



Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option GVR

— 2.5D Rotoscoping — A Space-Time Topological Approach

Boris Dalstein

June 21st, 2012

Research project performed at IMAGINE
(joint team between Laboratoire Jean-Kuntzmann and Inria)

Under the supervision of:
Dr. Rémi Ronfard, Inria

Consultants:
Prof. Marie-Paule Cani, Inria
Dr. Alla Sheffer, UBC

Evaluated by the Thesis Committee:
Dr. Dominique Attali
Prof. Nadia Brauner
Prof. James Crowley
Dr. Rémi Ronfard

Abstract

Rotoscoping is the task of reproducing an input video of a 3D scene with a sequence of 2D drawings. It is a common task for creating special visual effects (the drawings are used as masks). It is also a commonly used technique for professional and amateur animators, who use video footage or even recordings of themselves to capture subtle expressions in gestures and movements.

Rotoscoping is a difficult and tedious task, requiring a large number of keyframes to be drawn. One technique that could improve the efficiency of rotoscoping is automatic inbetweening between keyframes. But this has proven to be a very challenging task. The state-of-the-art in automatic inbetweening is limited to keyframes with very little or no change in the topology of the drawings.

In this thesis, we explore techniques for inbetweening keyframes with different topological structures by building an explicit representation of the space-time topology of the animation. Contrary to recent approaches, which tackle the problem by attempting to reconstruct the depth of the 3D scene ($2.5D = 2D + \text{depth}$), we instead attempt to reconstruct the history of topological events ($2.5D = 2D + \text{history}$).

The report is organized as follows: First, we review the literature on inbetweening. Then, we present the theory of our representation for space-time topology, as well as some details of its implementation. Then, we present an algorithm using this structure to compute automatically clean vectorial inbetweens from two drawings with inconsistent topologies. Finally, we present some results and discuss future work to improve them.

| | |
|--|-----------|
| Abstract | i |
| 1 Introduction | 1 |
| 2 Related Work | 5 |
| 2.1 Early Approaches | 5 |
| 2.2 Shape Morphing | 7 |
| 2.3 Vision-Based Approaches | 8 |
| 2.4 Using Stroke Graphs | 10 |
| 2.5 Restricting the Class of Animation | 13 |
| 3 Space-Time Topology | 15 |
| 3.1 Motivations | 15 |
| 3.2 Stroke Graphs As Input | 17 |
| 3.3 Animated Stroke Graph As Output | 19 |
| 3.4 Atomic Events | 24 |
| 4 Automatic Inbetweening | 27 |
| 4.1 Initialisation of the algorithm | 28 |
| 4.2 Operators: One Step of the Algorithm | 30 |
| 4.3 Deformation Energy | 32 |
| 4.4 Exploring the Search Space | 34 |
| 5 Experimental Results | 37 |
| 5.1 Implementation and Interface | 37 |
| 5.2 Results and Validation | 37 |
| 5.3 Limitations and Future Work | 43 |
| 6 Conclusion | 47 |
| Bibliography | 49 |

1 Introduction

In the past few years, there have been tremendous changes in the way people communicate. “Computers” (any computing device) are more and more affordable, such that almost every family has one, and even almost every student of the so called developed countries has a “laptop”, thanks to very cheap notebooks, and the recent interest for tablets or smartphones. All of these being more and more commonly fulltime connected through the Internet, virtually every person is connected permanently to any other person.

This revolution was such that a whole part of the Internet has been reorganized around social networks, mostly centered around *sharing*: sharing life information, sharing thoughts, sharing content. This content is for instance images, videos or music. This is a broad new heaven for expressiveness, but unfortunately there is very few tools for – creating – this content: it is mostly created by professionals, and Internet users only share it with their “friends”.

But we believe most people seek for creativity –eg not just sharing–, and they only lack simple tools for that, especially in the field of animation. To fill this gap, we would like to build an intuitive framework, for non skilled drawers or animators, to create their own animations. The simple idea we had to design such a tool is that creating content from scratch needs skills by essence, while modifying existing content is much more simpler. Then, it would be interesting to provide them proportions, timing and inspiration by using existing videos: this is the field of rotoscoping. This technique consists in creating an animated movie by drawing on top of a reference video, and has been first used and patented [30] by Fleischer Max in 1917. It has been notably popularized by Walt Disney by using it during the production of *Snow White and the Seven Dwarfs* in 1937. A famous reference video for rotoscoping is “The Horse in Motion” photographed by Eadweard Muybridge in 1878, see Figure 1.1. Animators can redraw on top of each frame to get an animation with the right timing and poses, that will ensure an appealing motion, but possibly with a completely new style, or even another animal.

However, even if it helps a lot compared to starting from a completely blank page, this still requires a lot of time, since every frame needs to be drawn individually, with good time consistency. Then, this is not enough for the purpose we seek. It would be better if the user only draw a few representative stylized frames on top of the video (eventually helped by some guidelines once he has drawn the first frame) and let the system compute all the frames inbetween.

Then, to design such an intuitive interface, we do need to tackle the difficult problem of automatic 2D inbetweening, eg given two frames of an animation at different times, compute all the intermediate frames. A lot of research on this field has been done from the 1970s, but despite recent advances, no general satisfactory solution has been found. Either there is a lot of restriction in the class of animation that can be drawn (and are not robust to unaccuracy of the drawer), or it requires too much manual intervention for the application we target.

What makes automatic inbetweening in its more general formulation really difficult

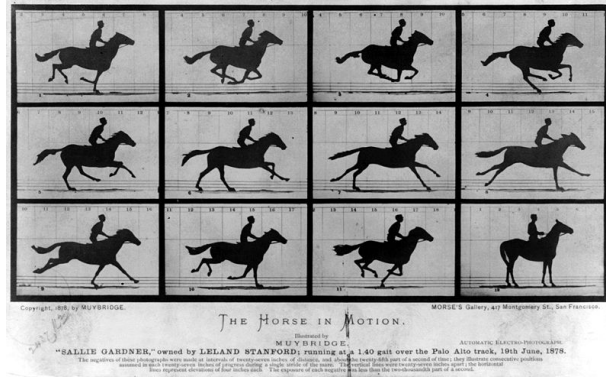


Figure 1.1: *The Horse in Motion*, photographed by Eadweard Muybridge in 1878

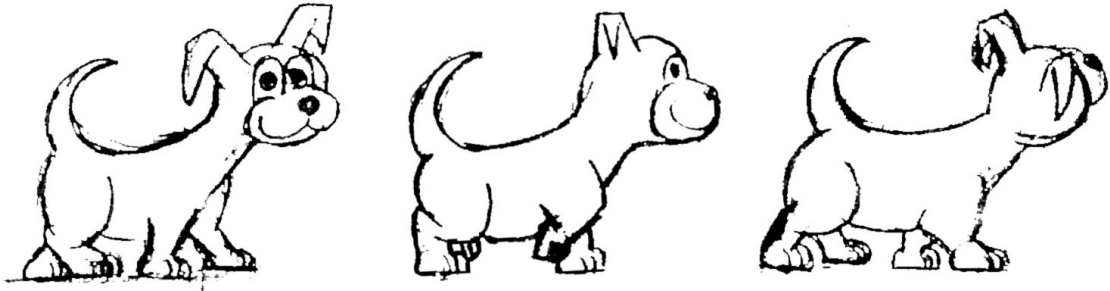


Figure 1.2: Three frames of a hand drawn animation, with huge topological inconsistencies

is the *topological* differences between the drawings, as we can see in Figure 1.2. Only a partial matching between strokes is possible (for instance the left ear is completely hidden in the second frame), but even when a matching exists, their topological incidence can change (for instance the junction between the tail and the back leg: in the first frame they join at the end of the strokes, in the others the stroke of the tail ends in the middle of the stroke of the leg). If we look at the head, only a very small subset of strokes could actually be matched all along the three frames. Then, any algorithm relying on a one-to-one correspondence would fail, and even assuming a many-to-many algorithm gives a perfect answer for strokes actually in correspondence, it doesn't tell us much on what to do with the remaining strokes, and what to do with strokes matched to several ones.

In such a difficult case, being able to compute the right inbetweening is probably impossible without a human interpretation of what the drawing actually represents, and then thanks to our knowledge of its 3D shape, draw the intermediate frames. In an algorithmic point of view, this means we need to reconstruct the 3D shape from these few stylised representations of our object, which is a vision problem probably more complex than our initial problem.

Due to this difficulty, so far, no existing algorithm can automatically generate inbetweens with topological inconsistencies, or only in very simple cases. In fact, very few research has been done in this direction, directly focused on these inconsistencies. Unfortunately, for our needs, we must handle them for three reasons: Firstly we want to use reference videos and they naturally occur (a professional animator would design the animation such that it doesn't occur unless really necessary). Secondly the targeted beg-

giner user will make “mistakes” that a professional animator would not do, among them inaccurate drawings or not properly closed faces. Lastly the targeted user do not want to learn a software, and then no “complicated” manipulation should be used to handle those topological inconsistencies (a professional animator would carefully create several layers and preform more complicated tricks).

In our case, the problem is still a bit simpler than the most general formulation: on the one hand, we can analyze the reference video to help us finding the right correspondence; and on the other hand we do not require an accurate inbetweening, since our targeted user will probably be fine with only a “not too bad” result.

The initial goal of this research was to design such an intuitive interface, where the user draw several frames on top of a reference video, with possibly topological inconsistencies between two drawings, due to the underlying 3D world (or unaccuracy of the drawer). Because the topology changes, deforming a 2D shape isn’t enough for that task: some lines should disappear, others should appear, some should split and merge together. This is what we call 2.5D rotoscoping.

However, we lacked a structure to describe such a 2.5D animation, where animated lines interact with each other, by splitting or merging. Defining this structure, how to manipulate it, how to perform computation on it, how to create inbetweens with it, and implement it appeared to be more complex than expected, and finally the research was mostly focused on this part. Using the video as a mine of information to help the inbetweening algorithm has been postponed to future works.

In this report, we first review the litterature on inbetweening, where the Section 2.4 defines what a stroke graph is, concept used all along our research. Then, we present the theory about a new spatio-temporal representation of 2D animation, as well as the data structure designed to implement this concept. Then, using this structure, we propose an algorithm to compute automatically a clean vectorial inbetweening from two drawings, in the cases of topological inconsistencies. Starting from some initial trustful stroke correspondences computed automatically, the algorithm iteratively builds the whole animation by attempting to minimize an energy defined over our spatio-temporal representation. Finally we present the different results of our approach, as well as what are its limitations, and discuss future work to tackle these limitations.

2 Related Work

The most important part of an intuitive rotoscoping system is to prevent the user to draw all the frames of a video (at 24fps for instance), since it requires so much time and accuracy to be time consistent. The problem of inbetweening occurs when doing 2D animation, the Walt Disney company being one of the first having to deal with it back in the 1920s. The best reference to get in touch with traditional animation and the problems of inbetweening is the very well-known “Illusion of Life” [48], written by two of the so-called “Nine Old Men”, heritage of Disney golden age and traditional techniques. It is also quite known in the Computer Science community, most probably because cited by the Lasseter’s 1987 article [25], presenting how the principles described in the book can be applied to 3D. Of course, some others good books are also available, for instance a lot of nice examples could be found in [8], written by Preston Blair who also worked at Disney. The book [52] also provides a mine of examples of 2D animations, the author Richard Williams was the animation director of “Who Framed Roger Rabbit”, an animated movie bringing back a Warner-like style, instead of the Disney style of that time.

Making these inbetweens manually being tedious and time-consuming, automatic ways to do it has been looked for. In 1978, Catmull [13] provides one of the very best introduction to automatic inbetweening. At that time, some attempts have been done to introduce the computer in the animation framework to help artists wherever possible, saving time and money. For instance, Marc Levoy presents in 1977 [26] one of the first digital animation systems, being developped at Cornell University. Catmull enumerates the several possible approaches to tackle the problem of inbetweening, here are his exact words back in 1978:

1. Try to infer the missing information from the line drawings.
2. Require the animators or program operators to specify the missing information by editing.
3. Break the characters into overlays.
4. Use skeletal drawings.
5. Use 3D outlines and centerlines.
6. Restrict the class of animation that may be drawn.

A detailed description of these is provided in [13]. Thirty years later, even if a lot of progress has been done, the problem in its more general formulation can still be considered unsolved, and it is striking to see how this enumeration is still very relevant to classify the different papers.

2.1 Early Approaches

As noticed in [51], many of the early approaches are stroke-based, it means they use a “vectorial” representation of the strokes. To clarify the ideas, I mean by “vectorial” more

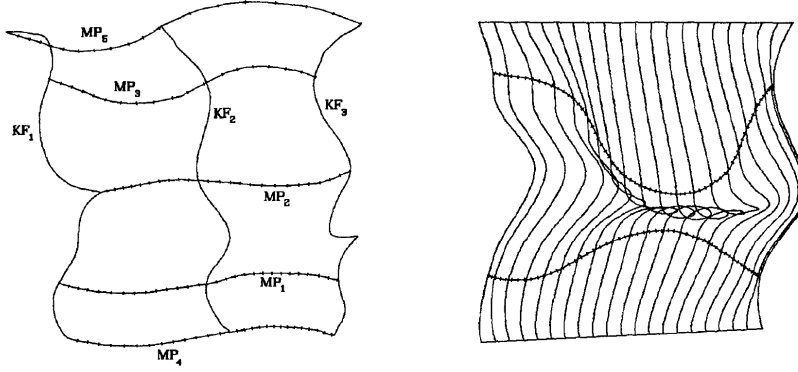


Figure 2.1: Left: a patch network (KF represent the drawn keyframes, MP the moving point constraints). Right: example of inbetweens generated, showing in this case contour issues (image from Reeves 1981 [34])

or less any representation of curve which is not a pixelized image: it could simply be a polyline with a dense list of 2D points (eg $p_{i+1} - p_i$ is approximatively the size of a pixel), or any explicit parameterized curves like Bézier curves. I tend not to consider as vectorial the curves represented by implicit functions, because if we do so, pixelized images can actually be seen as a discrete sampling of an implicit function defining our curves.

One of the reason why early approaches are stroke-based is that storing complete images in memory was very expensive at that time, and they did not have the computational power of current computers necessary to process these images (for instance, [13] explains that full pictures were stored for the background paintings of their animation system at New York Institute of Technology, and that getting these pictures from the disk was “longer than desirable”). At the contrary, vectorial representations of strokes are much more sparse, even represented as a dense list of points, and then do not need as much computational power. But I also believe the reason of this approach is that representing strokes this way seems very natural, and is probably what anyone would think of at first, without being biased by existing recent research.

Forty-five years ago, in 1967, [31] was a first attempt at automatic inbetweening. A deformed curve was generated by tweaking continuously its parameters via an electronic circuit, and display the resulting curve using an analog cathode ray tube. These parameters being still computed digitally, both analog and digital computers were used, and then this method was presented as “hybrid curve generation”. About ten years later, considering skeletons to tackle the problem of inbetweening by deforming strokes as been suggested by Burtnyk and Wein in [12].

in 1981, Reeves in [34] proposed an approach of inbetweening using “moving point constraints”. The correspondences between strokes is given by the artist, as well as the complete animation path for some of them. The paths of the others is deduced by using space-time Coons surfaces (see Figure 2.1). This idea became commonly used in almost all computer animation systems, since it is a convenient solution, easy to implement, and that it provides to the artist almost all the freedom he wants. For instance, a similar approach can be used in the implementation of [51] if the artist is not pleased with the automatic result.

Little by little, computers have replaced traditional media in almost all steps of ani-

mation production. In 1995, Fekete [16] presents a very good overview of their animation system TicTacToon, probably similar to all current animation systems. A lot of aspects need to be taken into account to build an efficient animation system, that computer scientists tend to ignore. For instance, the importance of the exposure sheet, as stated in [13], or how to handle sound syncing. The system should be convenient enough to handle huge data management and social aspects relative to the cooperation and communication of hundreds of people with different skills.

