# View Morphing for 2D Animation

Cynthia Jia
Stanford University
CS231A Winter 2021
cynthiaj@stanford.edu

## Abstract

*Traditional 2D animation requires manual interpolation between sparse keyframes to create intermediate images called "inbetweens." I explore the application of view morphing to create these in-betweens through the process described in Seitz and Dyer 1996 [8].*

## 1. Introduction

Traditional 2D animation relies on the artist's ability to draw keyframes, or sketches of major snapshots in time. Between these keyframes, the artist must interpolate by drawing intermediate frames, or "inbetweens," as shown in Figure 1, to fill out the range of motion. I have found that if I want to animate something simple like a rotating object, it can become frustrating and time-consuming to draw these intermediate panels.

I explore the use of view morphing to create the intermediate views between simple hand-drawn keyframes, hoping to replace the laborious task of creating inbetweens by hand with a more streamlined process, with the ultimate goal of creating visually pleasing animated rotations of line-drawn objects.

I experiment with pairs of personally hand-drawn keyframes, in the style of a simple black and white line drawings. I apply Seitz and Dyer's 1996 three-step view morphing process to these images, and then visually evaluate the results.

## 2. Background and Related Work

### 2.1. Computers and Automatic 2D Inbetweens

Automatic interpolation for animation has been a topic of interest for years, especially as the animated film industry demands high quality and smooth animations in limited production time.

Catmull describes the challenges that arise with solving this problem [4]:

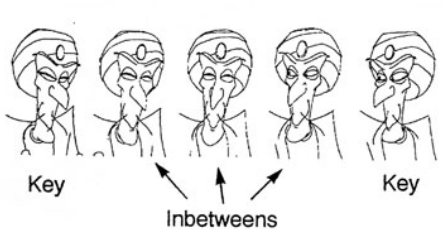


Figure 1. An example of inbetweens used for a 2D character.

> The principle difficulty is that the animator's drawings are really two dimensional projections of three dimensional characters as visualized in the animator's head, hence information is lost...Efforts to overcome this by drawing skeletons or increasing the number of overlays require more manual intervention thereby offsetting the gains of using the computer.

Like Catmull mentions, Carvalho et al propose the use of skeleton-like guide curves drawn by artists to automatically interpolate between poses of a figure [3].

Miura et al and Sun et al avoid extra manually-drawn skeletons by modelling characters as spline curves and interpolating between them automatically [6] [9]. However, this limits what animators can draw, since for good performance everything in their scene would have to be restricted to a spline curve.

In a different vein of thinking, Xing et al present an interface that uses past frames to predict a future shape in the next frame, using a global similarity based on fuzzy correspondences [10].

Products such as CACANi also exist, offering automatic inbetweens as a service [1]. The technical implementation of the product is unclear, but it aims to tackle the same problem I am approaching here.

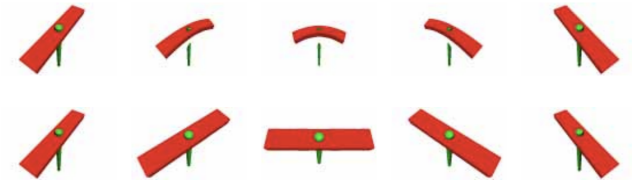

Figure 2. Simple image morphing (top) may distort straight lines, while view morphing (bottom) preserves them, yielding a more accurate result.

### 2.2. View Morphing for Automatic 2D Inbetweens

While many papers utilize features like skeletons, splines, and other tactics, none of them mention view morphing. I am intrigued by the way that Seitz and Dyer compare simple image morphing with view morphing. They note that plain image morphing, without considering camera or view geometry, may warp straight lines, while view morphing preserves these lines [8]. You can see this in an image from their paper in Figure 2. Furthermore, the way novel views are generated between two initial views reminds me of the way in which inbetweens are generated between keyframes.

While correspondences must be selected for view morphing, and this could act as an additional pain point like Catmull described above, I wanted to explore the avenue, which could potentially output interesting results.

Seitz and Dyer have written a series of works on the topic of view morphing, including an earlier paper focused on rectification (*Physically-Valid View Synthesis by Image Interpolation*) and two later works which are more focused on the entire view morphing process (*Toward Image-Based Scene Representation Using View Morphing* and *View Morphing*). All of their example images include photographs from real life, or physically realistic 3D models, so I am curious as to how the process will treat 2D line drawings which may not follow 3D physical rules.

Ji et al build upon Seitz and Dyer's work by applying CNNs for dense correspondences, creating more realistic results [5]. However, keeping time frame in mind, I aim to focus on the basic method described in Seitz and Dyer's 1996 paper *View Morphing*.

