in X that are at least a minimal distance away from Y
$X_{\text{seed}}(Y)$	Set of vertices in $X \setminus X_{\text{out}}(Y)$ that should appear first.

Table 1: Table of notation.

ance/disappearance of strokes throughout interpolation, visibility thresholds should smoothly vary in screen space (and hence across multiple strokes). This is exemplified in Figure 14, where some stroke vertices are made to disappear during a travelling motion to convey levels of detail.

5 Animation tools

We now introduce animation tools whose purpose is to control dynamic layouts (Section 5.1) and stroke visibility (Section 5.2). The full animation is updated in real-time thanks to an efficient implementation, detailed in Section 6.3.

5.1 Layout editing tools

In the following, we assume that at least one non-trivial drawing layout (such as in Figure 3) has been given at a key drawing.

Automatic layout propagation. The layout of a key drawing X is propagated to the next key drawing Y using a mapping between drawing parts from X to Y. In general, this is a many-to-one mapping since adjacent key drawings may be composed of different numbers of parts. For each pair of parts X_m in X and Y_n in Y, we measure how much X_m is covered by Y_n *at the key drawing* Y. To this end, we transform the strokes of X_m to their target position at interpolation time $t=1$ using their lattice deformation and we resample them uniformly, denoting the result by X_m^1. We then iterate over all the stroke vertices $\mathbf{x}_i \in X_m^1$ and count the number $N_{Y_n}(\mathbf{x_i})$ of vertices in Y_n encountered within a spatial window based on the local stroke thickness $\tau(\mathbf{x_i})$, using:

$$N_Y(\mathbf{x}) = \{\ \mathbf{y}_j \in Y\ /\ \|\mathbf{x}-\mathbf{y}_j\| < \tau(\mathbf{x}) + 2\ \}, \tag{1}$$

where thickness is measured in pixels, and we have added a tolerance of 2 pixels to handle very thin lines. The part X_* with the highest coverage is then selected for the mapping, and Y_n receives its depth level. We use the same approach to propagate a layout to a previous key drawing, using the inverse mapping. The process may also be repeated multiple times to propagate a layout throughout an animation.

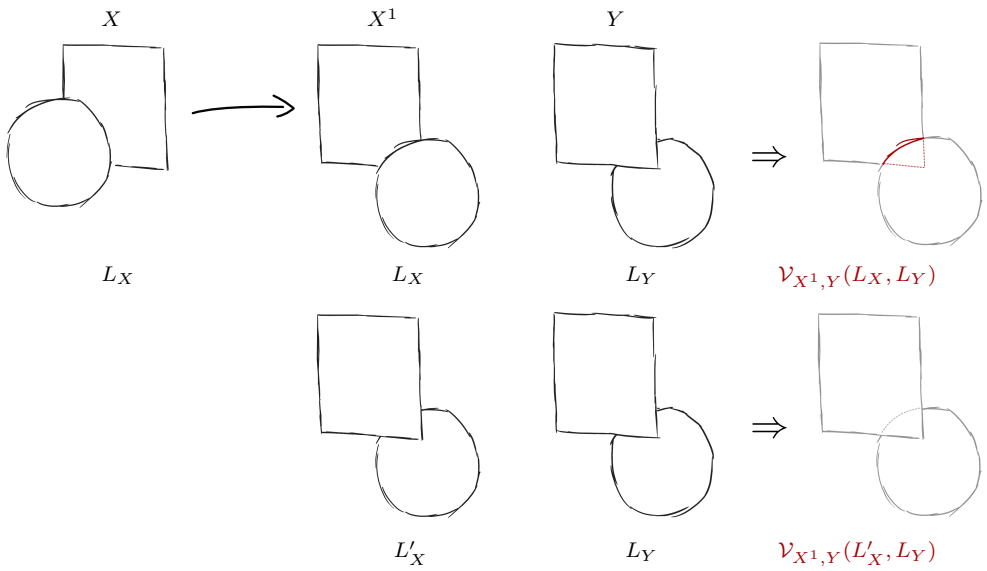


Figure 5: Visibility changes. We compute the baseline visibility changes score $\mathcal{V}_{X^1,Y}(L_X, L_Y)$ between the key drawing Y and the deformed version of the key drawing X at $t = 1$ (here, a translation of the circle) using their original layout (first row). The strokes highlighted in red corresponds to portions of the two key drawings whose visibility changes. With our heuristic layout L'_X (second row) there are no visibility changes.

Automatic layout change. Given two key drawings X and Y and their respective layouts, we propose an algorithm to detect automatically whether the layout of X must be modified to minimize visibility changes during the course of interpolation towards Y. We write L_X to denote the ordered list of drawing parts at X, as shown at the bottom of Figure 3. Therefore, starting from L_X and L_Y, our goal is twofold: find an optimal layout change L^*_X that produces the smallest amount of stroke visibility changes; and determine the best interpolation time t_* for this change to occur. In case L_X is already optimal, the default layout change should occur at $t = 1$.

We measure visibility changes between two layouts for two key drawings X and Y using:

$$\mathcal{V}_{X,Y}(L_X, L_Y) = \sum_{\mathbf{x}_i \in X} \sum_{\mathbf{y}_j \in N_Y(\mathbf{x}_i)} \frac{|\text{vis}_i(L_X) - \text{vis}_j(L_Y)|}{|N_Y(\mathbf{x}_i)|}, \tag{2}$$

where $\mathbf{x}_i \in X$ denotes a stroke vertex from any drawing part in X, $\text{vis}_i(L_X) \in \{0, 1\}$ indicates whether the i^{th} stroke vertex is visible according to L_X, and the set $N_Y(\mathbf{x}_i)$ identifies the set of stroke vertices of Y in the neighborhood of $\mathbf{x}_i$ (see Equation 1).

In principle, we would like to find a layout L^*_X that minimizes $\mathcal{V}_{X^1,Y}(L^*_X, L_Y)$, with X^1 the set of stroke vertices in X transformed at their position at $t = 1$. In practice, this is not feasible as the size of the search space is factorial in the number of drawing parts, and multiple optimal solutions exist since a layout is only a *relative* ordering of parts. We therefore use the following heuristic: we compute a possible layout change L'_X using the many-to-one mapping described previously for layout propagation. This layout change is relevant if it produces less stroke visibility changes than the current layout, which happens when $\mathcal{V}_{X^1,Y}(L'_X, L_Y) < \mathcal{V}_{X^1,Y}(L_X, L_Y)$ (see Figure 5). If the matching between key drawings is precise enough, we observed that the hybrid and optimal layouts are equivalent (i.e., $\mathcal{V}_{X^1,Y}(L'_X, L_Y) = \mathcal{V}_{X^1,Y}(L^*_X, L_Y)$), which happens in nearly all our

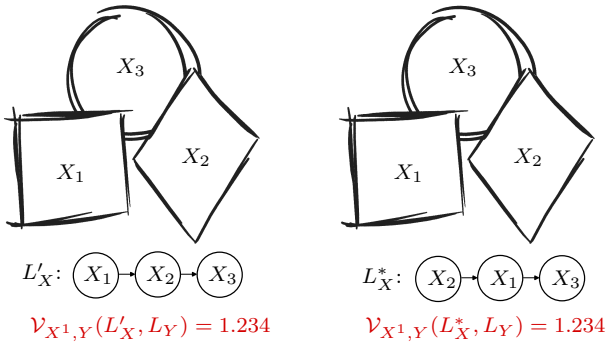


Figure 6: Equivalent layout changes. Since the drawings parts X_1 and X_2 do not overlap, their relative order does not affect the visibility change score. Their order may thus be swapped between our heuristic layout L'_X (left) and the optimal layout L^*_X (right).