But as far as inbetweening is concerned, at that time not a lot of advances had been done. For instance, it should be noted that so far, all methods required the artist to specify a one-to-one stroke correspondence, and then that no topological changes were handled. Then, despite the many advantages that automatic inbetweening provides, [16] claims “We have found that inputting and tuning simple in-betweens take about the same time as drawing them by hand”.

Nevertheless, more recent works could change that statement. Indeed, not only advances using stroke-based methods had been done, but also a bunch of new approaches has emerged, notably due to the progress of Shape Morphing, Image Registration, and Computer Vision.

2.2 Shape Morphing

A lot of paper dealing with “inbetweening” are in reality only dealing with the shape morphing problem (also called shape blending). This problem is, given two closed polygons, how to transform the first one into the other one. Then, it only consists in smoothly blend the overall silhouette of the shape, it doesn’t address any topological change that can happen *inside* the silhouette, and what is commonly done is texture blending.

Still, shape morphing is an interesting approach, where two problems needs to be solved , exactly the same way as in inbetweening. One is the vertex correspondence problem (which vertex corresponds to which), the other is the vertex path problem (how to move from the first position to the second position).

If the vertex correspondence is given, a naive approach consists in a linear interpolation of the positions of the vertices, but of course, it leads to unpleasing deformations or even self-intersection issues. In 1993, Sederberg [39] proposed an slightly better approach, using an intrinsic definition of the polygon (each vertex of the polygon is defined locally to the previous edge). It does not address the problem of vertex correspondence.

A solution to the vertex correspondence problem was proposed by Sebastian in 2003 [38], with a method to optimally align two curves (open or closed) based on length and curvatures. Note that this gives also a similarity metric between the two curves, that can actually be used to match several strokes (Remember that for the shape blending problem, there is only one stroke which is the silhouette).

But shape blending has mostly become popular thanks to the now very popular method called As-Rigid-As-Possible, proposed by Alexa et al. in 2000 [2], based on compatible triangulations that we interpolate minimizing their local deformation. This way, the internal structure of the shape is taken into account, and provides very natural blending. Note that it is only meant to solve the vertex path problem, it uses a simple user-guided approach for the vertex correspondence. It has lead to numerous improvements: in 2005, [19] adds rotation coherence constraints; in 2009, Baxter et al. [5] improved both

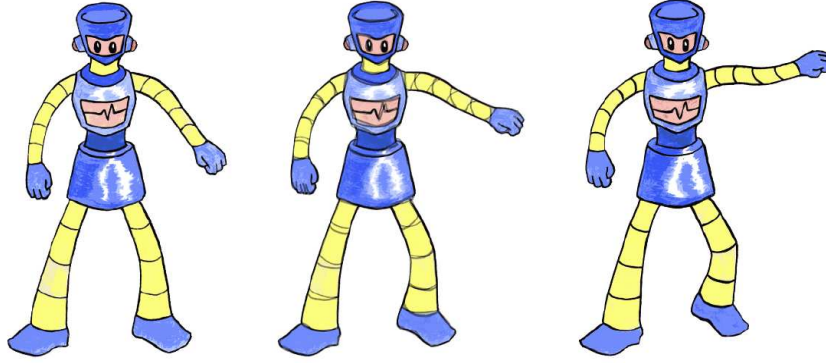


Figure 2.2: Blurring artefacts on the interior of the robot caused by any shape morphing method. Left: first image, Right: second image, Middle: inbetween image generated by shape morphing (image from Baxter et al. 2009 [5])

the vertex correspondence (however, a first pair of correspondence points still need to be provided by the user) and the compatible triangulation, based on a simplified shape. The same year Baxter et al. extended it to a N-way morphing [6], to interpolate between more than two shapes; and finally the automatic image registration approach proposed by Sykora et al. in 2009 [47] seems to be really promising. It has been reused for instance for TexToons [46] that won the best paper award of the NPAR conference in 2011. It solves pretty nicely the vertex correspondence problem in the case of shape morphing, and does not need any manual correspondence, even if real-time dragging can be achieved if necessary.

However, whatever good the shape morphing is, it cannot handle by essence the topological changes, and cannot be used for precise interpolation of details, notably in the interior of the shape. Indeed, what’s happening inside of the morphed polygon is only a texture blending between the initial textures, and then if lines are initially not at the same position, instead of “moving”, the first one will disappear while the other one will appear, causing an unpleasing blurring effect as seen in Figure 2.2. To deal with this problem, Baxter and al. in 2009 [6] uses a sharper and then less noticeable blending, and Fu and al. in 2005 [19] uses a streamline field over the spatial-temporal volume described by the morphing. However, these are only tricks that reduce slightly the visual impact, that do not at all solve the inherent problem.

Not exactly similar to shape morphing, but closed, there are all the methods to create a new shape by transforming/manipulating an existing one. For instance, As-Killing-As-Possible in 2011 [41], or the Igarashi method of 2005 using as-rigid-as-possible shape manipulation [22]

2.3 Vision-Based Approaches

Several techniques can be grouped into this category, even if they are very different. Their common point is that they use techniques mostly used by the Computer Vision community, and works in the image space.

A first class of approaches are the one that aimed at separating motion from content,

and then apply the motion to your specific case. For instance, Bregler et al. in 2002 [10] “turns to the masters” by reusing the motion of traditional animation to retarget it to any other media. The main idea is to best approximate this motion, for each shape, using an affine transformation and a weighted sum of key-shapes. This data is retrieved with different algorithms depending on the input form of the animation (cartoon contours point, raw segmented video, etc...) with generally a manual intervention depending on the animation, to segment correctly the input data (manual rotoscoping if necessary). The as rigid as possible approach is used to generate automatically plenty of other input key shapes, and then a Principal Component Analysis is used to select the best one and reduce the animation space. Then, they use this data to retarget the input media to an output animation, in different ways depending on the expected output. (3D shapes, 2D shapes, etc.). This is an interesting approach, but it only helps to get the right timing of you animation. It does not handle topological changes, and it takes time to model the “output key shapes”..

An other similar paper is from De Juan and Bodenheimer in 2006 [15], which attempts to reuse already existing 2D animation, but not only the motion: the whole animation of the character. For that, it presents a method of segmenting from videos to store the characters with their contour, and a method for inbetweening. The inbetweening part works this way:

- the user manually separate the character into layers, they will be treated independently
- for each keyframe of each layer, the contour is extracted: starts at a pixel on the edge of the silhouette, and trace the contour in clockwise order
- all these points are placed in a 3D embedding, the two keyframes have different z-values
- a 3D surface (an implicit surface generated with the points above and RBF interpolations) is extracted with the marching cube algorithm
- The inbetween contour is the slice at the middle of this 3d surface

In our perspective, this paper is interesting for two main reasons: first, because it breaks the character into several layers, it can handle some kind of topological events, this is what Nitzberg [32] call a 2.1D animation in 1993. However, layering is a very classical tool provided by any decent animation software, and we want to handle are the other types of topological events (layering would anyway be added to the interface afterward for simple cases). The second reason is that it generates a Space-Time surface (defined by an implicit function in their case) to compute the inbetweens for each layer, which is in the spirit of our method (even if in practice we are really far from this approach).

Of course, when it comes to reusing existing motion, rotoscoping is a nice idea. An interesting method is proposed by Agarwala et al. in 2004 [1]. Guided by the user, rotocurves are extracted from the videos to track in time actual curves existing in the video. The, the user can draw a completely different drawing, and the stroke drawn follow the paths of the rotocurves.

There are also “Image morphing” methods (do not confuse with shape morphing), they intend to create inbetweens of video frames. The state of the art method based on the optical flow can generate tight inbetweens of 2 frames of a video, which is proposed by Mahajan et al. in 2009 [29]. However, it is mostly intended to be able to change the frame rate of a video, for instance by doubling it, and cannot work for our case where in

the one hand we have a drawing instead of a photography, and where the time distance is way more important.

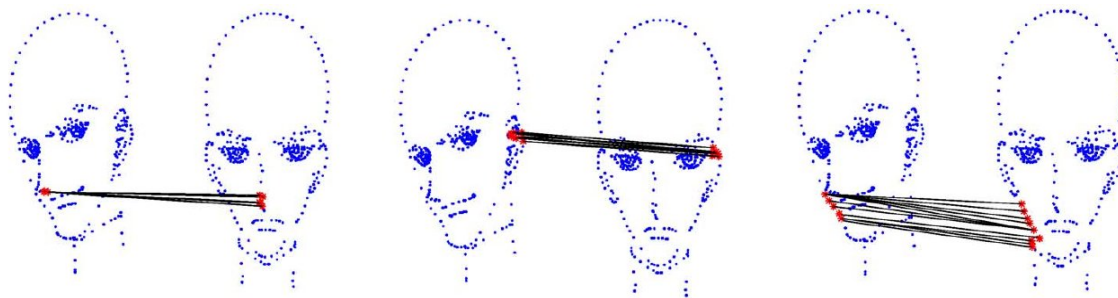


Figure 2.3: Example of many-to-many point correspondences (image from Yu et al. 2011 [54])

Recently, in 2011, a method based on machine learning has been proposed by Yu et al. [54]. They sample the strokes into points, and for each point they compute a local shape descriptor. Using semi-supervised learning taking into account local and global consistency [56], they are able to build a many-to-many correspondence between the sampled points, as you can see on Figure 2.3. This method has then been slightly improved in 2012 by the same authors [53], using a different technique of learning (*Graph-Based Transductive Learning* instead of *Patch Alignment Framework*). The main advantage to these methods (either the original or the improvement) is that it is robust, to a certain extent, to topological inconsistencies, which is what we desire in our case. However, the output of the algorithm is not enough to compute inbetweens: the many-to-many point correspondence, even if we assume that there is no mismatch (which is generally not the case), doesn't tell us how to interpolate the curve in those cases of topological inconsistencies.

2.4 Using Stroke Graphs

The last class of approaches, using *Stroke Graphs*, is by far the less explored one, while it seems (for reasons that will be explored throughout this report) the most natural to describe drawings as well as the most suitable to compute clean inbetweens. In order to very fastly convince you with a non scientific argument, just consider that the state of the art in inbetweening is the BetweenIT system, by Whited et al. 2010 [51], research coming mainly from Disney Research and that has been actually used in *The Princess and the Frog*, the last feature-length animation of the Walt Disney Animation Studios. The method using machine learning previously examined, while more recent, doesn't provide any final result of inbetweening, but only many-to-many point correspondences. It is still unclear how to exploit this information, but as an oracle helping another stroke matching algorithm.

2.4.1 Definition

A Stroke Graph is a vectorial representation of your drawing storing the topology as well as the geometry. To my knowledge, there are two different ways to represent it: one

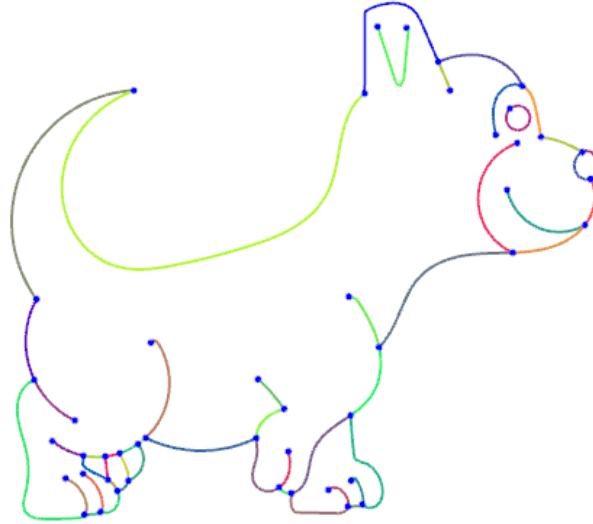


Figure 2.4: an example of Stroke Graph: each blue point is a node of the graph, while each stroke can be made of several edges, represented using different colors

is introduced by Kort in 2002 [24], and the other by Whited et al. in 2010 [51], which is in my opinion more suitable for our needs, and then the one I will use and describe here.

When sketching, an artist generally draws in pencil a lot of very small strokes to incrementally build a larger curve by refining it constantly: I call these little strokes *pencil strokes*. When the artist is done with his design comes the “inking stage”: he draws in ink, on top of the pencil strokes, a continuous curve “without raising the pen” as long as possible: I call such a stroke an *ink stroke*. Then the pencil strokes can be erased revealing the cleaner drawing with only few ink strokes (Of course, ink can be replaced by any other material that will not be erased, and is in fact more and more often done digitally). In all this paper, I will never consider pencil strokes, and then “stroke” will simply be a shortened version of “ink stroke”.

The set of all strokes in our drawing is then a vectorial representation of our drawing. Each stroke has a geometry associated, a piece-wise C_1 curve (the artist can stop drawing without raising the pen, and then continue in another direction. In those particular points the curve is only C_0). These strokes can intersect or join into a T-junction, and every of these intersections is a *Node* of the Stroke Graph. Every stroke has also two ends (except for loops), those point are also *Nodes* of the Stroke Graph. The edges are the sub-strokes that link the nodes together. An example of such a Stroke Graph is represented in Figure 2.4.

2.4.2 Inbetweening With Stroke Graphs

The goal of the research of Kort in 2002 [24] was very close to the one in this report: handle the case where topological changes appears. However, it has been tackled in a very different (and not very successful) way. Instead of considering nodes at the intersections of strokes and T-junctions, it consisted of keeping the strokes entirely, and defining different

relations between them (for instance “intersecting” or “dangling”). Then, a series of logical expressions were designed in order to detect two T-junctions corresponding to the same hidden stroke, which was then reconstructed. The main advantage is the originality of the approach, but in fact it worked only on very simple examples.

The approach of Whited et al. in 2010 [51] is very different: what they target is “tight inbetweening”, eg inbetweening for two very close drawings. The reason comes from the smart observation that those tight inbetweens are the more tedious to draw manually because of the accuracy it requires (small unaccuracy would be noticed), while at the same time they are the simplest to compute automatically. Then, it is assumed that the graph topology is isomorphic, or nearly isomorphic. The artist manually selects a pair of corresponding edges in the stroke graph (or several of them) to initiate an algorithm based on graph traversal and simple edge similarity metrics, with real-time feedback. Wherever the graph topology is the same, because strokes are very similar (tight inbetweening), this simple heuristic performs well and the algorithm matches every stroke of a connected component. The artist can run again the algorithm with other feeds to match the remaining strokes when topology has changed or when several connected components exist. The few topological changes are handled this way: if a stroke appears in only one keyframe, the user needs to manually draw the hidden line in the other keyframe, and where too much differences exist (a rotating hand is given as an example in their paper), the artist has the opportunity to draw himself all the inbetweens for this local difficult part of the drawing.

Then, this method has the main advantage to reduce the artistic cost when inbetweens can be trustfully computed, and the artist can focus on the parts where artistic interpretation is necessary. But a user interaction is always necessary, and it does not provide an abstract representation of topological events, neither a continuous transformation when these topological events occur (the user needs to draw each frame manually in this case), neither take into account these topological events in their matching algorithm, at the contrary of our approach.

2.4.3 About Graph Isomorphism

Surprisingly, even though stroke graphs are involved, the bibliography about graph theory doesn’t appear in inbetweening research papers. In an abstract point of view, it is clear that the problem can be formulated as belonging to the *Graph Isomorphism* class of problems. I present here a hierarchy of sub-classes of these problems, taken from a talk of Christine Solnon in 2009:

- Graph Isomorphism. This is an equivalence relation: “Are the graphs strictly isomorphic?”. The complexity of this problem is called Isomorphic-complete, which is considered rather easy in practical cases. We should notice that this is more or less the configuration considered by [51]
- Sub-Graph Isomorphism. This is an inclusion relation: “is this graph a strict sub-graph of that graph?”. This simple extension of the problem above is already NP-complete... but still tractable for “medium” size graphs.
- Maximum Common Subgraph. This is an intersection relation: “what are the two subgraphs of the two given graphs, which are isomorphic and maximize their size?”. Since it includes the problem above, it’s of course a NP-hard problem, and is gen-

erally untractable for actual application cases.