## 3. Technical Approach

I will follow the view morphing process described in Seitz and Dyer 1996 [8], with the following steps:

1. rectify or pre-warp corresponding images

2. interpolate or morph between rectified images to get an intermediate image

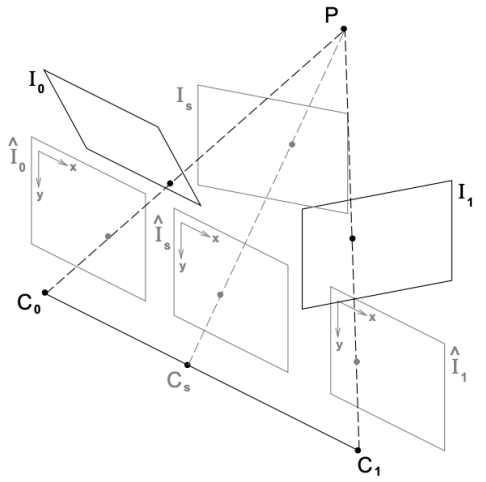


Figure 3. View Morphing in Three Steps. (1) Original images $I_0$ and $I_1$ are prewarped to form parallel views $\hat{I}_0$ and $\hat{I}_1$. (2) $\hat{I}_s$ is produced by morphing (interpolating) the prewarped images. (3) $\hat{I}_s$ is postwarped to form $I_s$

3. post-warp the intermediate image to get our final result.

To visualize this process, see figure 3 taken from Seitz and Dyer's paper [8].

Let us refer to our two keyframe images as $I_0$ and $I_1$.

### 3.1. Pre-warping

Pre-warping refers to the rectification of corresponding images, where the resulting image planes are parallel. To rectify our images, we require two homographies $H_0$ and $H_1$ that will transform points in $I_0$ and $I_1$ to points on parallel image planes.

Since 2D drawings do not exist in real geometric space, we do not know any of the camera geometry, or the intrinsic or extrinsic camera parameters. As a result, we need to find corresponding image points and use the eight-point algorithm in order to pre-warp the images.

#### 3.1.1 Finding Correspondences

Point correspondences are found manually through an OpenCV program, where the user can click points on an image to mark them with a crosshair and number, and the respective coordinate is printed for saving, as seen in Figure 4. The code for this program can be found in `get_coords.py`, located in the GitHub linked in the appendix.

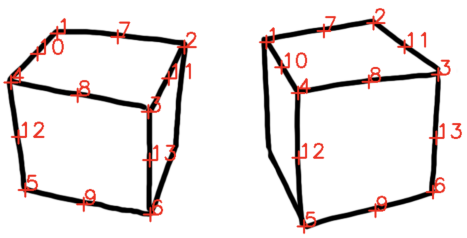

Figure 4. Simple drawings with clicks marked with OpenCV, whose coordinates were printed to the terminal.

#### 3.1.2 Finding $H_0$ and $H_1$

First we find the fundamental matrix F using the eight point algorithm on the corresponding image coordinates. I use a normalized eight point algorithm which first shifts the origin of correspondences to their centroid, and scales them so that they have similar magnitudes.

To find $H_0$ and $H_1$ which rectify the corresponding images, we can then use F to compute epipolar lines for each correspondence, and find the epipoles. Once we have the epipoles, we can create some transformations that combine translation, rotation, and a transform to bring the epipole to infinity.

A more detailed explanation of this process is described by Seitz and Dyer in the appendix of `View Morphing` [8].

### 3.2. Morph

To find an intermediate image $I_s$, we need to interpolate between rectified points. To do so, I use Beier-Neely field morphing as this is the method used by Seitz and Dyer [8]. Beier-Neely field morphing differs from simple cross-dissolving of image values in that it uses interpolated feature lines to first morph both images to an intermediate shape, before cross-dissolving [2]. An illustration of this can be seen in Figure 5

#### 3.2.1 Beier-Neely morphing

We will be interpolating or transitioning from one rectified image to another, based on a interpolation ratio $t$. We will use corresponding feature lines, which are simply connections between correspondence points in our rectified images. To prevent excessive morph calculation time, a subset of all the possible feature lines will be chosen.

We will first prewarp both rectified images to an intermediate shape, and then cross-dissolve between the two prewarped images to get the final intermediate image.