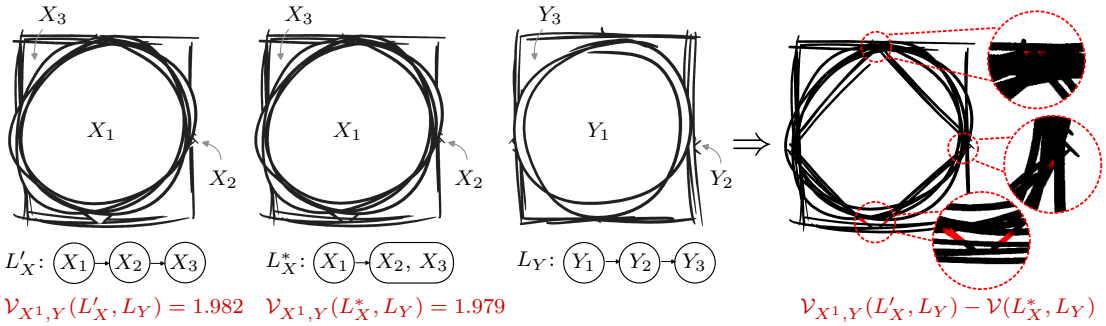


Figure 7: Heuristic vs. optimal layout change. In the optimal layout L^*_X (second column) the parts X_2 and X_3 are at the same depth level, whereas our heuristic layout (first column) separates them since, in the second key drawing (third column), their matching parts Y_2 and Y_3 are on different levels in layout L_Y. Due to the tolerance when measuring coverage, drawing sketchiness and slight misalignment in the matching process, there is a small score difference between L^*_X and L'_X. We highlight in red (last column) the difference between the visibility change score of L' and L^*.

tests (e.g., Figure 6). Otherwise, when key drawings are poorly matched or very sketchy, the layouts might differ but the difference in terms of visibility changes is minimal (see Figure 7).

When a layout change is detected, we also compute its optimal interpolation time t_* by applying L'_X to every interpolated time t corresponding to a frame f, keeping the one that produces the smallest amount of visibility changes (see Figure 8):

$$t_* = \arg\min_t \mathcal{V}_{X^t,X^t}(L_X, L'_X)\,. \tag{3}$$

As demonstrated in the supplemental video, artists may still edit the time t_* of the layout change directly on the spacing chart interface of Even et al. [1] (shown in Figure 4) or alternatively on motion trajectories, which complies with traditional animation workflows. Figure 9 demonstrates a typical use case of our layout tools.

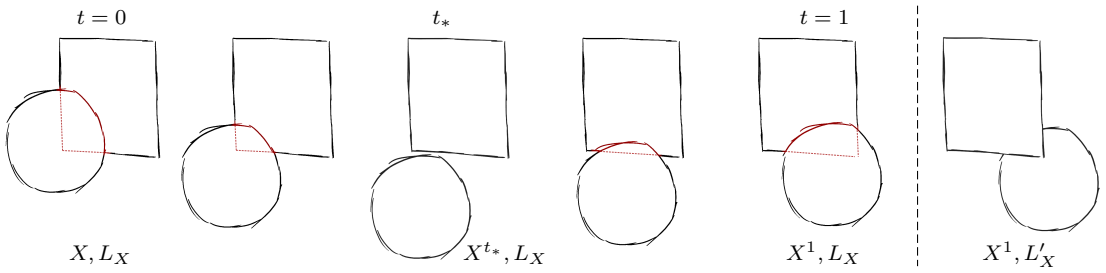


Figure 8: Optimal interpolation time. Once a layout change L'_X has been found, we select the optimal interpolation time t_* at which the layout L_X switches to L'_X by finding the frame that produces minimum visibility changes using Equation 3. The strokes highlighted in red corresponds to portions of the deformed key drawing whose visibility changes. In this case, we have $\mathcal{V}_{X^{t^*},X^{t^*}}(L_X, L'_X) = 0$.

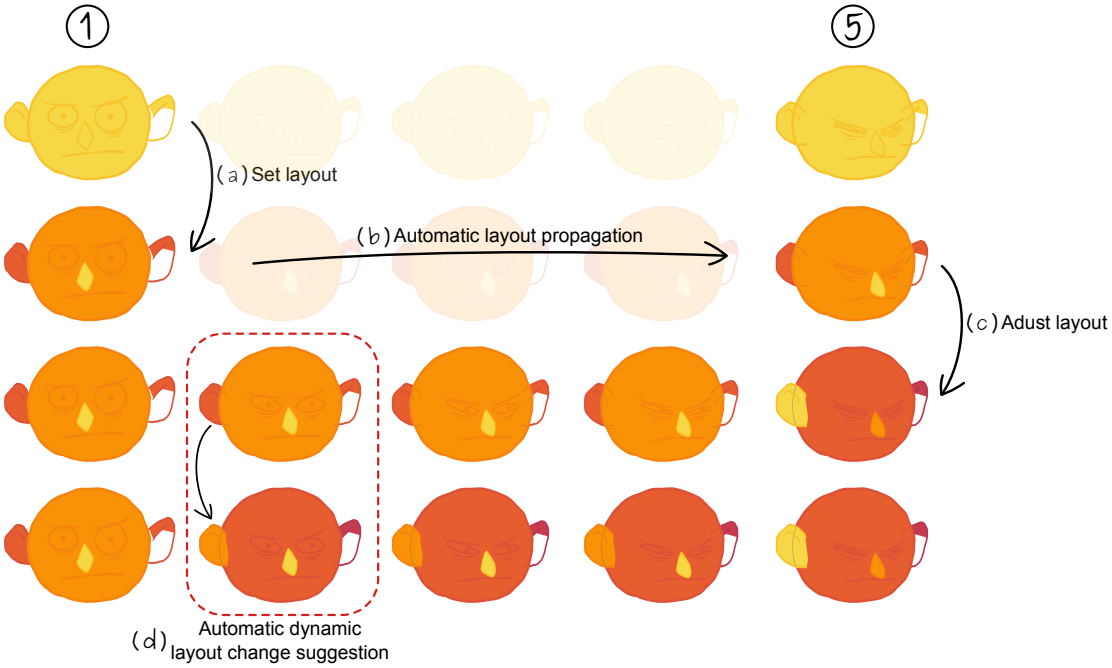


Figure 9: Layout tools overview. This example shows a typical use case of our layout tools. (a) Starting from two keyframes with no provided layout (top row), the user sets the static layout of the first keyframe. (b) The new layout is then automatically propagated to the second keyframe upon user request. (c) To refine the layout, the user adjusts the ear on the left to appear above the other elements in the second keyframe. However, this change does not affect the layout of the first keyframe, resulting in the ear suddenly changing depth just before the second keyframe. (d) Our automatic layout change algorithm suggests a layout change at the most appropriate time, if necessary. In this case, frame 2 is selected as it presents minimal overlap between the ear and the face, which reduces the amount of stroke segments suddenly appearing or disappearing.

5.2 Stroke visibility tools

By default, all stroke visibility thresholds are set to $\nu_i = 0$ (strokes fully visible). We now introduce tools that change ν_i values so that strokes progressively appear or disappear through interpolation.

Automatic visibility thresholds. When interpolating a key drawing from one key drawing to the next, portions of strokes may appear or disappear suddenly. Since this is often undesirable, we propose an automatic algorithm to gradually reveal or hide these portions.

First consider the simpler case of disappearing vertices. We identify by $X_{\text{out}}(Y)$ the set of vertices in key drawing X that must disappear when matched with key drawing Y:

$$X_{\text{out}}(Y) = \left\{ \mathbf{x}_i \in X^1_{\text{vis}} \;/\; \|\mathbf{x}_i - \mathbf{y}_j\| \geq \epsilon \;,\; \forall \mathbf{y}_j \in Y_{\text{vis}} \right\}, \tag{4}$$

with X^1_{vis} the set of visible vertices in X at time $t = 1$ based on the current layout L_X, and Y_{vis} the set of visible vertices in Y (at time $t = 0$) based on L_Y. We use $\epsilon = \sqrt{2}\tau(\mathbf{x})$ to account for small gaps between strokes. Geometrically, X_{out} corresponds to the stroke vertices of Y that are outside a dilated version of the key drawing X transformed at $t = 1$ (shown in blue in Figure 10(c)).

By default, we propose to make stroke vertices in X_{out} disappear based on their proximity to strokes in Y, i.e., the farthest vertices from Y should disappear first. To obtain such a behavior, we compute spatially-smooth temporal visibility thresholds ν_i through an isotropic diffusion process that starts at a set of vertices X_{seed} acting as sources with $\nu_i = 0$ and progresses until all vertices in X_{out} are processed. $X_{\text{seed}}(Y)$ (shown in red in Figure 10(c)) is given by:

$$\begin{aligned} X_{\text{seed}}(Y) = \{\, \mathbf{x}_i \in X^1_{\text{vis}} \;/\; &\|\mathbf{x}_i - \mathbf{y}_j\| < \epsilon \;,\; \forall \mathbf{y}_j \in Y_{\text{vis}} \\ &\wedge \|\mathbf{x}_i - \mathbf{x}_o\| < \epsilon \;,\; \forall \mathbf{x}_o \in X_{\text{out}}(Y) \;\}, \end{aligned}$$

The first condition ensures that seed vertices are close to the key drawing Y, and the second condition makes sure that they are close to vertices of X that must disappear (i.e., X_{out}). Note that $X_{\text{out}}(Y) \bigcap X_{\text{seed}}(Y) = \emptyset$. We now assign visibility thresholds $\nu_i \in [0, 1]$ to all vertices i in X_{out} through isotropic diffusion (see Figure 10(d)), using:

$$\nu'_i = \min_{\mathbf{x}_s \in X_{\text{seed}}} \|\mathbf{x}_i - \mathbf{x}_s\|, \text{ and } \nu_i = \frac{-\nu'_i}{\max_k \nu'_k}. \tag{5}$$

Note the negative sign before ν'_i since we are dealing with disappearing vertices and thus negative visibility thresholds.