- Graph Edit Distance. Find the best univalent matching (one node from first graph matched to at most one node of the second graph) that minimizes the edition costs. it is an NP-hard problem more complicated than the previous one.
- Extended Graph Edit Distance. Find the best multivalent matching (one node from first graph can be match to any number of nodes of the second graph) that minimizes the edition costs. NP-hard problem even more complicated than the previous one.

The first three problems are the more classical and treated in early literature. For instance, with as input two graphs with weighted edges and the same number of nodes, Umeyama in 1988 [49] finds the best matching using a spectral analysis of the adjacency matrix.

In our application case, this enumeration informs us about lots of bad news. The first important one is that not only we are in the latter case (the most complicated one), but in fact the “edition” is more complex than just adding nodes and edges, popping from nowhere. We are looking for a continuous geometric transformation from the edges of the first graph to the edges of the second graph. In short, we are looking for a “morphing” of strokes, not only a “matching”. In addition, aesthetics criteria should be considered: Even though it could be considered in designing an edit distance, it’s not even sure that the global minimum of any edit distance (assuming we could compute it) would give an appropriate result. Lastly, we would like something the user can eventually interact with, then it should be fast, possible real-time (eg far from the algorithms used in the problems above)

However, there is still some hope for the following reasons. First, we have a lot of information attached in the stroke graph (because the edges are embedded in \mathbb{R}^2), which not only makes the problem simpler because the graph is planar, but in addition the geometric information can help us in guessing the matching strokes. In addition, we would like something the user can eventually interact with. This is indeed also good news: it means the user can guide the algorithm, we are not necessarily looking for fully-automatism, even if ideally it would be the goal.

The thesis of Sébastien Sorlin in 2006 [42] and the following works provides good idea on how to solve this latter class of problems, with possibly labeled graphs which is our case. The different approaches are using global constraints to narrow the variable domains [43], using a reactive tabu search [44] or more recently using filtering algorithms [55, 45].

2.5 Restricting the Class of Animation

This is not really an approach for inbetweening, but a technique to animate that implicitly solve the problem of inbetweening, and is widely used in practice by some animation studios. It consists, instead of drawing separate frames of the animation, in performing 2D animation in a similar way than 3D animation: Through a dedicated interface (for instance ToonBoom or Adobe Flash), the artist create some vectorial 2D object that he will animate through time. The animation can be done either roughly by keyframing for instance properties of the object such as its position and scale, but also by keyframing the position of the control points of the vectorial object. By using several layers, it is possible to create complex animations.

However, to handle subtle topological changes, one need to carefully design the layers and use some tricks, what we would like to prevent for a begginer user. To tackle this problem, some research has been done in “2.5D animation”. It is important to specify that the term is not used in the same way as the one in the title of this report. The aim of these methods is to build a model of the 2D object we want to animate using some 3D information. To my knowledge, Fiore et al in 2001 [18] is the first having proposed such a method. However, it required a lot of intervention from the user.

More recently, Rivers et al. in 2010 [35] proposed a new 2.5D model which can be more easily created: the user draw part of the object in different view, and then the system automatically compute the 3D *anchor point* of this 2D drawing. Then, it is possible to turn around the model: the 2D shape is interpolated in 2D, but its position is given by the 3D position of the anchor point, which gives a nice feeling of consistent 3D, keeping a 2D style. Since it is still tedious to create those 2.5D model, An and Cai have designed in 2011 [17] a method to generate it from a 3D model.

However, if you are a casual drawer, you do not have 3D skills, and then this approach is not really adapted. Also, it can only handle the topological changes that can be solve through simple layering.

3 Space-Time Topology

3.1 Motivations

For the reasons stated in the introduction, if one want to be able to create an intuitive interface for 2.5D rotoscoping, an intermediate goal is to be able to generate automatically (or with very few intuitive user interaction) *inbetweens* from two drawings with *topological inconsistencies*. What we call inbetweens are all the intermediate frames between the two input drawings (which are called *keyframes*), so that the animation can be played smoothly, for instance at a rate of 24 frames per seconds (12 frames per second is often used in traditional 2D animation to save drawing time). Two drawings are called *topologically inconsistent* when their associated stroke graphs (cf Section 2.4) are not isomorphic¹. To clarify the ideas, the Figure 3.1 is an example of such topologically inconsistent stroke graphs.

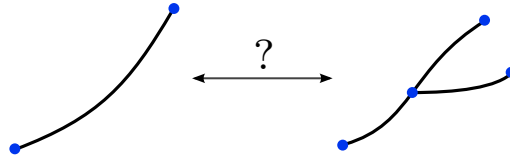


Figure 3.1: An example of two stroke graphs with topological inconsistencies.

Now that I've shown this simple example which is going to be our visual support throughout the description of our *Space-Time Topology*, you are probably wondering:

1. Why should we care about automatically creating inbetweens of this?
2. Does it happen in practice?

The answer to the first question is very simple: because we don't know what the user will draw (especially because it is a non-skilled drawer that have no idea how the interface work), and we do want our system to be robust to any kind of input. Of course, it is impossible to give the "perfect" inbetweens for any drawings, because sometimes such perfect inbetweens do not exist: this is the case when even two professional inbetweeners would draw different inbetweens for the same pair of keyframes. This happens when there is no clear human interpretation of what the drawings represent, neither we manage to find which strokes are in correspondence based on geometric similarities, then finding what's going on between the two is problematic. But we can still define what an ideal system should do in every cases:

- generate the good inbetweens for all sub-part of the drawings where human professional inbetweeners agreed on how they should transform

1. two graphs are isomorphic if there exist bijective mappings φ between the edges and ϕ between the nodes which are compatible with the relations $R(n, E = \{e_1, \dots, e_k\}) = "E \text{ is the set of edges incident to } n"$ (eg $R(n, E) \Leftrightarrow R(\phi(n), \varphi(E))$)



Figure 3.2: Two examples of inbetweens from two different studios. Above is *The Princess and the Frog* from Walt Disney Animation Studios and below is *Little Nemo: Adventures in Slumberland* from Tokyo Movie Shinsha. At the left is the first frame, at the right is the last frame, and in the middle are the inbetween frames, duplicated to highlight a specific part.

- generate inbetweens with the “lower visual impact” for the others, so that our attention is focused on the parts correctly inbetweened

Then comes the answer to the second question: Yes, in this particular case, it does happen in practice, and I think professional inbetweeners would agreed on how to inbetween this. To demonstrate this, let’s have a look to professional animations in Figure 3.2. Parts of the drawings contains topological inconsistencies similar to the one that has been presented (highlighted in red). An interesting point is that the reason why these topological inconsistencies occur is different: in the first case it is line at a different depth which become hidden by the arm, while in the second case it is a fold of the blanket which is created from an existing fold. But yet, the way it is inbetweened is similar: the new edge is “growing” from the existing edge. Then, whatever the semantic reason of the topological inconsistency in Figure 3.1, an automatic inbetweening system should probably generate inbetweens such as presented in Figure 3.3.

Now that we know what “should look like” the inbetweens for an input that “looks like” the Figure 3.1, we still need to decide on our approach to tackle the problem. The very first question to answer is how the input and output are represented. This is the most preliminary question one can ask, but this what is all about this chapter called *Space-Time Topology* and then almost half of my Master Thesis research.

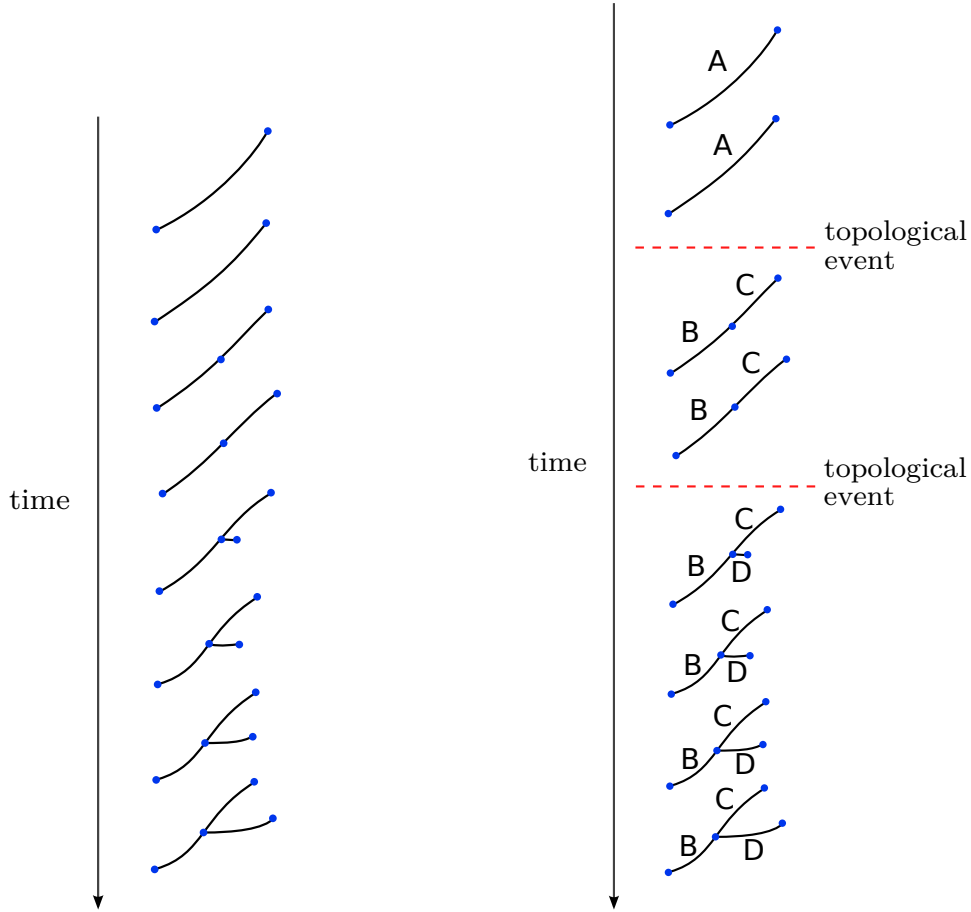


Figure 3.3: Left: how we would like the stroke graph to evolve across time. Right: Naming the edges makes it easy to locate the two topological events, at time t_1 and t_2 .

3.2 Stroke Graphs As Input

First, we always have the possibility to choose between a pixelized representation of our images or a vectorial one. They all have their pros and cons. One main advantage of vectorial representations is that they achieve easily a high quality rendering of our animation. Moreover, a shaky drawing with a mouse from a non professional drawer can be more easily turned into a nice smooth vectorized curved. Then, the style of the curve can be easily modified, it is possible to zoom in without any loss of quality, and the curves can be easily edited by hand if necessary.

On the other hand, considering pixelized images could also be a good idea. They have the main advantage of having more freedom: everything can be drawn, the quality only depends on the artist imagination, time and skills. In addition, the quite good results of Sykora et al. in 2009 [47] tends to made us think that inbetweening algorithms blending pixels together could work: it makes them much more “flexible” especially in our case: performing a “topological operation” on a pixelized image is in fact trivial because there is no topological structure by essence, we just need to combine values of pixels for instance.

However, my belief is that a good representation for inbetweening can only be vectorial. The first obvious advantage is that it guarantees that the rendering will be clean



Figure 3.4: A pair of stroke graphs containing lots of topological inconsistencies. The design is taken from Catmull 1978 [13]

and similar to the input frames, without any blurring issue. The second advantage is that the inbetweening obtained has way more useful information: not only any arbitrary number of frames can be generated (but it could also be the case of pixelized algorithms), but it tells you what it transformed into what. Then it can later be easily modified, processed, slowed in or out. A typical example is coloring: the reason of the existence of the color transfert method Textoons [46] from Sykora in 2011 is that it wasn't easy to do with his inbetweening method, while the problem doesn't exist with a vectorial algorithm: we know exactly what are the faces all along the animation, and transferring the coloring from one frame to all the other is trivial. Finally, I also believe that most of the information necessary to compute a good matching between strokes (and then do a good inbetweening) is encapsulated in the topology of the stroke graph, which can be computed only with a vectorized representation. The article from Whited at Walt Disney Animation Studios in 2010 [51] is mainly what makes me think so, when we see how easy it is to compute the matching by propagating good results.

For all these reasons, we choose that the input of the algorithm we are looking for is a vectorial drawing, with its 2D topology: eg the stroke graphs themselves, which are the same input used in [51]. For instance, a simple input could be the one presented in Figure 3.1, and a very complex input could be the pair of stroke graphs shown in Figure 3.4.

3.3 Animated Stroke Graph As Output

3.3.1 Abstract Data Type

Naturally, as we choosed stroke graphs for the representation of our drawing as input of our inbetweening algorithm, it makes more sense to have an output of the same nature rather than, for instance, a sampled pixelized animation at 24 frames per seconds. Then, an abstract definition of what our output should be is a representation that can give us, for any $t \in [0, 1]$, a stroke graph called $\text{StrokeGraph}(t)$ (see Figure 3.3, left side). But since for every t , $\text{StrokeGraph}(t)$ encapsulates the spatial neighbourhood information (eg which points belong to the same edge, and which edges have a common incident node), it is also more consistent if our representation encapsulates the temporal neighbourhood information. This temporal neighbourhood is which points (not necessarily at the same time) corresponds to the same edge, and which edges are “temporally neighbours”. In the same way as two edges are spatially neighbours is they share a common node, two edges are said temporally neighbours if they share a common *topological event*. For instance, in the Figure 3.3 (right side), we can see that A and B (as well as A and C) are temporally neighbours. To conclude, the abstract data type² *animated stroke graph* of our output should provide:

- the vectorial geometry of each edge at each time
- the spatial relationships between edges
- the temporal relationships between edges

We will now define how this abstract data type can be represented mathematically, object combining both *space-time geometry* and *space-time topology* (see Figure 3.5), as well as how it has been implemented.

3.3.2 Mathematical Representation

As for any vectorial representation of a drawing, the edges of a stroke graph are 1D objects embedded in a 2D space: each edge E is for instance described geometrically by a parameterized curve

$$\begin{aligned} \mathcal{C} : [0, 1] &\longrightarrow \mathbb{R}^2 \\ u &\longmapsto (x, y) \end{aligned}$$

If this edge is animated, then for each t in its temporal domain³ \mathcal{T} , its geometry is represented by a new curve \mathcal{C}_t . But there is no reason to discriminate space and time in this notation, and the geometry of our *animated edge* is in fact defined by

$$\begin{aligned} \mathcal{C} : \mathcal{T} \times [0, 1] &\longrightarrow \mathbb{R}^2 \\ (t, u) &\longmapsto (x, y) \end{aligned}$$

and then has been upgraded to a 2D object thanks to the time dimension. To visualize it, it is useful to embed it in the 3D space (x, y, t) , that gives us a classical surface. If we do

2. From Wikipedia: an abstract data type is a mathematical model for a certain class of data structures that have similar behavior.

3. should be a closed connex subset of $\overline{\mathbb{R}} = [-\infty, +\infty]$ (note the closed bracket), except at $-\infty$ and $+\infty$ where it is possibly open. For instance $[-\infty, t_0]$, $]-\infty, t_0]$ and $[t_0, t_1]$ are valid temporal domains, but $]t_0, t_1]$ is not valid is t_0 if a finite value.