Let $P_1$ and $Q_1$ be endpoints of a feature line in the first rectified image, and let $P_2$ and $Q_2$ be endpoints of a feature line in second image. Beier-Neely morphing works by first interpolating feature line positions in the two rectified

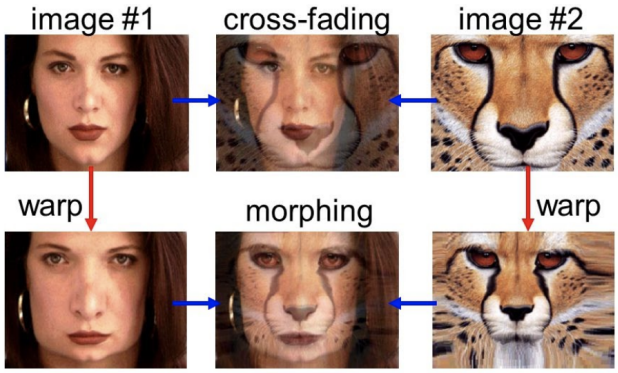


Figure 5. An example of two images first being warped to intermediate shapes on the lower left and right before interpolating to reach the bottom middle image, which is more convincing than the simple cross-dissolve seen in the top middle.

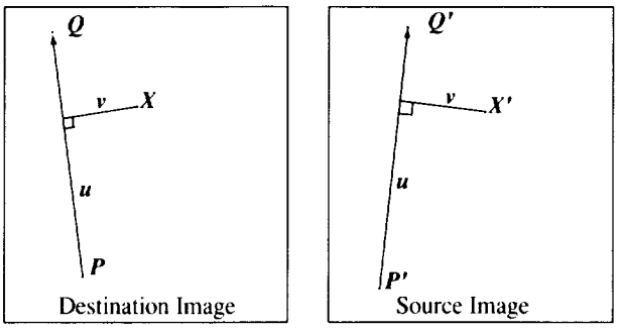


Figure 6. Offset $u$ and $v$ for a single feature line in Beier-Neely morphing [2]. In this case the "Destination" is our prewarp image, and our "Source" is either of the original rectified images.

images to get intermediate lines, using the following interpolation:

$$P = (1-t)P_1 + tP_2$$

$$Q = (1-t)Q_1 + tQ_2$$

where $P$ and $Q$ are interpolated endpoints, and $t$ is the interpolation ratio.

With the case of one interpolated feature line, we can then calculate the distance of every pixel $X$ in the intermediate image to that feature line $PQ$, and use this distance to calculate offsets $u$ and $v$ as shown in figure 6. $u$ and $v$ can be used to find the pixel to sample from in the "Source Image" as you see in figure 6; in this case the "Source Image" is either of our rectified images.

In the case of multiple feature lines, which is what I will use in my process, we need to calculate $u$ and $v$ for every pixel in the intermediate image in comparison to every single feature line, and use a weighted average to find the final pixel to sample.

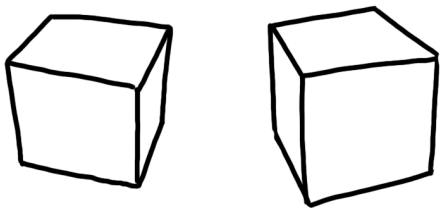

Figure 7. Two simple drawings of cubes that I made.

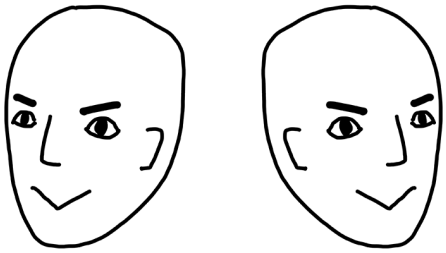

Figure 8. A face that I drew and reflected to create a second view.

### 3.3. Post-warp

To bring the intermediate image back to its desired location in world space from its parallel position, we apply a post-warp step, which essentially applies the reverse of an intermediate rectifying homography $H_s$.

Because $H_s$ is unknown in the situation where camera geometry is unknown, Seitz and Dyer suggest supplying control points to give a homography, or to interpolate between $H_0$ and $H_1$ [7].

## 4. Experiment

I applied my implementation of Seitz and Dyer's view morphing process to two main hand-drawn image sets, and observed the outcome to see if I could create smooth and natural-looking animations of rotation using the view morphing method.

### 4.1. Keyframe dataset

In order to test the view morphing procedure, we need 2D keyframes to interpolate. I created the small dataset used here by drawing my own 2D keyframes. As someone who dabbles in 2D animation both as a hobby and for work, I wanted to try view morphing my own drawings. I drew a simple cube from two different angles, shown in Figure 7, and a simple face shown in Figure 8. Each of the cubes was drawn individually, while only one face was drawn and reflected for a second view.

Notably, imprecise hand-drawn images such as these were of interest to me for investigation, as they do not necessarily make sense by 3D physical rules (for example, in Figure 7 you can see that lines that should be parallel are not necessarily parallel, and if you were to extend the edges as lines they would not intersect to the expected vanishing points).

As is detailed in more depth in the Results section, I also added a cube created in Blender for comparison of a more realistic shape which follows 3D physical rules.