For appearing stroke portions, we first need to identify the set of appearing stroke vertices *in the key drawing* Y based on its matching with the previous key drawing X. This amounts to identifying $Y_{\text{out}}(X)$ which is defined similarly to $X_{\text{out}}(Y)$ in Equation 4, except that the process is applied backward in time, from Y to X.

We then need to transport this set of appearing vertices backward to the key drawing X for interpolation. We first segment the vertices in $X_{\text{out}}(Y)$ into connected components. Then, on each of these components, we apply the following process. If the stroke vertices are covered by an existing lattice, we simply store a copy of these vertices in the lattice using its inverse transformation. When this is not the case, we first attempt to extend every lattice in X that overlaps with the current connected component until its vertices are fully covered, using the lattice expansion algorithm detailed in A. If this fails, we create a new drawing part for the current connected component. We eventually try to estimate its motion toward the previous

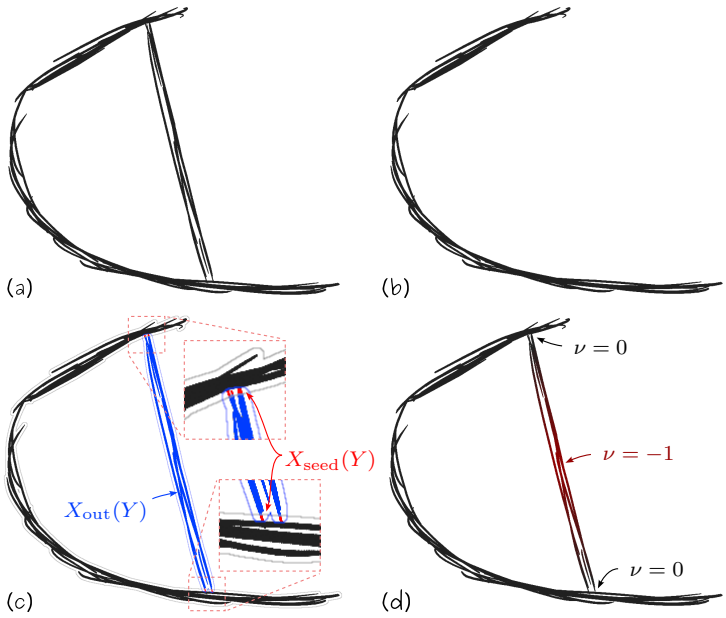


Figure 10: Stroke visibility. (a,b) Input key drawings X and Y. (c) Computation of $X_{\text{out}}(Y)$ and $X_{\text{seed}}(Y)$, the tolerance ϵ around strokes is visualized with thin lines. (d) Stroke visibility thresholds ν are automatically computed through isotropic diffusion from $X_{\text{seed}}(Y)$ over all of $X_{\text{out}}(Y)$, with their values visualized in red.

keyframe by placing motion constraints at the intersection with the original lattice, and using ARAP regularization [31] for interpolation at unconstrained vertices.

Once all appearing vertices have been copied into key drawing X, we apply the previously described isotropic diffusion process, but using positive visibility thresholds in Equation 5. Figure 11 demonstrates the application of this algorithm on the final two keyframes of the head-turn animation from Figure 1.

Painting tools. The isotropic diffusion process used to estimate visibility thresholds may fail in tricky configurations, such as the spiral animation shown in the supplemental video. We then resort to a more direct editing approach through a visibility gradient tool, as illustrated in Figure 12. Using a brush metaphor, the user paints a guiding path along which strokes should appear or disappear. Vertices inside the brush stroke footprint are projected onto the guiding path and assigned the threshold $\nu(u) = (1-u)\nu^{\text{start}} + u\nu^{\text{end}}$ at the parametric arc-length location u. As shown in the supplemental video, we provide additional controls to adjust ν^{start} and ν^{end}, the thresholds at the extremities of the guiding stroke.

An even more direct artistic control is provided through temporal painting tools, which are applied at interpolated frames. We first provide a temporal eraser tool, which instead of erasing vertices, sets their threshold to $\nu = -\frac{f-X}{Y-X}$, where f is the current frame, and X and Y are the previous and next key drawings. As a result, these stroke vertices disappear at frame f. Conversely, the temporal pen tool allows artists to draw *new* strokes at an intermediate frame f, effectively creating a *partial drawing*. These strokes are embedded in the deformed lattice then carried out to the previous key drawing X by applying the inverse lattice transform, and their visibility thresholds are set to $\nu = \frac{f-X}{Y-X}$. This is used on the nose in Figure 1 for instance.

Figure 11: Automatic visibility thresholds. The default interpolation of the two input keyframes (top row) results in the sudden disappearance of strokes vertices *at* the second keyframe. Our algorithm automatically assigns visibility thresholds to keyframe 1 (shown in red), resulting in the gradual disappearance of stroke vertices during the animation (bottom row).

6 Results

We demonstrate our animation system in a three-parts supplemental video, starting with complex animation results, followed by a detailed explanation of how the tools of Section 5 work, and concluded by comparisons with the previous system of Even et al. [1].

In this section, we provide details on how the user typically interacts with our system (Section 6.1), on how we construct complex animation results (Section 6.2) and performance measurements for the core algorithmic steps (Section 6.3).

6.1 User interactions

Matching and interpolation. In the system of Even et al. [1], users are allowed to mix and match multiple animation workflows. For example, using a "pose-to-pose" workflow, they can start by drawing at least two key drawings and then register them using semi-automatic tools, manually decomposing them into drawing parts when needed. Alternatively, they can use a "shift-and-trace" workflow: sketch a first key drawing, deform it using manual tools, and then either copy the deformed version of the key drawing to a new keyframe or redraw on top of it. During any of these matching interactions, the interpolated frames are updated in real time, allowing instant playback and "onion-skin" visualization. Users can further refine the interpolation by adding trajectory constraints (as Bézier curves) between keyframes and between drawing parts. They can also specify the dynamic of the animation (acceleration and deceleration) with a spacing chart interface. For more details regarding drawing, matching and interpolation interactions please refer to the supplemental video of Even et al. [1].

In the following, we describe the user interactions related to our novel contributions which are also demonstrated in the second part of our supplemental video. Our new tools can be used in conjunction with those of Even et al. [1] in a non-linear fashion (i.e., before or after matching, manipulation of the timing and spacing, and with any workflow); the only requirement is to have at least one drawing part.