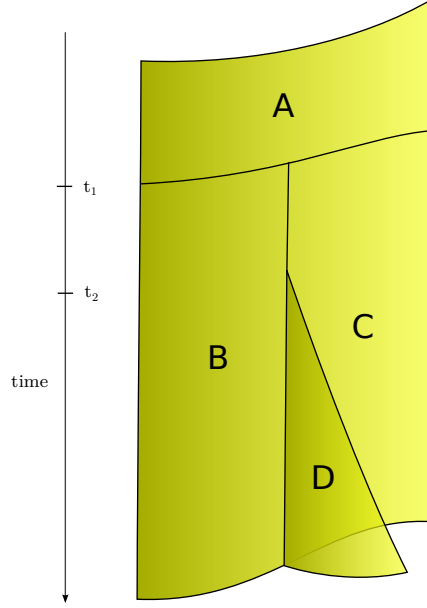


Figure 3.5: The mathematical space-time representation (geometry and topology) of the animation in Figure 3.3. The space-time geometry is a non-manifold 3D surface (piecewise C_1), while the space-time topology is a non-manifold 3D topological mesh, here with 4 faces: A, B, C and D (If the geometry is described itself by a mesh, then the topology is a meta-mesh of the surface)

this for every animated edge of the animated stroke graph, we finally get the *space-time geometry*. In the case of our example, it can be visualized in Figure 3.5. Although this 3D geometric representation of a 2D animation as a surface seems really natural and adapted for inbetweening (the goal is to find this surface given the extremal edges), it has not been much used in previous research. To my knowledge, only De Juan and Bodenheimer in 2006 [15] briefly use it, defined as an implicit surface extracted using the marching cube algorithm, and in a domain different from inbetween Buchholz et al. in 2011 [11] use it extensively to get a time-consistent parameterization of silhouettes in stylized rendering of 3D animation, given the fully pre-computed animation.

However, this recent work, even if it tracks the topological events in order to find this time-consistent parameterization, do not have any real topological representation of the 2D animation: it only uses the space-time geometry, represented by a classical triangle mesh. Our approach extends this space-time geometry by including topological information to be able to use it has an animated stroke graph.

Mathematically, as can be seen in Figure 3.5, this space-time topology has the structure of a non-manifold 3D topological mesh, and this is all we need to represent an *animated stroke graph*, since it provides everything we need:

- For any given t , $\text{StrokeGraph}(t)$ is obtained by intersecting the space-time representation with an horizontal plane: the stroke-graph edges are where the plane intersects space-time faces, while stroke-graph nodes are where the plane intersects (non-horizontal) space-time edges.
- The topological events are represented by the space-time nodes (and the horizontal space-time edges).

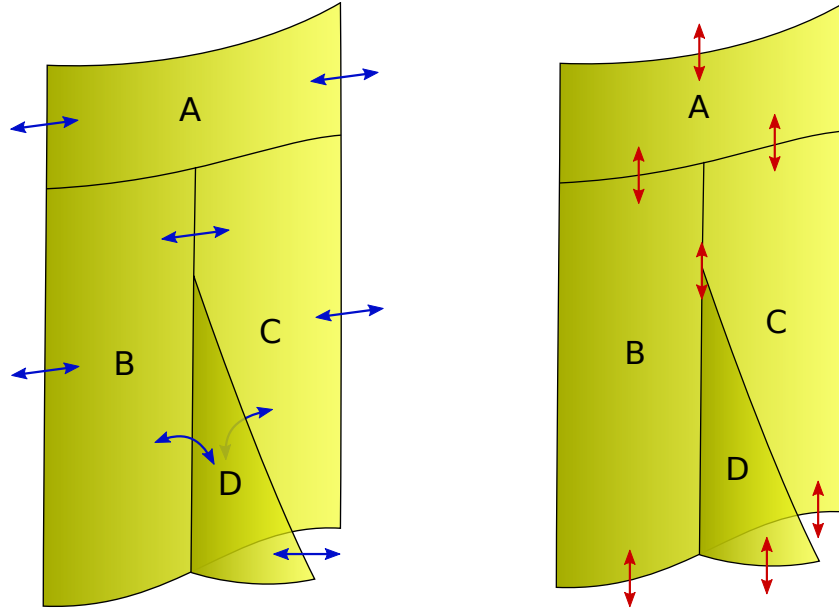


Figure 3.6: The blue arrows represent the spatial neighbourhood relationships (left). The right arrows represent the temporal neighbourhood relationships (right)

- Two stroke-graph edges are spatially neighbours iff their space-time face share a common (non-horizontal) space-time edge
- Two stroke-graph edges are temporally neighbours if their space-time face share a common space-time nodes.

We can observe that every object is downgraded by one dimension when removing the time component: faces become edges, edges become nodes, and nodes (which represent the topological events) doesn't exist anymore, as one would expected from a non-spatial object, representing an evolution through time.

3.3.3 Data Structure

By looking at the mathematical representation described in the previous subsection, one could implement an animated stroke graph by implementing a general non-manifold topological data-structure, where each topological face links to its actual geometry, and it would probably be completely fine. However, because our mathematical world is a space-time world (2D+1D) horizontal edges has a different interpretation than non horizontal edges, and it seems relevant not to describe them with the same object (eg keeping separate spatial information and temporal information). The data structure implemented, named *space-time stroke graph* (STSG) is more specifically designed for our needs, and the most important aspects are presented here.

It is composed of three types of objects: *edges*, *nodes* and *events*. The role of edges is to encapsulate the space-time geometry, the role of nodes is to encapsulate the spatial relationships, while the role of events is to encapsulate the temporal relationships. These relationships are represented in Figure 3.6.

Edges

An edge corresponds to a face of the space-time topology (*Animated edge* is a more accurate name, but it has been shortened for convenience). For instance, the animated stroke graph represented in Figure 3.5 contains four edges: A, B, C and D. Its main role is to contain the information about geometry, which is given by a function:

$$\begin{aligned} \mathcal{C} : \mathcal{T} \times [0, 1] &\longrightarrow \mathbb{R}^2 \\ (t, u) &\longmapsto (x, y) \end{aligned}$$

Edges should be considered as not oriented. However, because it is useful for matching to work with oriented edges, and to navigate through the graph, *half-edges* are defined. They are simply containing a pointer to an edge and a boolean to indicate if the orientation is from $u = 0$ to $u = 1$, or the opposite. They support the classical operations $\text{next}(t)$, $\text{previous}(t)$ and $\text{opposite}(t)$ (spatial neighbourhood depends on t , cf the edge B and C of our example).

Nodes

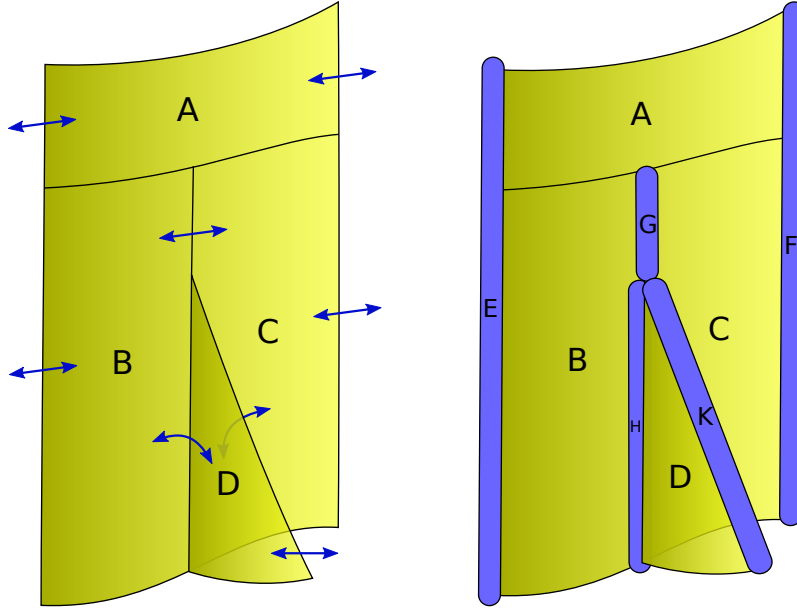


Figure 3.7: The blue arrows represent the temporal relationships between the spatio-temporal edges. *Spatio-temporal nodes* are the objects whose role is to store those relationships.

A node corresponds to the spatial relationships between edges, cf Figure 3.7: there are five nodes named E, F, G, H, and K. At any time t , an edge has exactly two pointers to nodes: $\text{node}_0(t)$ and $\text{node}_1(t)$ (corresponding to the neighbourhood at $u = 0$ and $u = 1$). Loops are supported: in this case the two pointers are equals (pure topological loops without a node breaking the circle are not supported). A node contains a list of pointers to its neighbours: edges(t).

Because both edges and nodes are defined over a temporal domain \mathcal{T} , they can be both designed under the common terminology *animated elements*,

Events

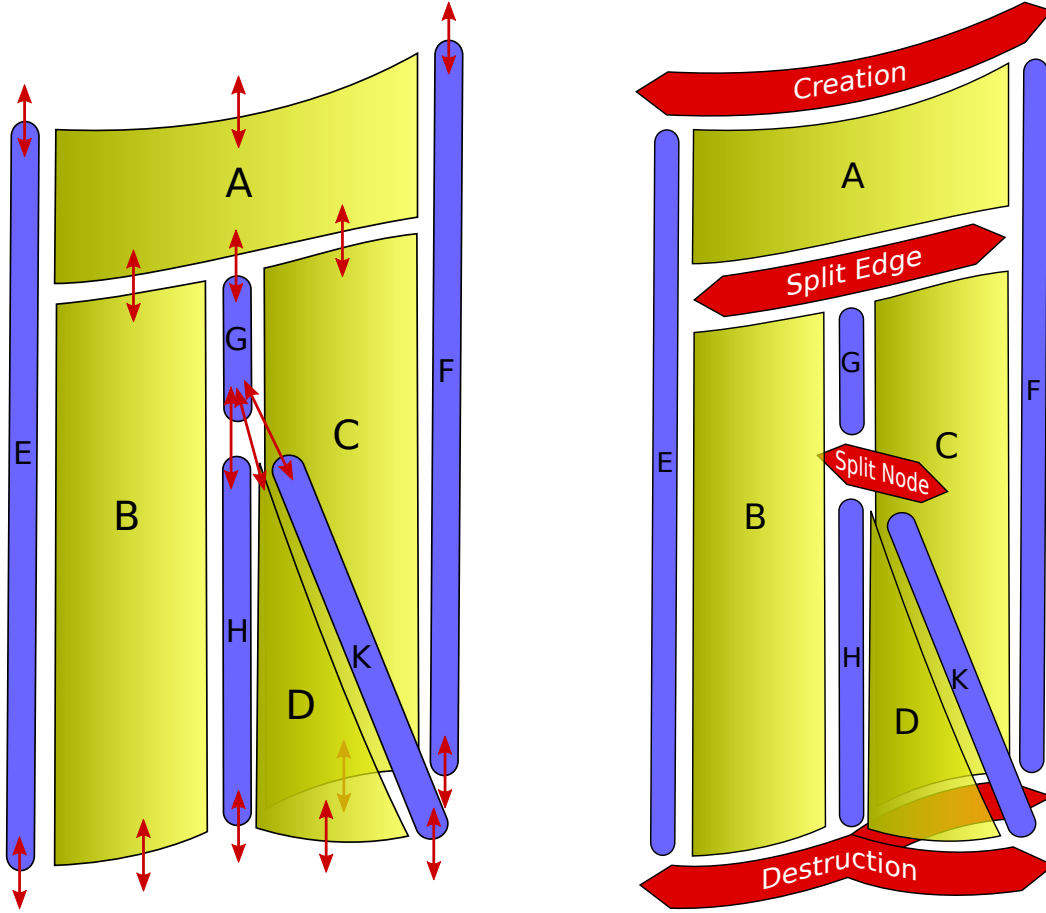


Figure 3.8: The red arrows represents the temporal relationships between the “animated elements” (either nodes or edges). *Topological events* are the objects whose role is to store those relationships.

An event corresponds to the temporal relationships between animated elements (either edges and nodes), cf Figure 3.8: there are in this example four events which are named Creation, Split Edge, Split Node and Destruction.

An event is defined by a specific finite or infinite time (the creation can appear at $t = -\infty$ for instance), as well as the list of *input elements* (the elements that do not exist anymore after the event), and a list of *output elements* (the element that only exist from the event). Every animated element contains a pointer to its start event and its event, which always exist (but can be at infinite time). You can notice two types of duality in this representation:

space-time duality Every edge is spatially delimited by two nodes, and equivalently every animated element is temporally delimited by two events.

edge-node duality Split Edge is an event which transform one edge into two edges separated by a node, while Split Node is an event which transform one node into two nodes separated by an edge.

We have already seen that the spatial neighbourhood depends on t . In fact, the neighbourhood change each time a neighbour is destructed/created by an event. For instance, the neighbour of the node E is initially the edge A, but the Split Edge occurs and then its neighbour becomes B. Then, to compute the function $\text{neighbours}(t)$, of the node E, what we do is to also store a pointer to Split Edge: Split Edge is called a *side event* of E, and E is called a *side element* of Split Edge. $\text{Neighbours}(\text{startTime})$ is initially given by the start event, and then every side event informs about the changement of neighbourhood, until we arrive to the end event. This operation can be done either by chronological order, or reverse order: the data structure is completely symmetric, every event is revertible.

3.4 Atomic Events

Our structure can allow any kind of events: an event is entirely described by:

- its input elements, and a function giving for each input element what are its neighbours just before the element is destructed by the event.
- its output elements, and a function giving for each output element what are its neighbours just after the element is created by the event
- its side elements, and a function giving for each side elements, which of its neighbours are destructed by the event, as well as which new neighbours does the element have after the event

With this information, all the spatial and temporal relationships of any element can be easily retrieved when asked: there is no special need to be able to manipulate the structure to differentiate for instance SplitNode events and SplitEdge events. However, it is more convenient to create special classes to distinguishes the events according to their *type*: how many nodes/edges do they create/destroy, and what are their spatial relationships. This way, by having predefined objects for common events (such as the SplitEdge and SplitNode already seen), it is easier to operate on the structure using these predefined events than specifying the elements and functions presented in the list above.

Atomic events are a subset of all possible events, which are able to “simulate” any other general event when combined, and that cannot be themselves simulated by simpler events. Their interest is that any animation can be decribed by using only those atomic events. Of course, the space-time topology is not the same by using two sequential events separated by $dt = 0$, or directly the biggest event, since intermediate elements are created in the first case, even if they have a null lifetime. However, it would be visually the same when played, and this is our final application.

The atomic events can be separated into two categories:

- The *continuous events*: they create elements temporally neighbours of existing elements, and then should ensure a geometric continuity with them. This is for instance the case of SplitNode and SplitEdge already presented.
- The *non-continuous events*: they create elements with no temporal neighbours: then no geometric constraints should be met, they will appear as popping from nowhere.

The Figure 3.9 enumerate the list of atomic continuous events, while the Figure 3.10 enumerate the list of atomic non-continuous events. If we call N^- (resp E^-) the number of nodes (resp edges) in the stroke graph before the event, and N^+ (resp E^+) the number of nodes (resp edges) in the stroke graph after the event, then the first column indicates $(\Delta N, \Delta E) = (N^+ - N^-, E^+ - E^-)$, which highlight the duality between nodes and edges in the terminology. An event is always invertible, and the invert of an atomic event is also atomic. In these tables are only listed the event in their “natural direction” (eg creating elements), reversing the direction of the red arrow gives you the inverted atomic event.

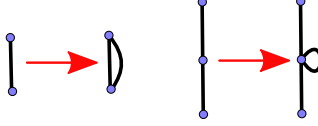
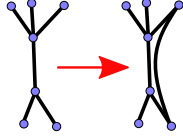
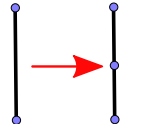
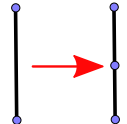
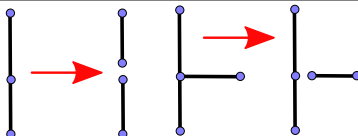
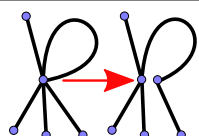
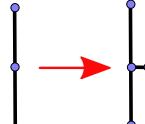
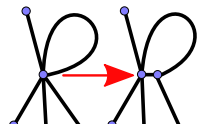
| $(\Delta N, \Delta E)$ | Name | Common Cases | General Case |
|------------------------|---------------|---|---|
| $(+0, +1)$ | EdgeDuplicate |  |  |
| $(+1, +1)$ | EdgeSplit |  |  |
| $(+1, +0)$ | NodeDuplicate |  |  |
| $(+1, +1)$ | NodeSplit |  |  |

Figure 3.9: The enumeration of the four continuous events.

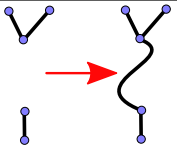
| $(\Delta N, \Delta E)$ | Name | General Case |
|------------------------|--------------|---|
| $(+1, +0)$ | NodeCreation | $\emptyset \rightarrow \bullet$ |
| $(+0, +1)$ | EdgeCreation |  |

Figure 3.10: The enumeration of the two non-continuous events.

4 Automatic Inbetweening

Now that a data structure has been defined to represent an animated stroke graph, with topological events, we need to construct such an animation from two input stroke graphs to generate the inbetweening, as is presented in Figure 4.1

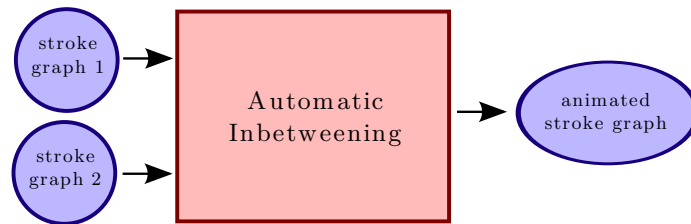


Figure 4.1: The input and output of our algorithm.

To achieve that, the input stroke graphs need first to be segmented. Then, they are combined into a single trivial animated stroke graph, where all the edges from the stroke graph 1 disappear instantly, and all the edges from the stroke graph 2 appear instantly. This animation is finally iteratively refined: a set of operators are generated, and we apply the one of them, depending on the strategy chosen by the algorithm. We want to find an animation that has all its edges bounded, we return this animation. This framework can be seen in Figure 4.2.

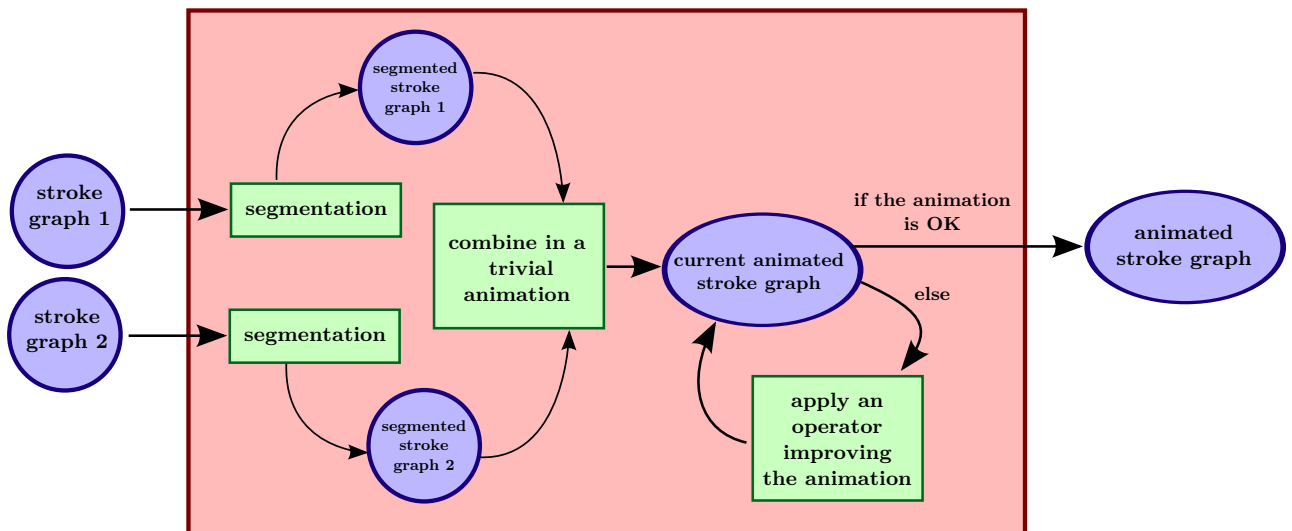


Figure 4.2: The algorithm with some little details.

4.1 Initialisation of the algorithm

4.1.1 Providing Input Stroke Graphs

To obtain the input stroke graph, one can:

1. directly specify them (for instance Bezier control points and tangents)
2. draw on screen and fit to a vectorized line
3. take as input a pixelized image and retrieve the vectorized strokes

This list is sorted in increasing order of simplicity for the user, but decreasing order of simplicity and reliability of the algorithm. For the sake of my research, I've decided to choose the second one: It is a compromise between implementation time and convenience for the user. Clothoids are used for the vectorial representation, which are fitted from mouse input using Ilya Baran's own implementation of [4].

4.1.2 Segmentation

Because we want to be robust to topological changes, the first thing we need to do is to segment the input stroke graphs. For instance, we can see in Figure 4.3 that if we do not perform any segmentation, the first drawing is composed of three edges, while the second one is only composed of one drawing, and then we would have no chance to match some edges.

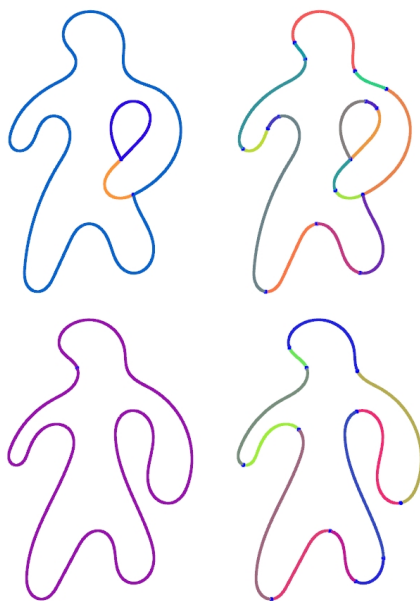


Figure 4.3: Left: without segmentation. Right: After segmentation.

For the reasons detailed by Leyton in 2006 [27], meaningful points on a shape are the extrema of curvature, since we perceptually decompose a shape at these points. This approach has also been chosen by Baxter et al. in 2009 [5] and appeared to be successful. Then, this is where we want to segment our edges. However, finding these local extrema of curvature is not trivial in the general case, since because of noise we will get a lot

of false positive. This way, what we look for are in fact *stable* extrema of curvature, a method has been designed by [5] to compute then.

This method is far from obvious to implement, but this is where comes the choice of clothoid curves for our vectorial representation. By definition, a clothoid spline is a curve whose curvature is piecewise linear. Then, once the curve is fitted into a clothoid, the information about extrema of curvature is already encapsulated, there is no computation to do at all. The curve is given by its local minima and maxima of curvature, which are stable thanks to the good quality of the fitting. An example of segmentation is given on Figure 4.3.

4.1.3 Providing First Seeds

In the same way as [51], our algorithm first start by binding two edges together, and then try to bind the edges of the neighbourhood of edges already binded.

But then, the choice of the first binded edge is really important: we need a good metric for that. Our approach consisted in taking into account three features of an edge: the length, the mean curvature (multiplied by the length, so that this value equals 2π for a circle), and the *diff curvature* (the difference between the max of curvature and the min curvature, multiplied by the length). We know that it is a good metric, for instance, to use the ratio of the logarithms of the lengths. However, it was less clear for the mean curvature and the diff curvature. In order to know how much those features impacted on the “difference” on two curve, we made a little perceptual study, to answer manually the question: “should the two edges be considered as a potential match” for different pairs of values of a specific feature. The results can be seen on Figure 4.4

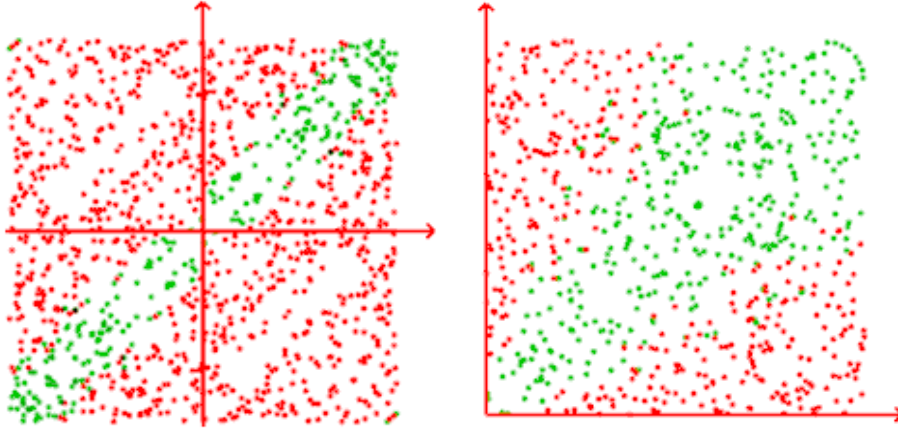


Figure 4.4: Green points indicates that we should consider it, red points that we shouldn’t (based on that information alone). Left: mean curvature, between -2π and π . Right: diff curvature, between 0 and 4π .

Based on this information, for a value x of our feature, we defined $\delta(x)$ the distance such that the pair $(x - \delta(x), x + \delta(x))$ is no longer among the green points. Then, for any given x_1 and x_2 , we define our metric of confidence, between 0 and 1 as:

$$e^{-\frac{dx^2}{2*\delta(x)^2}}$$

where $x = \frac{x_1+x_2}{2}$ and $dx = |x_2 - x_1|$

Finally, we compute the histogram of the features among all the edges of the drawings, and divide the metric by the number of edges with similar values of this feature. This way, two edges obtain a high score not only if they are in the green area for each three features, but also if their values of features are different enough from the other. To summarize, a good score is obtained when two edges are similar, and at the same time different from all the others.

With the features selected, which are rotation and position invariant, it improved drastically the matching than for instance using the ratio of the features.

4.1.4 Creating the Initial Animation

The last step of the initialization consists in creating the initial trivial animation, which will be iteratively improved. This initial animation is simply composed of the edges from the first drawing that appear at $t = -\infty$, then disappear at $t = 0$, then the edges of the second drawing appear at $t = 1$ and disappear at $t = +\infty$. This animation can be visualized by its space-time topology in Figure 4.5.

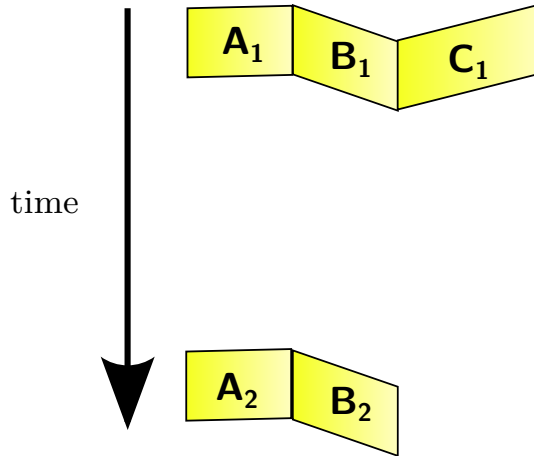


Figure 4.5: The initial animation displayed as a space-time topology. The first drawing was composed of the edges A_1 , B_1 and C_1 , while the second drawing was composed of the edges A_2 and B_2 .

4.2 Operators: One Step of the Algorithm

At each step of our iterative algorithm, we will consider several operations that could be done to advance one step further in the creation of the animation.

Binding a new edge If there is no already binded edges (as it is the case for the initial animation), or if no one of the binded edges have unbinded neighbours, then the only thing we can do is binding a new edge, not connected to what has already been bindind. This operation can be seen in Figure 4.6. The potential edges considered in this case are the best matches provided by the first seeds computation in the initialization.

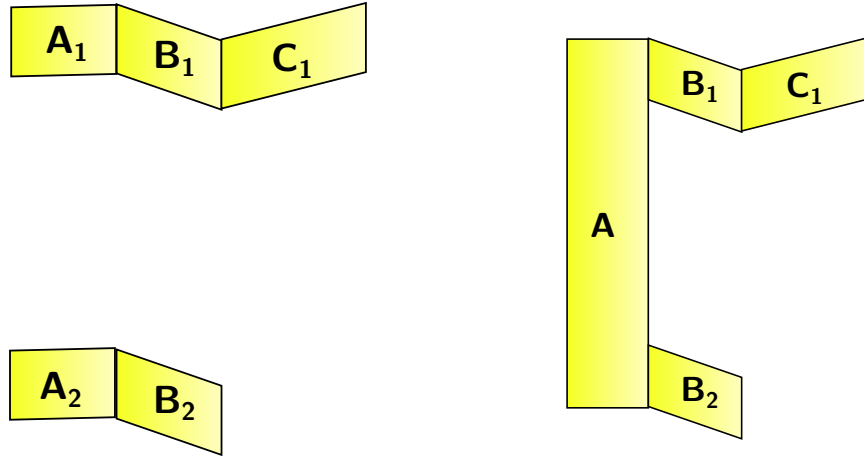


Figure 4.6: The result of binding a new edge. Left: before the bind. Right: after the bind

All the other three operators are used in the neighbourhood of an already binded edge.

Binding a neighbour edge This is the same operation as the previous one, except that it happen next to an edge already binded. It is shown in Figure 4.7.

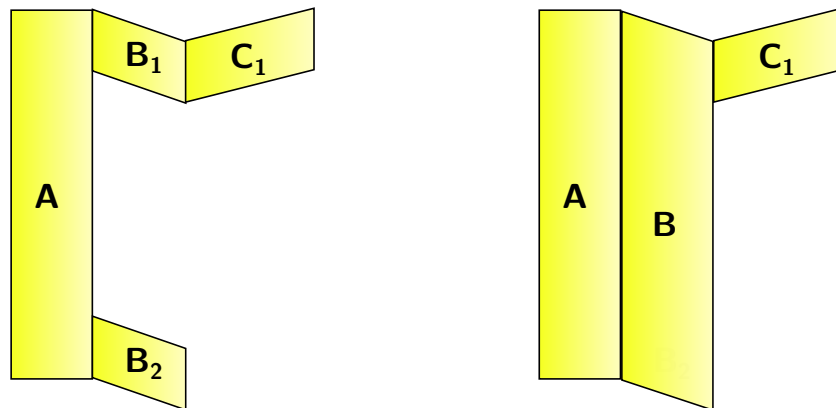


Figure 4.7: The result of binding a neighbour edge. Left: before the bind. Right: after the bind

Making an edge grow This operator can be used to handle the case where a binded edge has a neighbour which cannot be binded. It is shown in Figure 4.8.

Splitting, then binding an edge because some topological inconsistencies need to be taken into account, it is not guaranteed (and is generally not the case), the we have the same number of nodes along a stroke. Then, it should be possible to split an edge in two. This split is followed immediately by a bind, to ensure the algorithm is always advancing. This operator can be vizualized in Figure 4.9.