Mask completion tool. When using any layout tool, we display the masks of the current key drawing with color gradients indicating their relative depth (lighter shades on top of darker shades, as seen in frames 1 and 2 of Figure 15). Users may then select one of these masks, and use the mask completion tool like a pencil to extend it. Every time a stroke is drawn with this

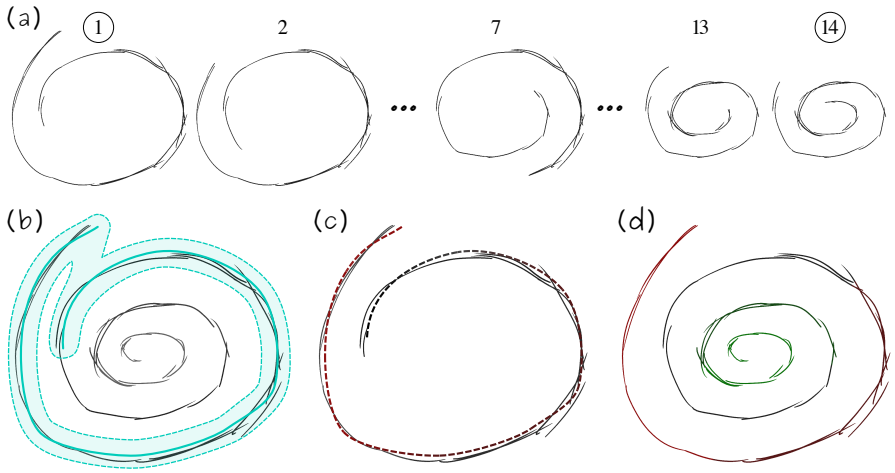
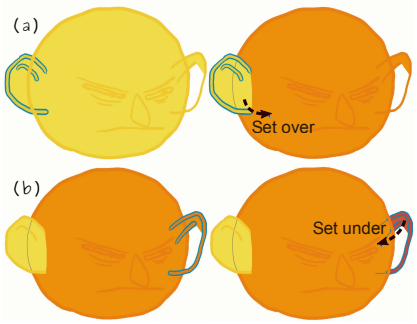


Figure 12: Temporal visibility. (a) Animation of a spiraling curve with appearing and disappearing parts. Disappearance is controlled by a temporal gradient tool: in (b), we show the brush footprint (in blue) of the tool; in (c), strokes in the footprint are projected on its guiding path (dashed) and are assigned corresponding negative temporal visibility thresholds (colored red); the result is shown in (d) on input strokes, along with the positive temporal visibility thresholds (colored green), obtained by direct painting.

tool the selected mask is updated. Note that we only add strokes that intersect the selected mask to avoid creating multiple connected components.

Layout tool. Using this tool, users may set the relative order of parts by first selecting one or more drawing parts, and then by left-clicking on a target drawing part. By default, this tool puts the selected parts on top of the target part (see inset (a)), but they may also be set at the same depth or under the target (see inset(b)) by holding the Shift or Ctrl key respectively. A preview of any layout change action is shown when hovering the mouse over a drawing part. Note that the layout change is set at the current frame in the timeline, which may be a keyframe or an inbetween. Layout changes are shown on both the spacing chart and trajectories with a yellow diamond that can be dragged to change its location in time.



Automatic layout actions. Propagating a layout to adjacent keyframes or suggesting layout changes (Section 5.1) are actions that may be triggered from the context menu opened when right-clicking on the canvas. These actions are instantaneous and may be activated at any time. Details about the effect of the action are displayed on screen (e.g., if a layout change was found, and at what time it was set), and users may cancel the result by pressing the Escape key.

Automatic stroke visibility action. It is used similarly to the automatic layout actions. After the action is triggered, visibility thresholds are displayed with colors (green and red for

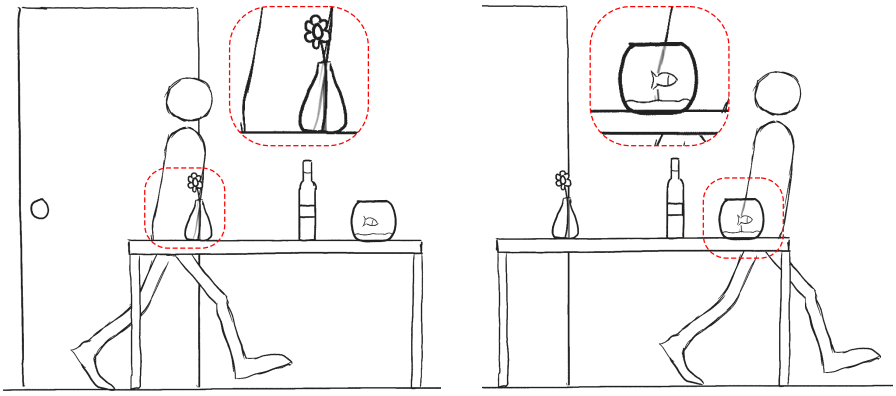


Figure 13: Parallax scrolling. A character (at depth level 1) walks in front of a door (at depth level 0) and behind objects laid on a table (at depth level 2). The masks of the semi-transparent vase, bottle and fish bowl are used to modulate the thickness and opacity of the character strokes (inset zooms). The drawings at the three depth levels are translated at a slightly different speed to create a parallax scrolling effect, as is best seen in the video.

appearing/disappearing stroke portions respectively). Users may cancel the result by pressing the Escape key.

Visibility thresholds tools. The temporal pen and eraser tools are merged with the standard pencil and eraser, simply changing their behavior when drawing on an inbetween frame (see the last paragraph of Section 5.2). In addition, the painting tool uses a brush-like metaphor to define a gradient of smoothly-varying visibility thresholds along a guiding path. By default, the assigned thresholds are positive, making strokes appear along the path. Conversely, holding Shift while drawing the path makes strokes disappear. The start and end frame numbers of the gradient are displayed. Users may adjust each of them at anytime by clicking on the number and moving the mouse to the left (resp. right) to decrease (resp. increase) the frame.

6.2 Case studies

In the following examples, we did not use the cross-fade feature of Even et al. system to smoothly blend sketchy keyframes, as we did not find it necessary for the design of rough 2D animations.

Parallax scrolling. In Figure 13, we show a simple and direct use of masks and layout to give the classic illusion of drawings in different planes moving at different depths, i.e., parallax scrolling. We reuse a walk cycle animation from Even et al. [1] (their Figure 16), and place it between background and foreground elements. Some of the drawing parts in the foreground are semi-transparent. Compared to the original animation with the character alone, the artist only needs to set up the layout once at keyframes and make the different drawing parts move at different speeds to obtain the new animation.

Travelling motion. A motion that conveys 3D-like cues is when the camera zooms in or out, also known as travelling motion. Here depth is conveyed by the gain or loss of details. This is

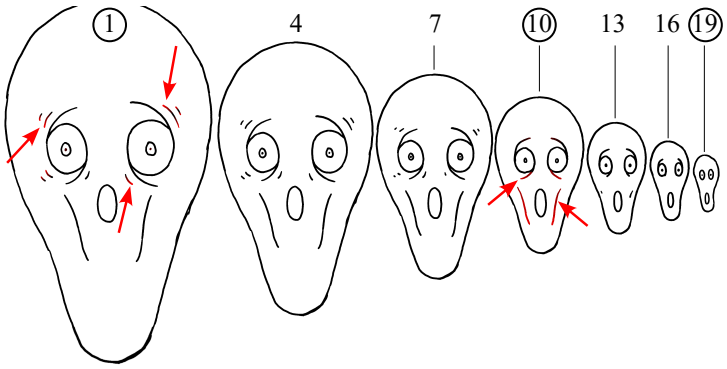


Figure 14: Travelling motion. Temporal visibility thresholds (shown in red at keyframes) are used to control stroke disappearance while zooming out.

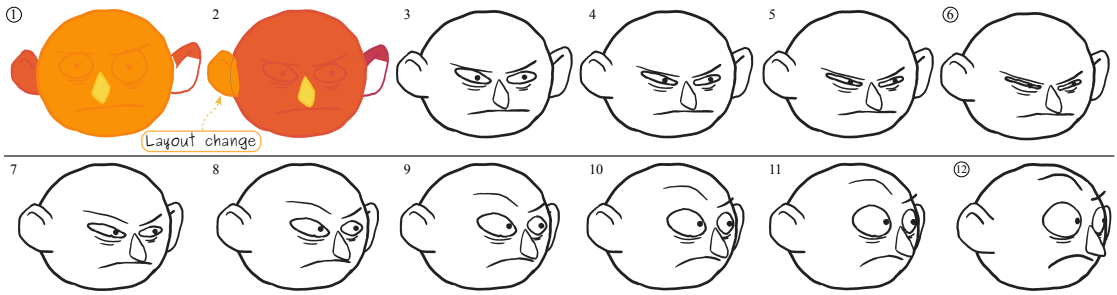


Figure 15: Out-of-plane rotation. The drawings parts along with their mask are depicted with colors on the first keyframe and the first inbetween. The depth level of the parts only changes at keyframes, except for the character's right ear that pops out in the foreground at frame 2 thanks to a dynamic layout change. The mask of the character's left ear does not need to be closed since it stays in the background.

trivially achieved with our temporal eraser or temporal pen tools, as shown in Figure 14 for the case of a zoom out motion, inspired by the animation "Love and Theft" by Studio Film Bilder[1].

Out-of-plane rotations. Among the most difficult 2D animations to produce by hand are the ones that convey the rotation of a 3D object or character around an axis different than the view direction, also called out-of-plane rotations. We show a first example in Figure 15, where we start from three key drawings traced from an animation made by artist Łukasz Rusinek[2]. The key drawings are decomposed into 13 parts: lines that lie in disconnected lattice elements are automatically grouped in different parts, some of which (e.g., the ears) are further separated. Masks are then automatically created and a layout assigned to the first keyframe. The mask of the left ear must be closed as it is moved to the front at frame 2 via a layout change. The drawing layout then remains the same until the end of the animation, and is thus easily propagated automatically.

Another example of out-of-plane rotation is shown in Figure 16, where this time we use the

[1] https://youtu.be/rEUxlwb2uFI?si=z2vDrunhITNdlEOR
[2] https://youtu.be/uEXZ9wICeWc?si=VdgLinA9pJuQbxHHk

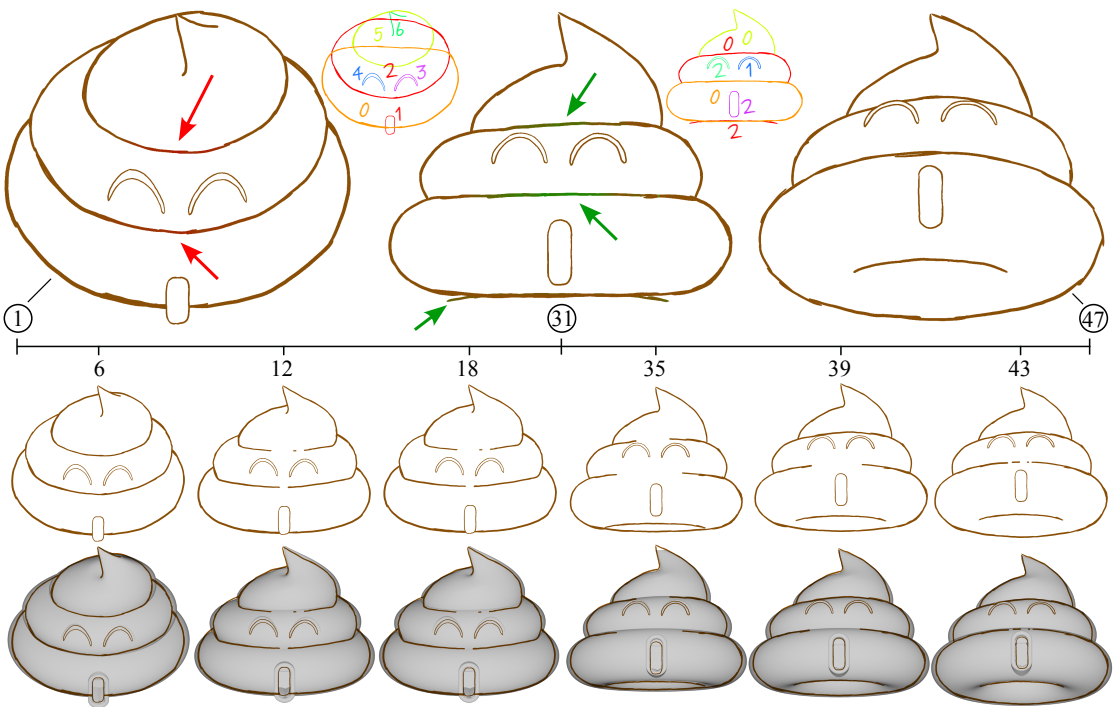


Figure 16: Out-of-plane rotation. Up and down tilting of an organic shape, rotoscoped on 3 frames of a 3D reference. Key drawings are decomposed into 7 parts whose layout only changes at keyframes (colored insets with depth levels). The temporal gradient tool was used to make stroke vertices progressively disappear (shown in red at keyframe 1) and appear (shown in green at keyframe 31), and their temporal visibility thresholds were then manually adjusted with the direct painting tool. The full line of the torus hole is made to appear at frame 32 using the temporal pen tool. In the bottom row, we show selected inbetweens generated by our system over a headlight rendering of the reference 3D animation.

rotation of a 3D cartoon character[3] as a reference. We first obtain keyframes by drawing over three frames of that animation (i.e., rotoscoping). These key drawings are decomposed into 7 parts to facilitate pose-to-pose matching. Their layouts differ in all three keyframes, as illustrated in the insets. After part matching, the animation is refined by controlling the trajectory of the top drawing part. Finally, we use stroke visibility to animate the appearance and disappearance of lines separating the three bottom parts of the object. Compared to a headlight rendering of the 3D animation (bottom row), even though inbetweens generated by our system follow rather faithfully the 3D motion of the object, our tools allow artistic expression in the way strokes appear and disappear, making them depart from their 3D counterpart.

Partial drawings. As explained in the book of Williams [36], 3D-like effects may be provided at specific frames through partial drawings. He gives several examples of walking characters where feet or hands suddenly appear to change orientation in 3D, revealing otherwise occluded parts. We reproduce such an example in Figure 17, using a walking animation toward the camera. The initial animation is produced as before: the different character parts are animated with the

[3] https://www.thingiverse.com/thing:3429760/files

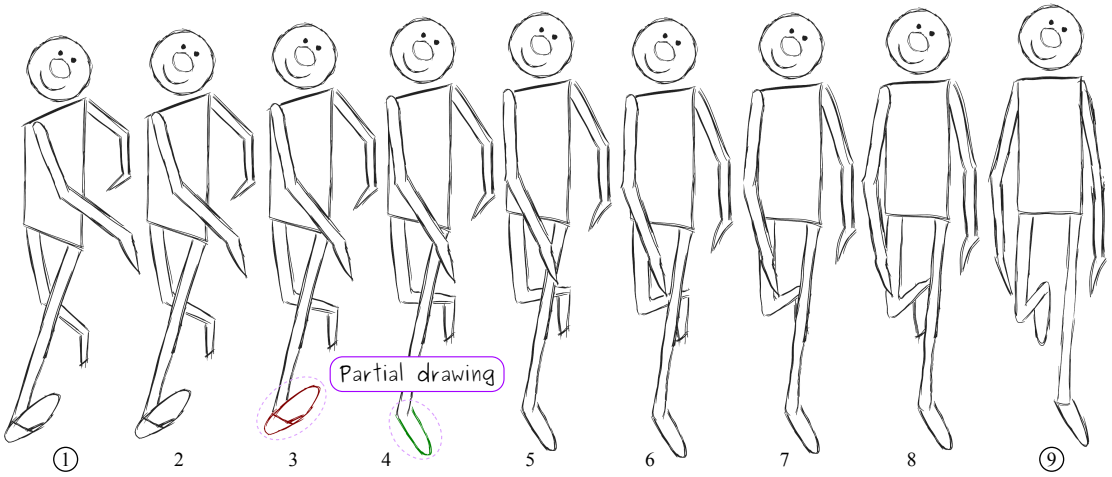


Figure 17: Partial drawings. Interval between the first and second keyframe of a walk cycle. A partial drawing is used to suddenly change the side of the right foot: the bottom (resp. top) side is made to disappear (resp. appear) at frame 4 using temporal visibility thresholds set to $-3/8$ (resp. $3/8$). Note that partial drawings are drawn at intermediate frames but stored at keyframes.

system of Even et al., masks are automatically created, and layout changes are set so that arms and legs pop in front or behind at the desired frame. We add a partial drawing in the style of Williams at frame 4 by using two different drawing parts for the feet with antagonist visibility thresholds: one where strokes are visible *before* that frame, and another where strokes appear *at* that frame. Hence both parts are never visible together at a same frame.

A similar example is shown in Figure 18 which also uses partial drawings at frame 4 to handle the self-occlusion of the folding fingers, along with visibility threshold gradients to achieve the gradual appearance of the folds in the fingers.

Deforming character. Depicting a deformable character in motion with hand-drawn animation requires training. A common exercise consists in animating a flour sack folding onto itself as it jumps around. We reproduce this animation with our approach in Figure 19 using three key drawings made of a single part. The fold of the first key drawing representing a self-occlusion disappears thanks to temporal visibility. We let the top part of the sack appear halfway between the second and third keyframes through a partial drawing. Thanks to its relative positioning in time, it remains correct after retiming of the animation.

6.3 Performance

Our animation system works in real-time on an Intel Core i7-4990K CPU, as detailed below for key routines of our implementation.

Mask creation. The time complexity of the mask creation algorithm scales linearly with the number of quads of the drawing part and is in the order of 1ms for a part with 2592 vertices and 905 quads on the tested computer. Masks are only recomputed when a new stroke is added. With a similar number of vertices, the *Alpha Contours* [33] algorithm takes more than 1 minute.