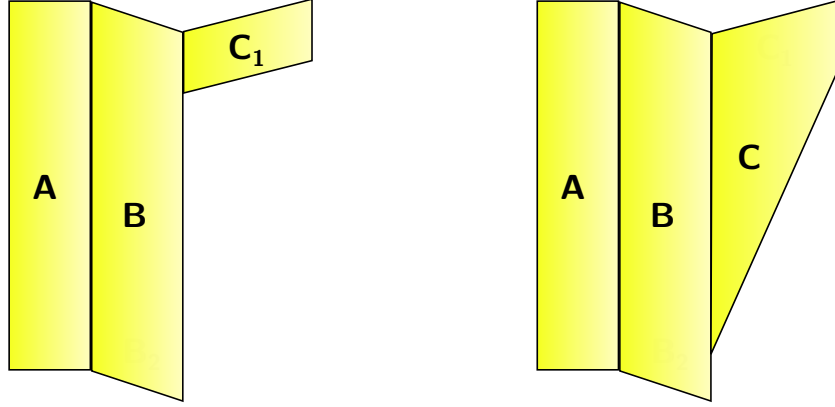


Figure 4.8: The result of making an edge grow. Left: before the growing. Right: after the growing

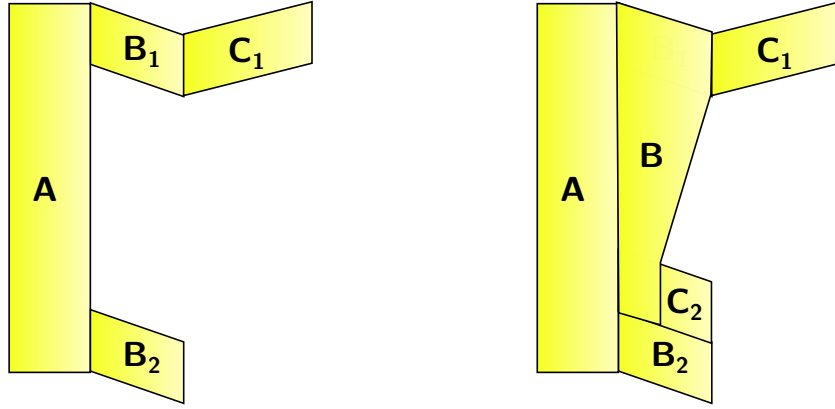


Figure 4.9: Applying a Split and Bind. Left: before the operation. Right: after the operation

Other operators should be implemented, in order to be able to represent any kind of operation (notably, here, no NodeDuplicate or EdgeDuplicate can be inserted) and then be able to create inbetweenings for a wider variety of input. However, it has not been the case due to lack of time for this research.

It is interesting to see that from the previous example, we can apply an operator Bind, that gives the result on Figure 4.10. This way, both this result and the result on Figure 4.8 are what we call a “goal animation”, eg all the edges have been binded. However, we would like to choose which one is the best. For that, we design in the next section an energy, and we would choose the one with a lower energy.

4.3 Deformation Energy

As mentioned in the introduction of this chapter, our method relies on applying iteratively the operators that has just been described. However, several choices are possible

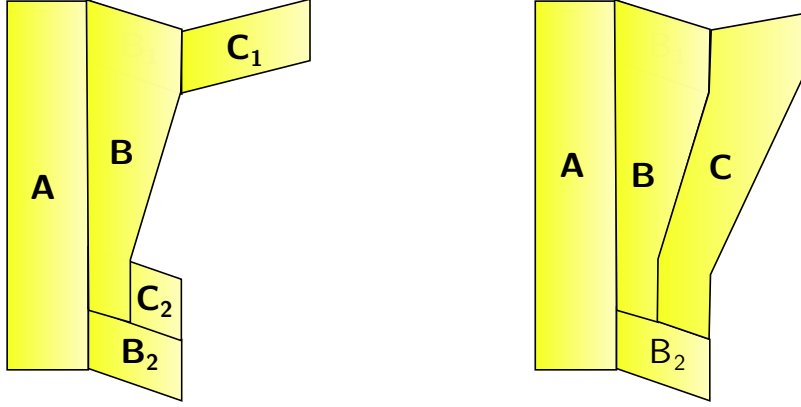


Figure 4.10: Result if we apply a last bind in our example.

at each step, and then in the end, the number of possible animations is really huge.

What we want is to select the “best one”, in a sense that will be described in this section: this is the one that would minimize a deformation energy defined over our structure.

Because of the success of *as-rigid-as-possible* approaches, it seems a good idea to get inspired from them in the design of our energy. In addition, it is clear that if the second drawing can be exactly deduced from the first one by a rigid transform, it is probably the one a user would expect.

A rigid transformation has the fundamental property that every distance (or equivalently angle) between two elements of an object is preserved all along the transformation. Let’s take $t_1 < t_2$ two different time close enough so that there is no event inbetween. We call $G_1 = \text{StrokeGraph}(t_1)$ and $G_2 = \text{StrokeGraph}(t_2)$. Our object being a stroke graph, the elements to consider are the strokes: the nodes are only here for connectivity and do not have any physical meaning, eg they have no “weight”.

Then, our physical object G_1 (as well as G_2) is made from infinitesimal 1D stroke elements, which have a position and a weight ds , corresponding to its infinitesimal length. Note: We could have also considered 2D surface elements for closed polygon, but since there exists anyway open strokes which are only in 1D, it is hopeless (at least without artificial trick) to unify them in one homogeneous cost.

Consequently, for a given stroke graph G , the distances we can measure are:

$$\forall ds \in G, \forall ds' \in G, \quad d(ds, ds') = ||\text{pos}(ds) - \text{pos}(ds')||$$

where “pos” is of course the position of the infinitesimal element ds . Since its length is infinitesimal, its position is well defined (at the contrary to the complete stroke, where we would need to make an arbitrary choice, for instance the barycenter, or the point at mid-arclength).

Any element ds_1 in G_1 is continuously transformed (because there are no event inbetween) to an element ds_2 in G_2 : it is given by the parameterization seen in the space-time representation: In fact ds is an animated infinitesimal element of an edge, and then $ds_1 = ds(t_1)$ and $ds_2 = ds(t_2)$. we call $ds = \frac{ds_1 + ds_2}{2}$ the mean length between t_1 and t_2 (we use the same notation for the infinitesimal edge and its length). Let \mathcal{S} be the set of

all these infinitesimal ds . Each $ds \in \mathcal{S}$ have a starting position $\text{pos}_1(ds)$ and an ending position $\text{pos}_2(ds)$. Then, for each $(ds, ds') \in \mathcal{S} \times \mathcal{S}$ we can define:

$$d_1(ds, ds') = ||\text{pos}_1(ds) - \text{pos}_1(ds')||$$

$$d_2(ds, ds') = ||\text{pos}_2(ds) - \text{pos}_2(ds')||$$

If the transformation were rigid, we would have:

$$\forall (ds, ds') \in \mathcal{S} \times \mathcal{S}, \quad d_1(ds, ds') = d_2(ds, ds')$$

And then our cost function should look like:

$$\iint_{\mathcal{S} \times \mathcal{S}} f(d_1(ds, ds'), d_2(ds, ds')) ds ds' \quad \text{where} \quad f(d, d) = 0.$$

We have derived the main idea of our cost function. We just need to find the right function f . We figured out that using $f(x, y) = (\log(x) - \log(y))^2$ works well in practice, for instance compared to $f(x, y) = (x - y)^2$. The reason is that it lowers the impact of deformation from strokes far from each other.

Then, by taking $t_2 = t_1 + dt$, we have the possibility to integrate over time, between any two events.

4.4 Exploring the Search Space

Finally, the only things we lack to complete this algorithm is a search strategy. Indeed, due to the combinatorial approach, the search space grow exponentially, and it is important not to explore every possibility.

Greedy A naive approach consists in using a greedy algorithm. This way, we only explore one single branch, without going back. This approach does not work at all, because there is in almost all cases one moment where the algorithm would do a bad choice, and then there is no way to recover from this mistake.

Other approaches are Best-First strategies, eg based on a priority queue. For each new animation to explore, we attach a priority on it, and then explore it only when this priority is the highest among the others. Depending on how we define this priority, it can have a lot of different behaviours.

Dijkstra an interesting thing is that in the computation of our energy, we do compute only the energy of binded edges, and then the energy is increasing little by little until we find a complete animation, that we expect not having a too high energy. Since the energy is strictly increasing, it can be interpreted as a distance in a graph, the nodes being the animations, and the edges the operators. Then, if we use as priority the distance itself, we have an implementation of Dijkstra, which gives us the guarantee that the first goal node found (eg without unbinded edges) is actually the one with the lowest energy. However, it is generally too slow in practice.

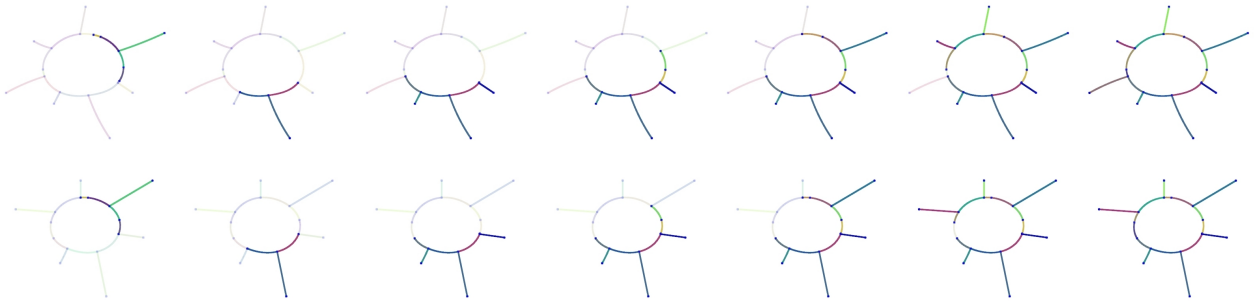


Figure 4.11: Some of the 13 steps that our algorithm Closest First explores before finding the solution on this example.

A* We can also try an implementation of A*. What we need for that is an heuristic giving an estimated distance to a goal node. and in fact, we do have a good heuristic for that. Indeed, if the total length of edges in the input is L , and that we have already binded a length l of edges, with a current energy of E , then the energy of the goal node is probably close to $E(\frac{L}{l})^2$, due to the quadratic complexity of the energy. This approach finds more often that Dijkstra a solution in a reasonable amount of time (eg less than a minute), but still doesn't always manage to find a solution.

Closest First Then, we finally use something hybrid between a Greedy algorithm and A*. In fact, it is an A* but by using –purely– the estimated remaining distance to the goal node, instead of the sum of the current distance plus the estimated remaining distance. This comes from the fact that the only thing we want is to find a goal node, instead of finding the shortest path to a goal node. In fact, in our search space, all paths to a node have the same distance, because we interpreted as distance the energy of the animation, and then does not depend on how we get there.

This last algorithm performs way better than the others. For instance, in the example Figure 4.11, it has found a correct solution in only 13 steps, whereas A* never managed to find a solution. Two videos showing these search behaviours are available as supplemental material at the webpage www.dalboris.fr/masterthesis.htm. A* does explore too many potential combinations between “Bind” or “Split and Bind”. In the contrary, Closest First tends to keep going in the same direction since it decreases the estimated remaining distance to the final animation. It would change its path only if the energy increases suddenly and then compensate the fact that there is only very few edges yet to match. Fortunately, this is exactly what happens generally when the algorithm goes into a wrong path.

5 Experimental Results

In this section are presented the different results that have been obtained during this Master Thesis. Some of them are best seen with a video, those videos can be downloaded from the address www.dalboris.fr/masterthesis.htm.

5.1 Implementation and Interface

Mostly all the different aspects discussed in the previous sections has been implemented to test our algorithms on real examples. The first step was to create an interface for drawing our input stroke graphs. It is really intuitive since the only possible action is to draw with the left clic. The stroke is automatically fitted into a clothoid, the intersections with existing clothoids are computed, and T-junctions are cleaned. A video demonstration of this drawing interface is available.

An important aspect of the interface is the possibility to select directly edges in order to specify manually some correspondences, and then guide the algorithm when the fully automatic method does not work. Edges can also be selected by navigating using the half-edge data structure. A video demonstration of this is also available.

Concerning the data structure, the abstract definition of the Space-Time Stroke Graph has been implemented, but not all the topological events, due to lack of time. This way, the only topological events we can use for now are Creation, SplitNode and SplitEdge (as well as their inverse). The operators relative to them has been implemented, and then it is possible to manipulate conveniently the data structure, for instance adding a Split-Edge event by applying the operators $\text{SplitEdgeAfter}(t, \text{edge})$ or $\text{SplitEdgeBefore}(\text{edge}, t)$. For now, this can be done only programmatically, but it would be relatively easy with the current interface to do it graphically, since there is already a timeline to select the appropriate time: we would just need to clic on the edge to perform the operation.

Lastly, the computation of the energy has been implemented as well as a convenient way to navigate into our search space, either manually or by applying algorithms such as Greedy, Dijkstra, A* or Closest First. By default, for all our following example here, it is Closest First which is used, unless specified differently.

5.2 Results and Validation

In order to validate our method, we have tested it on different pairs of drawings. Examples of succesful inbetweening are presented in Figure 5.1, 5.2 and 5.3. The videos of the animations can be downloaded at www.dalboris.fr/masterthesis.htm. In the most simple cases, no user interaction is necessary. However, in some difficult parts, the user needs to indicate manually the good correspondence by a simple drag-and-drop.

It is important to note that in all cases that have been tested, the energy of the manually corrected solution is the lowest among all the other visited solutions. This tends to validate the definition of our energy. For instance, the first result provided by

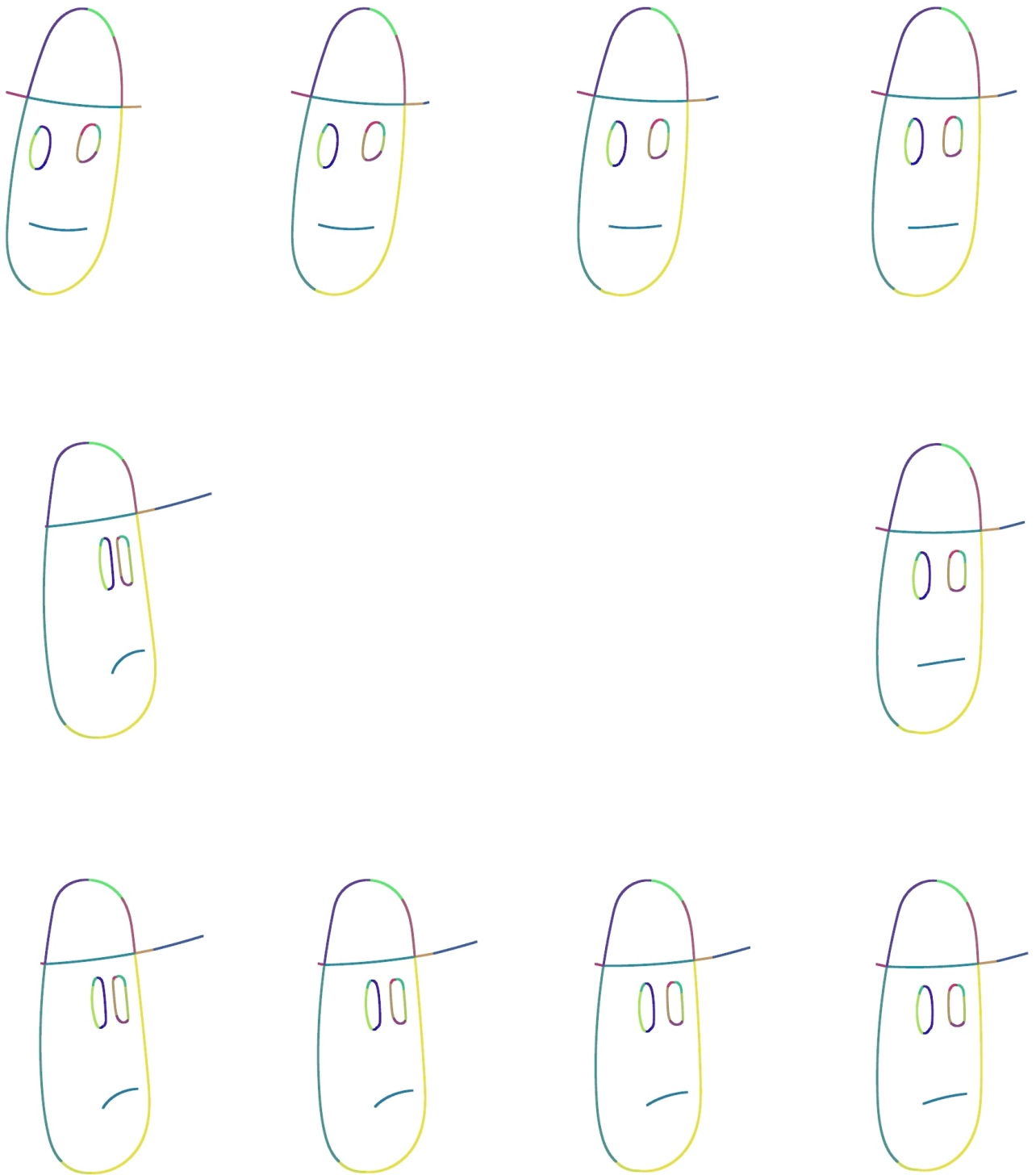


Figure 5.1: A successful result of our algorithm. The animation should be read clockwise from the top-left corner. One user interaction was necessary.