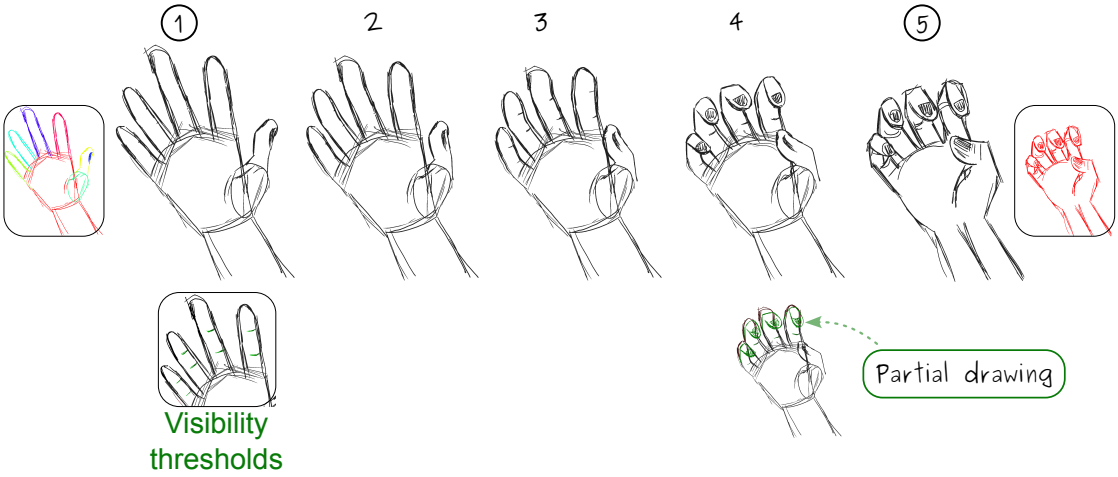


Figure 18: Fingers closing. Inbetween frames (2 to 4) are generated by our system from the drawings at keyframes 1 and 5. Their decomposition into embeddings is highlighted with colors in the insets. The inset under keyframe 1 shows the visibility thresholds used for the creases.

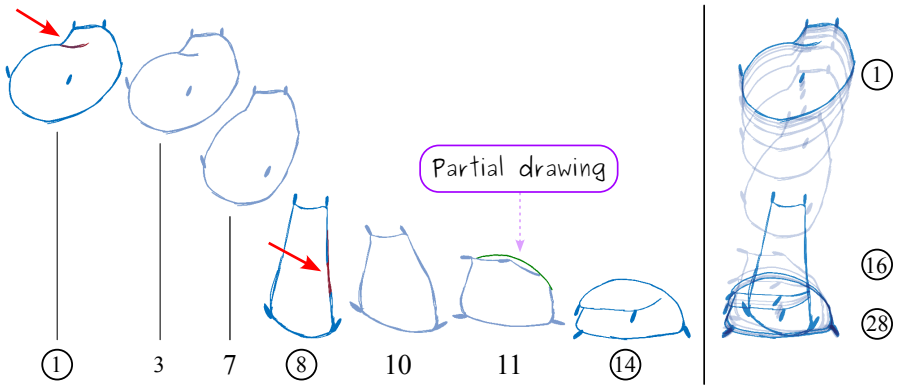


Figure 19: Deforming character. Classical flour sack animation, where we rely on stroke visibility tools to create 3D cues: strokes disappear (shown in red at keyframes 1 and 8) to convey moving cusps, while a partial drawing (in green) is used to make the back of the flour sack appear at frame 11. We show in the right column, the animation with onion skin visualization after retiming (doubling the number of frames).

Neighborhood queries. The automatic layout propagation, layout change and stroke visibility estimation algorithms all rely on neighborhood queries to evaluate window functions. We perform these queries efficiently by computing one k-d tree per keyframe using *nanoflann* [37]. We also precompute mask/vertices intersections to speedup the vis() function described in Section 5. Thanks to these implementation choices, we achieve the following performance: Layout propagation takes 3ms for Figure 15, made of 13 drawing parts, for a total of 3561 stroke vertices; Layout change detection takes 25ms on Figure 15 as well, where 5 inbetween frames are considered; Stroke visibility thresholds estimation takes 28ms on Figure 1, which contains 3095 stroke vertices.

7 Comparison with the system of Even et al.

The last part of the supplemental video shows a comparison of our approach to the previous system of Even et al. [1].

7.1 Non-linear editing with occlusions

The only way to convey occlusions among parts or self-occlusions in the original system of Even et al. is to convert (or "bake") each interpolated frame to a keyframe, and manually erase all hidden lines. This has several drawbacks, as shown on a couple use cases (one for each type of occlusion, based respectively on Figures 15 and 1) in the supplemental video: it is not only time consuming, but it also makes non-linear edition functionalities useless, as shown on a retiming example. In contrast, our semi-automatic tools retain such functionalities while requiring little user interaction. Most notably, when retiming the animation, our approach produces a proper interpolation without the need of further adjustments. This would be the same for other non-linear editing operations such as modifications of the spacing.

This set of comparisons was performed by one of the authors, who completed the two animation use cases 5.1 and 8.5 times faster respectively with the proposed tools. Such completion speedups are of course dependent on the number of inbetween frames that must be produced, with better speedups obtained when more inbetweens are generated.

7.2 Usability study

Protocol. We conducted a usability study with 5 participants who were not familiar with our animation system, one of them being an hobbyist 2D artist. Participants were given a brief introduction of our animation tools: they had a few minutes to familiarize themselves with the interface and the tools on a sandbox example. We then asked them to complete two animation use cases, first with our system, and then with the earlier system of Even et al. [1].

The first use case focused on evaluating layout tools. Participants were given the out-of-plane rotation animation of Figure 15 without any occlusion, but with drawing parts already segmented and matched across the three keyframes. They were first asked to resolve occlusions using our system, through the following actions in the order they wanted: close masks when necessary, order drawing parts, propagate layout and detect layout changes. They were then presented with the animation of Figure 15 without any occlusion once again, but this time without access to our animation tools. In order to convey occlusions in the system of Even et al., each frame was "baked" to a keyframe, and the participants were asked to erase all the lines that should be hidden due to occlusions.

The second use-case was specific to stroke visibility tools. Participants were given the out-of-plane rotation animation of Figure 1, where all occlusions between parts had been already handled, leaving only self-occlusions to be dealt with. They were first asked to resolve these occlusions with two of our animation tools: the gradient visibility tool, and the automatic suggestion of visibility thresholds. They were then presented with the input animation once again, but this time without access to our animation tools, and with all frames "baked" as keyframes as before, with the goal of erasing portions of lines to avoid popping artifacts.

Throughout these sessions, we have recorded the time between the first and last user input for a given session.

Results. The timings for the five participants (P1 to P5) are summarized in Table 2. All participants achieved their tasks faster using our system, even though they were not familiar

	Layout Ours	**Layout Even**	**Speedup**	**Viz Ours**	**Viz Even**	**Speedup**
P1	92s	227s	×2.4	95s	530s	×5.5
P2	152s	278s	×1.8	31s	181s	×5.8
P3	206s	411s	×2	70s	334s	×4.7
P4	142s	363s	×2.5	92s	388s	×4.2
P5	117s	388s	×3.3	57s	287s	×5
Avg	153.8s	333.4s	×2.1	69s	344s	×5

Table 2: Task completion time. We have recorded the time between the first and last user input for a given task. Participants were always faster with our tools, with greater speedups for the simpler visibility task.

with it. For the first use case (layout), they were on average 2.1 times faster; while for the second use case (visibility), they were 5 times faster on average. The hobbyist 2D artist (P1) showed speedups similar to other participants, but spent more time refining the visual results than other participants.

Such completion speedups are of course dependent on the number of inbetween frames that must be produced, with better speedups obtained when more inbetweens need to be produced. The two use cases of the study contained 13 and 17 inbetween frames respectively, which amounts to roughly half a second of animation per use case at 24 frames-per-second.

Overall, we received positive feedback about the system. All participants enjoyed using our animation tools. The hobbyist 2D artist told us he would be using it for his own projects, in particular thanks to the amount of automation found in the system (e.g., automatic mask computation) and the possibility to refine results by hand down to individual frames.

All criticisms concerned the user interface of our prototype animation system. For instance, we were suggested to provide a legend for the color used for displaying the drawing layout, to provide additional shortcuts to order masks more quickly, or to have a window explicitly listing the set of drawing parts.

8 Discussion

We have presented a 2D animation system offering automatic and semi-automatic tools whose purpose is to convey occlusion cues. It is by design more adapted to traditional 2D animation workflows than 2.5D methods, which are more suited to interactive environments or live performance retargeting. A central feature of our approach is the ability to control occlusions among drawing parts and self-occlusions *between* keyframes. We do so while retaining ordering, drawing and erasing interaction metaphors, which stays close to traditional 2D animation workflows, while providing fast and non-linear controls. In particular, timing & spacing or trajectories may be edited at anytime without breaking the animation, as shown in the supplemental video. We have demonstrated the benefits of our approach on a variety of example motions: parallax, out-of-plane rotations, zooms with LOD, deforming characters, etc.

Limitations. As shown in Figure 16, our method is able to create occlusions where parts of an object move in front of others, creating typical sliding T-junctions. However, this requires separating the drawing into smaller parts, which may occlude each other via their masks. In Figure 20 we show an example of this limitation. Imposing such a drawing decomposition may become tedious for artists, especially when animating very flexible elements such as cloth, re-

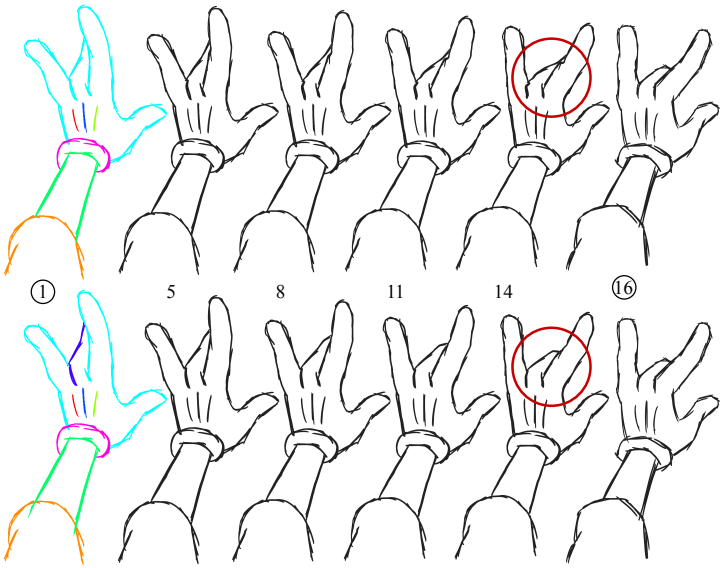


Figure 20: Sliding T-junctions. We traced over the glove animation from [24] in a sketchy style and segmented the drawing in 7 parts. The sliding T-junctions of the fingers cannot be accurately represented without separating them into different parts (second row). We use trajectory constraints to guide the junctions as shown in the supplemental video.

quiring many parts. One solution would be to implement local layering [38] in our animation structure, which would also help deal with occlusion cycles.

Our animation tools have limitations of their own. Automatic layout propagation may not find the best many-to-one mapping between drawing parts in some tricky cases. However, we have found it to provide adequate suggestions in practice in all our experiments. The automatic detection of visibility thresholds relies on an isotropic diffusion, which does not capture complex stroke (dis)appearance as shown with the spiral example in the supplemental video. A more sophisticated diffusion might correct that issue, even though the visibility gradient tool already provides a practical alternative. Lastly, dynamic layouts and stroke visibility are controlled independently. This may raise some issues in practice when a T-junction turns into a cusp. Coupling tools that control the two types of occlusions might thus constitute a useful extension.

Future work. In our system, perspective effects must still be handled by the artist. Hence they are "baked" in key drawings or partial drawings. An interesting direction of future work would be to provide non-linear controls over perspective distortions, using 2D widgets such as editable vanishing lines.

Another aspect that remains in the hands of artists is the depiction of volumes. This could be incorporated in our system by taking inspiration from Disney's solid animation technique, using drawings of simple 3D volumes (e.g., ellipsoids) that are used as proxies to guide the animation while remaining hidden. Finding semi-automatic techniques that achieve such a high-level control is a challenging avenue of future work.

Finally, our system does not seem to be adapted to complex special effects animation, such as growing elements, water splashes, or sparks and explosions. Assisting artists in producing such effects might require an altogether different solution due to the amount of changes in drawings across frames.

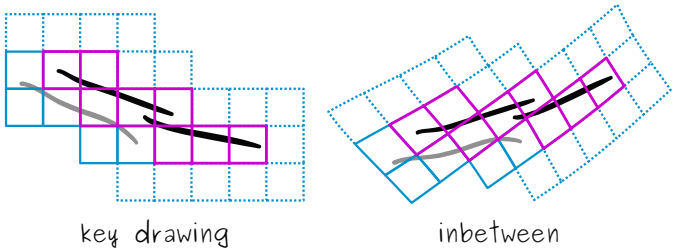


Figure 21: Lattice expansion. Right: A new stroke (in gray) is drawn at an intermediate frame partially outside the deformed lattice (in magenta). Left: The original lattice of the previous key drawing is enlarged by one ring of quads (in cyan). Each of these quads is deformed by the affine transformation of one of its neighbor, and disconnected quad corners are merged by averaging. The new stroke is eventually embedded in this expanded grid, and empty quads (dotted lines) are removed.

A Lattice expansion

Our automatic lattice expansion routine works as follows (see Figure 21). The axis-aligned (undeformed) lattice stored at the previous keyframe is expanded by a ring of quads. Each new quad is then deformed through the affine transformation of its neighboring quad in the original lattice and relaxed using ARAP regularization, after which disconnected quad corners are merged by averaging, similarly to Sykorà et al. [31]. A stroke fits into the lattice if adjacent stroke vertices can be embedded in the same or adjacent quads of the lattice. If all the new strokes fit in the expanded lattice, the process stops and empty quads are removed. Otherwise, we iterate until this termination criterion is met.

B Animation structures overview

Please see Figure 22.

References

[1] M. Even, P. Bénard, P. Barla, Non-linear rough 2d animation using transient embeddings, *Computer Graphics Forum* 42 (2) (2023) 411–425. doi:10.1111/cgf.14771.

[2] O. Johnston, F. Thomas, The illusion of life : Disney animation, 1995.

[3] J. J. Gibson, The ecological approach to the visual perception of pictures, *Leonardo* 11 (3) (1978) 227–235.

[4] S. A. Engel, D. A. Remus, R. Sainath, Motion from occlusion, *Journal of Vision* 6 (5) (2006) 9–9.

[5] N. Burtnyk, M. Wein, Computer animation of free form images, *in: Proc. of the 2nd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, 1975, p. 78–80. doi:10.1145/563732.563743.