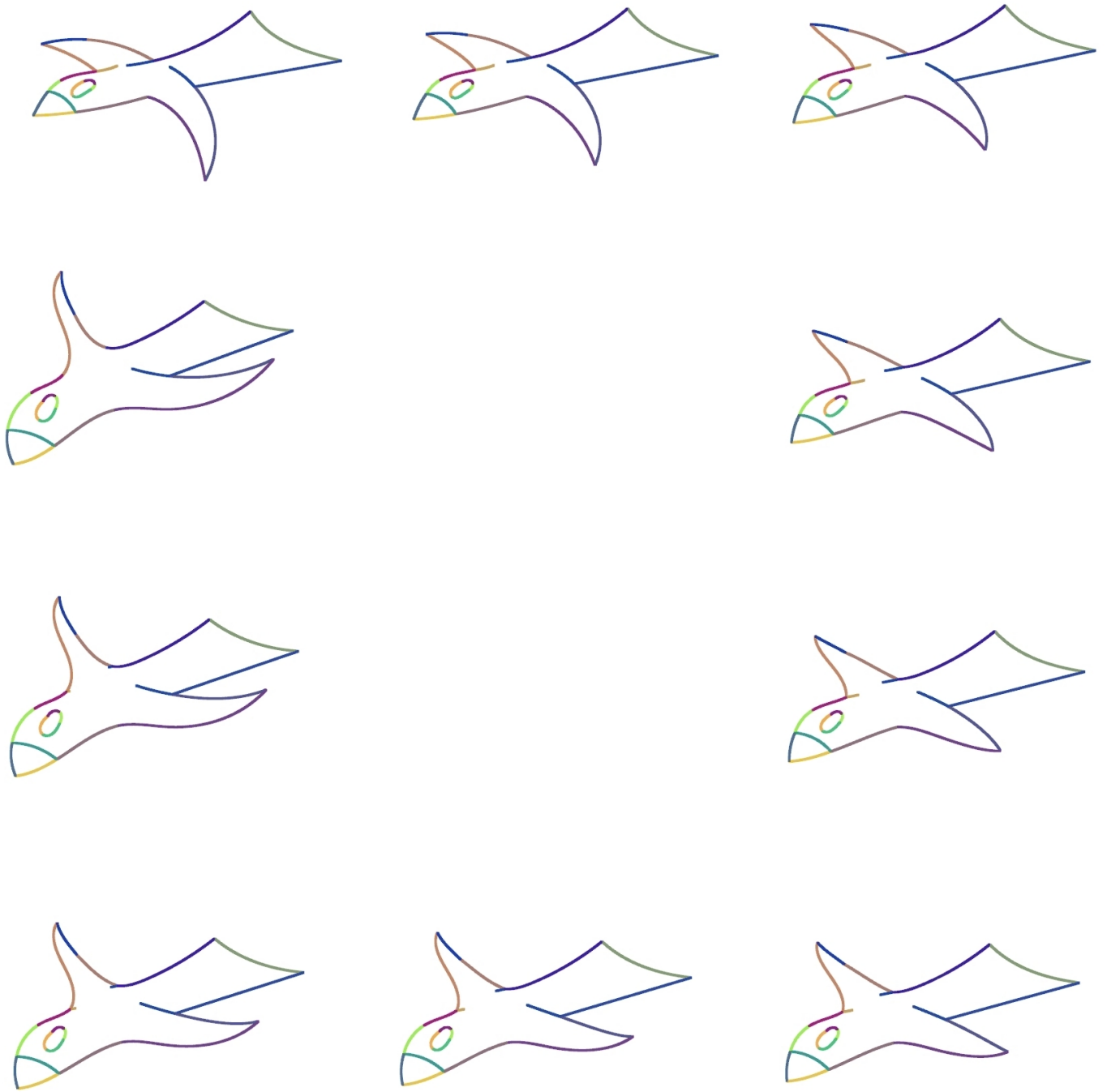


Figure 5.2: A successful result of our algorithm. The animation should be read clockwise from the top-left corner. No user interaction was necessary, it has been fully automatically inbetweened.

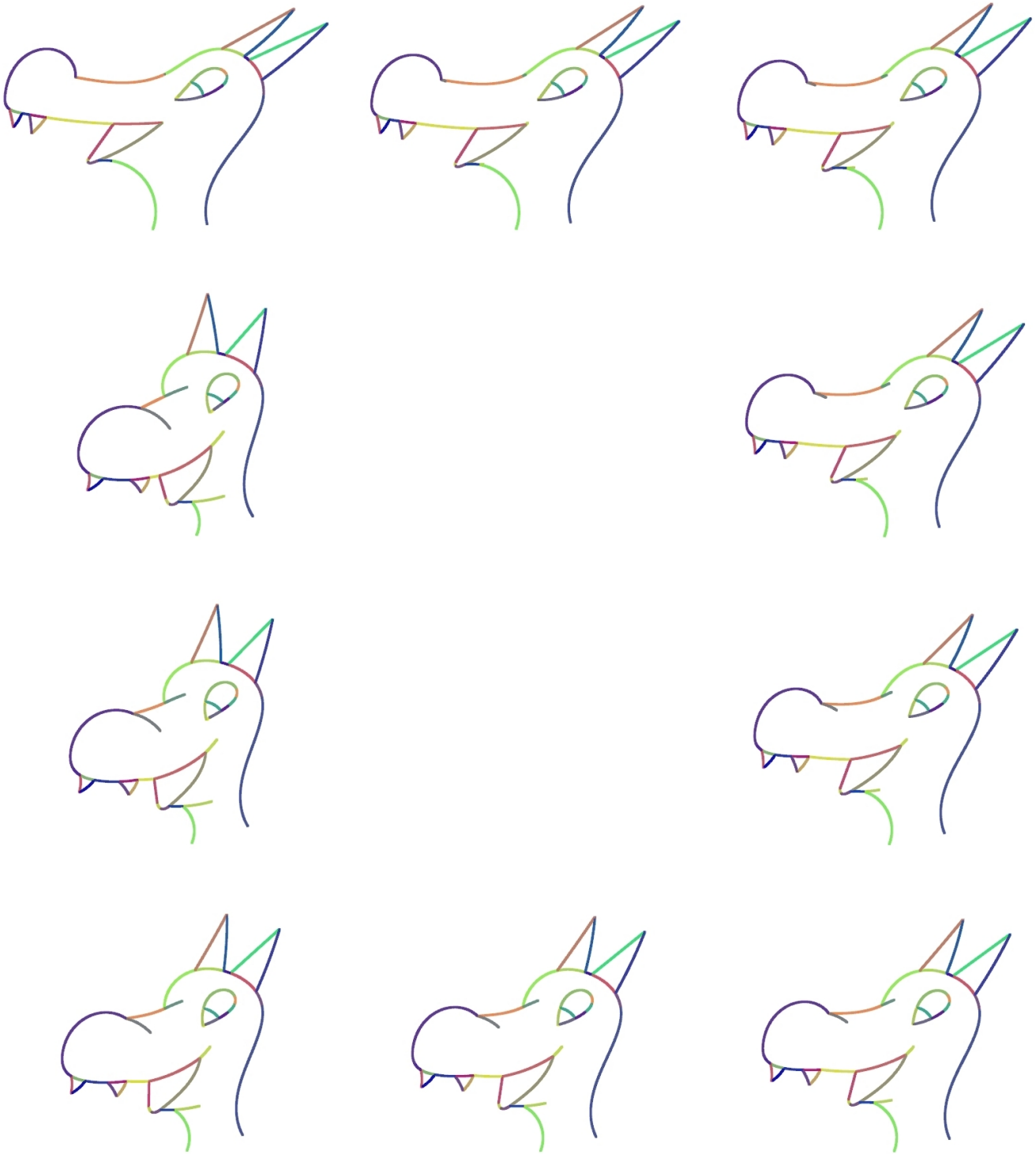


Figure 5.3: A successful result of our algorithm. The animation should be read clockwise from the top-left corner. Two user interactions were necessary.

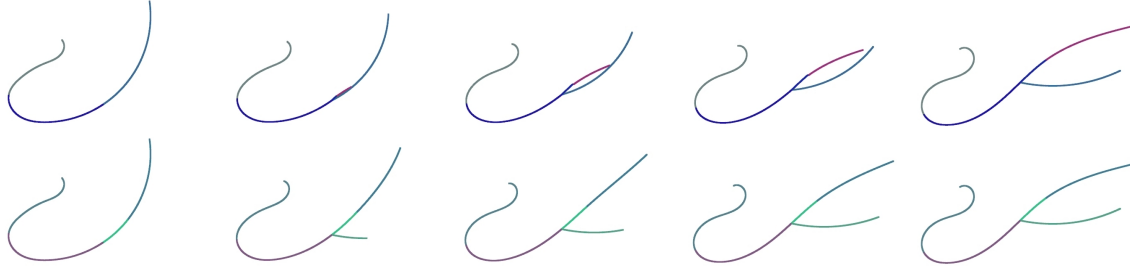


Figure 5.4: The top row is an animation whose energy is 120,203, while the bottom row is an animation whose energy is 36,734.



Figure 5.5: A naive approach to inbetween edges not automatically matched by the current state of art would be to make them suddenly disappearing and appearing.

the Closest First algorithm on the example Figure 5.4 was the one on the top row, whose energy is approximatively 120,000. By executing Dijkstra instead, which can be done in about one second on this very simple example, we are guaranteed to find the global minimum (restricted by our choice of operators) that can be seen on the bottom row, whose energy is around 40,000. In general, when Closest First finds a first goal node, it is a good idea to continue the research through the remaining priority queue, in order to find new solutions with a lower energy. On this example, the optimal solution was the third goal node discovered, the energy were 120,203, then 66,380 and finally 36,734.

A very good point of our approach is that it performs automatic inbetweening, or with very few user interaction, in the case of topological inconsistencies and non tight inbetweens, as the contrary to the state of the art [51] which is restricted to this case. For instance, their algorithm would have stopped in every example in Figure 5.1, 5.2 and 5.3. whenever a topological inconsistency occurs, or if the edges are too different, which is not the case of our method. A naive solution to handle this problem automatically would be to make suddenly appear and disappear the edges not automatically mathed. The edges that would not be matched automatically are presented in Figure 5.5, and a video of the naive solution can be downloaded, which should be compared to the video using our approach. Our clean interpolation of parts with topological events, even if it is a simple 2D linear interpolation, induce a much stronger perception of what would be the 3D shape.

In addition, it works exactly as well as [51] in the cases they specifically target: when there is no topological inconsistency. We just need to disable all operators but Bind, and

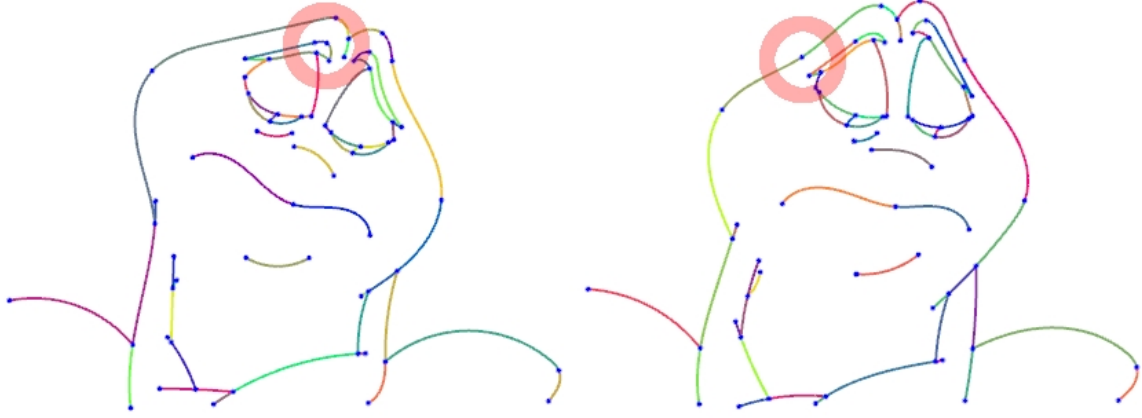


Figure 5.6: Segmenting two stroke graphs topologically consistent can result in two stroke graphs topologically inconsistent. Each node highlighted in red do not exist in the other stroke graph. This example has been rotoscoped from two keyframes of *The Princess and the Frog* (Walt Disney Animation Studios), separated by 13 frames (at 24fps, eg approximatively half a second).

not to perform the segmentation beforehand. Then, there is only one decision at each time, the search space becomes linear and all our algorithms are equivalent, which makes it as efficient as [51] (in fact, probably much less because our structure is heavier, but it is equivalent in terms of number of steps). In fact, it works even better thanks to our preliminary pairwise likelihood: no manual interaction at all is necessary to match all the strokes correctly in about half a second (whereas the user should give a manual seed in their method). The reason why we can avoid user interaction is not only because we have a trustful automatic seed, but also because we have our energy, which will very quickly grow exponentially in case of a wrong seed, and the next seed will be chosen instead, after only one or two steps.

But one can argue that in our targeted application (2.5D rotoscoping for non-skilled user), the algorithm does not know beforehand that there is no topological inconsistency. Then, we cannot disable the segmentation, and this step could in fact create topological inconsistencies as can be seen in Figure 5.6. This happens whenever a local minimum of curvatures exist in only one of the two drawings. However, running our algorithm with its classical parameters perform fully automatically the good inbetweening, creating the two appropriate SplitEdge events. The computation took in this case four seconds in my personal laptop (Intel Core i7), and then is “only” height times slower than the approach of [51]. This may seem a lot, but it has to be compared to combinatorial size of the search space: at each step, because we can bind edges, split edges, or make them grow, with all their possible combinations where at T-junctions, there are about 5 possibilities at each step, and then all in all 5^n configurations could be potentially examined (where n is max number of edges, $n = 61$ in our example). A video of the inbetweening can be downloaded, you can observe the two edges splitted or unsplit at the beginning and the end of the animation.

For a more indeep validation of the method, something that should be investigated is to ask a professional animator to draw the inbetweens for the given examples, and then

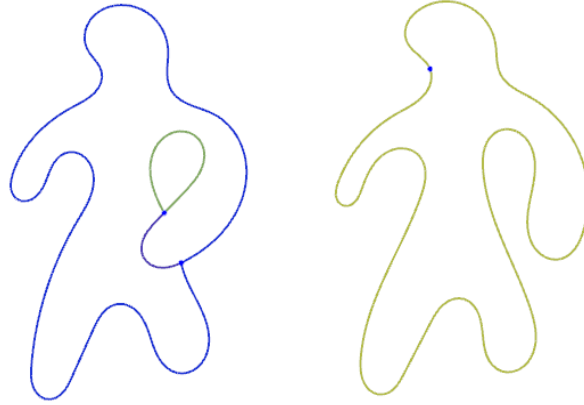


Figure 5.7: An example of keyframe pair that our implementation cannot inbetween yet.

compare the results to our automatic algorithm.

5.3 Limitations and Future Work

Naturally, even if it is yet very promising, there are currently a lot of limitations that need to be tackled in the future.