[6] TVPaint Developpement, Tvpaint (2023).
URL https://www.tvpaint.com

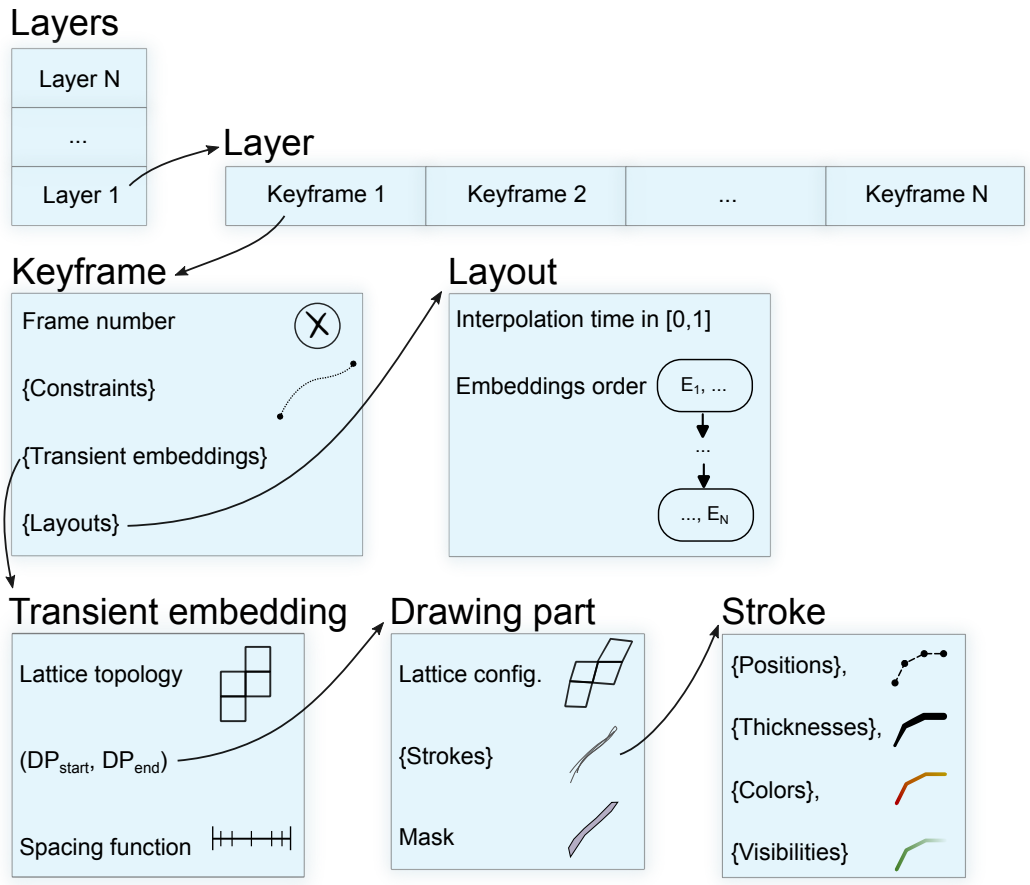


Figure 22: Animation structures. A layer stack determines the order in which drawings are composited. Each layer consists of a list of keyframes located at discrete frames in the animation, which define its *timing*. Keyframes store information about a drawing through transient embeddings in addition to how it should be interpolated towards the next keyframe using trajectory constraints and each embedding's spacing function. Transient embeddings control the motion of stroke groups based on a 2D square lattice structure and two *drawings parts* (DP). Each of them identifies a set of strokes, one aligned in time with the current keyframe ($\mathrm{DP}_{\mathrm{start}}$) and the other aligned with the following keyframe ($\mathrm{DP}_{\mathrm{end}}$). Together, these two drawing parts specify a rough matching between keyframes, marking the start and end of the interpolation. Each keyframe also contains at least one layout, specifying the arrangement of transient embeddings and allowing occlusions between parts. Layout changes can occur at any time during the interpolation.

[7] Celsys, Inc., Clip studio paint (2024).
URL https://www.clipstudio.net/

[8] S. C. Hsu, I. H. H. Lee, N. E. Wiseman, Skeletal strokes, *in: Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, UIST '93, ACM, 1993, p. 197–206. doi:10.1145/168642.168662.

[9] S. C. Hsu, I. H. H. Lee, Drawing and animation using skeletal strokes, *in: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, ACM, 1994, p. 109–118. doi:10.1145/192161.192186.

[10] C.-K. Yeh, P. Song, P.-Y. Lin, C.-W. Fu, C.-H. Lin, T.-Y. Lee, Double-sided 2.5d graphics, *IEEE Transactions on Visualization and Computer Graphics* 19 (2) (2013) 225–235. doi:10.1109/TVCG.2012.116.

[11] A. Rivers, T. Igarashi, F. Durand, 2.5d cartoon models, *ACM Trans. Graph.* 29 (4) (2010). doi:10.1145/1778765.1778796.

[12] J. Coutinho, B. A. D. Marques, J. P. Gois, Puppeteering 2.5d models, *in: 2016 29th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, 2016, pp. 1–8. doi:10.1109/SIBGRAPI.2016.010.

[13] T. Fukusato, A. Maejima, View-dependent deformation for 2.5-d cartoon models, *IEEE Computer Graphics and Applications* 42 (5) (2022) 66–75. doi:10.1109/MCG.2022.3174202.

[14] P. Rademacher, View-dependent geometry, *in: Proc. of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, ACM, 1999, p. 439–446. doi:10.1145/311535.311612.

[15] M. Alexa, D. Cohen-Or, D. Levin, As-rigid-as-possible shape interpolation, *in: Proc. of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, 2000, p. 157–164. doi:10.1145/344779.344859.

[16] F. Di Fiore, P. Schaeken, K. Elens, F. Van Reeth, Automatic in-betweening in computer assisted animation by exploiting 2.5d modelling techniques, *in: Proc. Computer Animation 2001*, 2001, pp. 192–200. doi:10.1109/CA.2001.982393.

[17] I. E, N. S. Willett, A. Finkelstein, 2.5d simulated keyframe animation in blender, UIST '21 Adjunct, ACM, 2021, p. 35–36. doi:10.1145/3474349.3480222.

[18] B. O. Community, Blender - a 3d modelling and rendering package (2023).

[19] Toon Boom Animation Inc., Toon boom harmony (2023).
URL https://cacani.sg/

[20] Reallusion Inc., Reallusion 360 head creator (2023).
URL https://www.reallusion.com/cartoon-animator/360-head-creator.html

[21] Live2D Inc., Live2d cubism (2023).
URL https://www.live2d.com/

[22] J. Jiang, H. S. Seah, H. Z. Liew, Stroke-based drawing and inbetweening with boundary strokes, *Computer Graphics Forum* 41 (1) (2022) 257–269. doi:10.1111/cgf.14433.

[23] L. Carvalho, R. Marroquim, E. Vital Brazil, Dilight: Digital light table – inbetweening for 2d animations using guidelines, *Computers & Graphics* 65 (2017) 31–44. doi:https://doi.org/10.1016/j.cag.2017.04.001.

[24] B. Whited, G. Noris, M. Simmons, R. W. Sumner, M. Gross, J. Rossignac, Betweenit: An interactive tool for tight inbetweening, *Computer Graphics Forum* 29 (2) (2010) 605–614. doi:10.1111/j.1467-8659.2009.01630.x.

[25] W. Yang, Context-aware computer aided inbetweening, *IEEE Transactions on Visualization and Computer Graphics* 24 (2) (2018) 1049–1062. doi:10.1109/TVCG.2017.2657511.

[26] B. Dalstein, R. Ronfard, M. van de Panne, Vector graphics animation with time-varying topology, *ACM Trans. Graph.* 34 (4) (2015). doi:10.1145/2766913.

[27] L. Zhang, H. Huang, H. Fu, Excol: An extract-and-complete layering approach to cartoon animation reusing, *IEEE Transactions on Visualization and Computer Graphics* 18 (7) (2012) 1156–1169. doi:10.1109/TVCG.2011.111.

[28] H. Zhu, X. Liu, T.-T. Wong, P.-A. Heng, Globally optimal toon tracking, *ACM Trans. Graph.* 35 (4) (2016). doi:10.1145/2897824.2925872.

[29] J. Xing, L.-Y. Wei, T. Shiratori, K. Yatani, Autocomplete hand-drawn animations, *ACM Trans. Graph.* 34 (6) (2015). doi:10.1145/2816795.2818079.

[30] M. Even, P. Bénard, P. Barla, Frite, [software] (Sep. 2023).
URL https://inria.hal.science/hal-04202866

[31] D. Sýkora, J. Dingliana, S. Collins, As-rigid-as-possible image registration for hand-drawn cartoon animations, *in: Proc. of the 7th International Symposium on Non-Photorealistic Animation and Rendering*, ACM, 2009, p. 25–33. doi:10.1145/1572614.1572619.

[32] W. Baxter, P. Barla, K.-I. Anjyo, Rigid shape interpolation using normal equations, *in: Proc. of the International Symposium on Non-photorealistic Animation and Rendering*, ACM, 2008, pp. 59–64. doi:10.1145/1377980.1377993.

[33] M. Myronova, W. Neveu, M. Bessmeltsev, Differential operators on sketches via alpha contours, *ACM Trans. Graph.* 42 (4) (2023). doi:10.1145/3592420.

[34] D. Shreiner, OpenGL programming guide: the official guide to learning OpenGL, 7th Edition, Pearson Education, 2009, Ch. 11.

[35] D. Sykora, D. Sedlacek, S. Jinchao, J. Dingliana, S. Collins, Adding Depth to Cartoons Using Sparse Depth (In)equalities, *Computer Graphics Forum* (2010). doi:10.1111/j.1467-8659.2009.01631.x.

[36] R. Williams, The animator's survival kit, Faber and Faber, 2001.

[37] J. L. Blanco, P. K. Rai, nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees, https://github.com/jlblancoc/nanoflann (2014).

[38] J. McCann, N. Pollard, Local layering, *in: ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, Association for Computing Machinery, New York, NY, USA, 2009. doi:10.1145/1576246.1531390.

RESEARCH CENTRE
Centre Inria de l'Université de Bordeaux

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399