5.3.1 Limitations

The first obvious limitation is that not all topological events are implemented, and then currently not all types of animation can be inbetweened. For instance, there is no way our current implementation can handle the input pair of drawing in Figure 5.7. Our algorithm get stuck and do not manage to converge, as can be seen in the video available at www.dalboris.fr/masterthesis.htm. Typically, our approach cannot work well in situation where layering is involved: in this case, a good approach should be first to separate the animation into several layers, and then to use our algorithm. However, this example is interesting because no simple layering can be done (unless cutting the character in two): what should be investigated is to add the concept of faces in our approach (can already be defined by the half-edges), and specify a depth to the edges or the points of the face. For this reasons, Our implementation cannot currently inbetween complex animation such as the puppy in the introduction, or the princess from [13], since this kind of events occur at several places.

Another implicit limitation of this approach is that it works purely in 2D space, even if it targets topological inconsistencies which are often caused by the underlying 3D world. The idea behind this approach is that in 2D animation, the stylization is what matters most, rather than an accurate 3D consistent motion. Our definition of energy was designed so that the matching favors an as rigid as possible interpolation, and then the result is probably not too far from an actual 3D motion. However, we have yet no guarantee of this behaviour, and a counter-example can be seen in Figure 5.8, which can also be downloaded in video. This inbetweening has been obtained by running

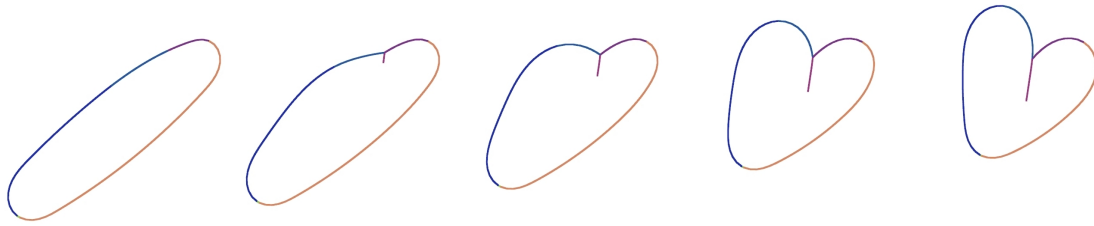


Figure 5.8: An example where minimizing the energy does not give the correct 3D expected behaviour.

the Dijkstra algorithm, and then is a global minimum of our energy. We can see that the 3D correct interpretation (a cylindric shape which is bending) is completely missed, providing instead a motion with less 2D deformations, but not adapted to this case.

Also, the combinatorial nature of the space of research is a serious limitation, since – except on simple study cases – no exhaustive search can be done to find a global minimum. Even though our Closest First approach seems a good start, I have the belief that there exists methods drastically better. For instance, currently, the same operators are tried an exponentially number of times in a different combination when we are stuck to a barrier of energy it does not manage to pass. It seems that a better approach would be in these cases that the algorithm tries an operation never done yet, even if at the moment it increases too much the energy. Maybe this could be achieved by a randomized algorithm, which explores the configurations with a probability determined both by our energy, but also depending on the number of time we have already tried that operator.

5.3.2 Future Work

Then, the most important future work should be done in the search algorithm. Having a more precise idea on the nature of the search space is necessary, to be able to design a more adapted algorithm, with possibly proofs of convergence toward a global minimum, guarantee that we do not have using the Closest First algorithm of our approach, and that we have using Dijkstra but whose computational time prohibits for any use in typical inputs.

Of course, the user interface should be improved in order to have more interaction with the inbetweening algorithm. Also, having the possibility to draw a whole animation: not only two keyframes. As an obvious improvement, it is also necessary to use another interpolation between matched strokes than the current linear interpolation. Using logarithmic spirals as a first candidate is probably a good idea, if we refer to [51].

In addition, a lot of information could now be added to the data structure and algorithm to make it more task oriented. The first application we want to target is the use of a video: how the information extracted could help us to predict potential topological events and anticipate them?

Finally, a generalisation of our Space-Time Topology could be done: it is probably possible to add depth information, so that we can explain 2.1D layering event in a simpler way that actually modifying the structure to handle the event. Also, we didn't consider at all faces in our approach, while they convey a lot of information. The reason is that faces not always exist, and then an energy using edges seemed more natural. But inserting faces

in our stroke graphs seems a very good idea, since it would be interpreted as volume by adding time, and we could design an actual as rigid as possible geometric interpolation.

Finally, the “Space” component was in this research a 2D space. However, it seems that there is no reason it could not be extended to a 3D Space-Time Topology: eg a 4D space. This kind of space has already been studied by Schmid et al. in 2010 [37], because they he needed to track polygons in time to create stylized motion effects using shaders.

6 Conclusion

Vectorial automatic inbetweening in the case of topological inconsistencies was clearly an open problem: no previous attempts have been realized successfully. However, it is necessary to perform 2.5D rotoscoping, and then this is what has been targeted during this Master Thesis, focusing in developping a new approach for this problem, based on an innovative space-time approach.

The solution we proposed already works in a variety of situations, and even if it still has some limitations, it is a clear advance on the state of the art. In addition, it opens a lot of doors for future research, since at each step of the algorithm there are rooms for improvements. Our solution works thanks to the combination of several contributions together:

- Using clothoids to perform a good segmentation.
- A new statistical and perceptual method to compute the likelihood that two strokes are in correspondence, in the context of inaccurate inbetweening.
- A data structure to be able to represent the spatio-temporal topology of a vectorial 2D animation with topological events, which as never been done before.
- A suitable measure of non-rigid deformation over this space-time structure.
- A Best-First algorithm combining A* and Greedy ideas, well adapted to our problem.

Moreover, our data structure could be used for a wide range of applications other than inbetweening. Of course, it can be considered for manual 2D animation, through intuitive user interaction. But it can also be used for multi-scale representation of an object, if we replace the time by the viewing distance. One could also see biological applications, to model for instance how plants are growing. To summarize, it can be used for any application that needs a continuous representation of topological events.

- [1] Aseem Agarwala, Aaron Hertzmann, David H. Salesin, and Steven M. Seitz. Keyframe-based tracking for rotoscoping and animation. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 584–591, New York, NY, USA, 2004. ACM.
- [2] Marc Alexa, Daniel Cohen-Or, and David Levin. As-rigid-as-possible shape interpolation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 157–164, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.
- [3] Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. Ilovesketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, pages 151–160, New York, NY, USA, 2008. ACM.
- [4] Ilya Baran, Jaakko Lehtinen, and Jovan Popović. Sketching clothoid splines using shortest paths. *Computer Graphics Forum*, 29(2):655–664, 2010.
- [5] William Baxter, Pascal Barla, and Ken-Ichi Anjyo. Compatible Embedding for 2D Shape Animation. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):867–879, 2009.
- [6] William Baxter, Pascal Barla, and Ken-Ichi Anjyo. N-way morphing for 2D Animation. *Computer Animation and Virtual Worlds (Proceedings of CASA)*, 20(2), 2009.
- [7] Adrien Bernhardt, Adeline Pihuit, Marie-Paule Cani, and Loïc Barthe. Matisse: Painting 2D regions for modeling free-form shapes. In Christine Alvarado and Marie-Paule Cani, editors, *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling, SBIM 2008, June, 2008*, pages 57–64, Annecy, France, June 2008.
- [8] P. Blair. *Cartoon animation*. How to Draw and Paint Series. W. Foster Pub., 1994.
- [9] B. Bratt. *Rotoscoping: Techniques and Tools for the Aspiring Artist*. Focal Press. Focal Press, 2011.
- [10] Christoph Bregler, Lorie Loeb, Erika Chuang, and Hrishi Deshpande. Turning to the masters: motion capturing cartoons. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 399–407, New York, NY, USA, 2002. ACM.
- [11] Bert Buchholz, Noura Faraj, Sylvain Paris, Elmar Eisemann, and Tamy Boubekeur. Spatio-temporal analysis for parameterizing animated lines. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 85–92, New York, NY, USA, 2011. ACM.
- [12] N. Burtnyk and M. Wein. Interactive skeleton techniques for enhancing motion dynamics in key frame animation. *Commun. ACM*, 19(10):564–569, October 1976.

- [13] Edwin Catmull. The problems of computer-assisted animation. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 348–353, New York, NY, USA, 1978. ACM.
- [14] James Davis, Maneesh Agrawala, Erika Chuang, Zoran Popović, and David Salesin. A sketching interface for articulated figure animation. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 320–328, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.
- [15] Christina N. de Juan and Bobby Bodenheimer. Re-using traditional animation: methods for semi-automatic segmentation and inbetweening. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '06, pages 223–232, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.
- [16] Jean-Daniel Fekete, Érick Bizouarn, Éric Cournarie, Thierry Galas, and Frédéric Taillefer. Tictactoon: a paperless system for professional 2d animation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 79–90, New York, NY, USA, 1995. ACM.
- [17] Arcot Sowmya Fengqi An, XiongCai Cai. Automatic 2.5d cartoon modelling. *IVCNZ 2011*, 0(0):0–0, 2011.
- [18] Fabian Di Fiore, Philip Schaeken, Koen Elens, and Frank Van Reeth. Automatic inbetweening in computer assisted animation by exploiting 2.5d modelling techniques. In *In Proceedings of Computer Animation (CA2001)*, page pages, 2001.
- [19] Hongbo Fu, Chiew lan Tai, and Oscar Kin chung Au. Morphing with laplacian coordinates and spatial-temporal texture. In *Proceedings of Pacific Graphics 2005*, pages 100–102, 2005.
- [20] Yotam Gingold, Takeo Igarashi, and Denis Zorin. Structured annotations for 2d-to-3d modeling. In *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH Asia '09, pages 148:1–148:9, New York, NY, USA, 2009. ACM.
- [21] Samuel Greengard. Living in a digital world. *Communications of the ACM*, 54(10):17–19, October 2011.
- [22] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1134–1141, New York, NY, USA, 2005. ACM.
- [23] J.J. Koenderink. *Solid shape*. Artificial intelligence. MIT Press, 1990.
- [24] Alexander Kort. Computer aided inbetweening. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, NPAR '02, pages 125–132, New York, NY, USA, 2002. ACM.
- [25] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 35–44, New York, NY, USA, 1987. ACM.

- [26] Marc Levoy. A color animation system: based on the multiplane technique. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 65–71, New York, NY, USA, 1977. ACM.
- [27] M. Leyton. *The Structure of Paintings*. Springer, 2006.
- [28] Ji Lu, Hock Soon Seah, and Feng Tian. Computer-assisted cel animation: post-processing after inbetweening. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 13–ff, New York, NY, USA, 2003. ACM.
- [29] Dhruv Mahajan, Fu-Chung Huang, Wojciech Matusik, Ravi Ramamoorthi, and Peter Belhumeur. Moving gradients: A path-based method for plausible image interpolation. *ACM Transactions on Graphics (SIGGRAPH 09)*, 28(3), July 2009.
- [30] Fleischer Max. Method of producing moving-picture cartoons. *Patent, US 1242674 (A)*, October 1917.
- [31] Takeo Miura, Junzo Iwata, and Junji Tsuda. An application of hybrid curve generation: cartoon animation by electronic computers. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 141–148, New York, NY, USA, 1967. ACM.
- [32] M. Nitzberg, D. Mumford, and T. Shiota. *Filtering, Segmentation, and Depth*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
- [33] G. Noris, D. Sýkora, S. Coros, B. Whited, M. Simmons, A. Hornung, M. Gross, and R. W. Sumner. Temporal noise control for sketchy animation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 93–98, New York, NY, USA, 2011. ACM.
- [34] William T. Reeves. Inbetweening for computer animation utilizing moving point constraints. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '81, pages 263–269, New York, NY, USA, 1981. ACM.
- [35] Alec Rivers, Takeo Igarashi, and Frédo Durand. 2.5d cartoon models. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 59:1–59:7, New York, NY, USA, 2010. ACM.
- [36] Damien Rohmer, Stefanie Hahmann, and Marie-Paule Cani. Exact volume preserving skinning with shape control. In *Symposium on Computer Animation, SCA '09: "Moving Research", August, 2009*, pages 83–92, New Orleans, Etats-Unis, August 2009. Eurographics/ACM SIGGRAPH, ACM.
- [37] Johannes Schmid, Robert W. Sumner, Huw Bowles, and Markus Gross. Programmable motion effects. *ACM Trans. Graph.*, 29:57:1–57:9, July 2010.
- [38] T. B. Sebastian, P. N. Klein, and B. B. Kimia. On aligning curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(1):116–125, 2003.
- [39] Thomas W. Sederberg, Peisheng Gao, Guojin Wang, and Hong Mu. 2-d shape blending: an intrinsic solution to the vertex path problem. In *Proceedings of the 20th*

- annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 15–18, New York, NY, USA, 1993. ACM.
- [40] Dana Sharon and Michiel van de Panne. Constellation Models for Sketch Recognition. In Thomas Stahovich and Mario C. Sousa, editors, *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 19–26. Eurographics Association, 2006.
 - [41] Justin Solomon, Mirela Ben-Chen, Adrian Butscher, and Leonidas Guibas. As-killing-as-possible vector fields for planar deformation. *Computer Graphics Forum*, 30(5):1543–1552, 2011.
 - [42] Sébastien Sorlin. *Mesurer la similarité de graphes*. Thèse de doctorat en informatique, Université Claude Bernard Lyon I, November 2006. Thèse encadrée par Christine Solnon et Mohand-Saïd Hacid Ecole doctorale : EDIIS.
 - [43] Sébastien Sorlin and Christine Solnon. A global constraint for graph isomorphism problems. In *The 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004)*, Lecture Notes in Computer Science, pages 287–301. Springer Verlag, April 2004.
 - [44] Sébastien Sorlin and Christine Solnon. Reactive tabu search for measuring graph similarity. In Mario Vento Luc Brun, editor, *5th IAPR-TC-15 workshop on Graph-based Representations in Pattern Recognition*, pages 172–182. Springer-Verlag, April 2005.
 - [45] Sébastien Sorlin and Christine Solnon. A parametric filtering algorithm for the graph isomorphism problem. *journal of constraints*, 13(4):518–537, December 2008.
 - [46] Daniel Sýkora, Mirela Ben-Chen, Martin Čadík, Brian Whited, and Maryann Simmons. Textoons: practical texture mapping for hand-drawn cartoon animations. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 75–84, New York, NY, USA, 2011. ACM.
 - [47] Daniel Sýkora, John Dingliana, and Steven Collins. As-rigid-as-possible image registration for hand-drawn cartoon animations. In *Proceedings of International Symposium on Non-photorealistic Animation and Rendering*, pages 25–33, 2009.
 - [48] F. Thomas and O. Johnston. *Disney Animation: The Illusion of Life*. Abbeville Press, 1987.
 - [49] S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(5):695–703, September 1988.
 - [50] Anton van den Hengel, Anthony Dick, Thorsten Thormählen, Ben Ward, and Philip H. S. Torr. Videotrace: rapid interactive scene modelling from video. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.
 - [51] B. Whited, G. Noris, M. Simmons, R. Sumner, M. Gross, and J. Rossignac. Betweenit: An interactive tool for tight inbetweening. *Comput. Graphics Forum (Proc. Eurographics)*, 29(2):605–614, 2010.

- [52] R. Williams. *The Animator's Survival Kit—Revised Edition: A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Internet Animators*. Faber and Faber, 2009.
- [53] Jun Yu, Wei Bian, Mingli Song, Jun Cheng, and Dacheng Tao. Graph based transductive learning for cartoon correspondence construction. *Neurocomputing*, 79(0):105 – 114, 2012.
- [54] Jun Yu, Dongquan Liu, Dacheng Tao, and Hock Soon Seah. Complex object correspondence construction in two-dimensional animation. *Image Processing, IEEE Transactions on*, 20(11):3257 – 3269, nov. 2011.
- [55] Stéphane Zampelli, Yves Deville, Christine Solnon, Sébastien Sorlin, and Pierre Dupont. Filtering for Subgraph Isomorphism. In *13th International Conference on Principles and Practice of Constraint Programming (CP'2007)*, LNCS, pages 728–742. Springer, September 2007.
- [56] Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.