### 4.2. Method of evaluation

For my personal keyframes, which do not have "ground truth" inbetweens, I will evaluate the resulting animations with subjective visual observations - whether or not it looks natural, whether or not it has a lot of artifacts and noise, and if it seems to retain the general shape of the object.

I also compare results of the view morphing process with results of simple image morphing (which just applies the Beier-Neely morph directly), to see if the view morphing process works better and preserves straight lines better like Setiz and Dyer mention.

### 4.3. Limitations in My Implementation

Due to limitations of timeframe, I was unable to complete the third and final post-warp step in Seitz and Dyer's process. As a result, the images I produced are all simply interpolated rectified images, which may appear unnaturally warped.

The implementations of the first two steps in the process can be found in `rectify_blend.py`, in the GitHub repository linked in the appendix.

### 4.4. Choosing Correspondences and Feature Lines

Coorespondence points were chosen by hand using the OpenGL program I wrote. When the user clicks on a point, its pixel coordinate will be printed. I used minimal correspondence points partly because I wanted to speed up the process, and partly because I was limited by manually clicking and I wanted to be precise where I could be.

For the hand-drawn cube, I used 13 correspondences and 7 feature lines.

For the blender cube, I used 15 correspondences and 15 feature lines.

For the face, I used 24 correspondences and 21 feature lines.

The reason for choosing a subset of feature lines rather than using all possible pairs of correspondence points is simply for speed of the program. Because Beier-Neely needs to compare all pixels to all lines for each intermediate morph, it can become very computationally slow with too many feature lines.

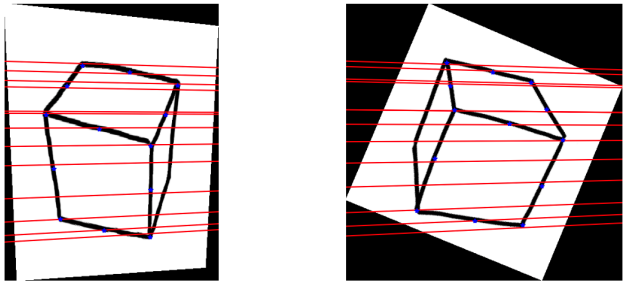

Figure 9. Rectified hand drawn cubes with epipolar lines plotted.

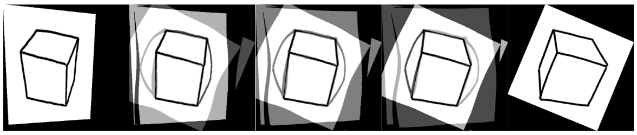

Figure 10. Morphing hand drawn cubes with rectification, where $t = 0, 0.3, 0.5, 0.7, 1.0$.

Figure 11. Morphing hand drawn cubes **without** rectification, where $t = 0, 0.3, 0.5, 0.7, 1.0$.

### 4.5. Results

Animations of all the results described here are available on an external slidedeck. The link can be found in the appendix.

#### 4.5.1 Hand drawn cube

The first set I tried was the hand-drawn cubes. Again for this set, I used 13 correspondences and 7 feature lines.

You can see the rectified images with epipolar lines drawn in Figure 9. While corresponding points are indeed on the same epipolar lines, the overall rectification did not work as I expected; you can see that the baselines of the front faces are not horizontal as one would expect from rectification like this. This is most likely due to a combination of too few and too noisy correspondences, as well as the non-3D-rules aspect of the cube.

I compared both morphing with the premorph rectification step as seen in Figure 10, and without (simply Beier-Neely morphing) as seen in Figure 11. Again, note that the morphs with rectification do not have the third post-morph step applied.

I was surprised to find that even with the pre-morph step, the incorrect warping of straight edges (seen on the left and right faces of the cube) appeared in almost equal intensity between the rectified and non-rectified morphs. From Seitz and Dyer's remarks on image morphing vs view morphing, I had expected the pre-morphed rectified version to better preserve straight lines [8].

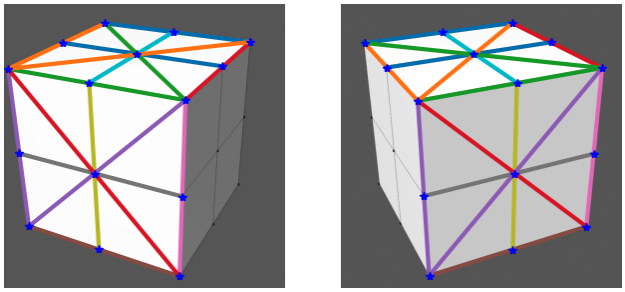

Figure 12. Large cube from Blender with 15 feature lines marked.

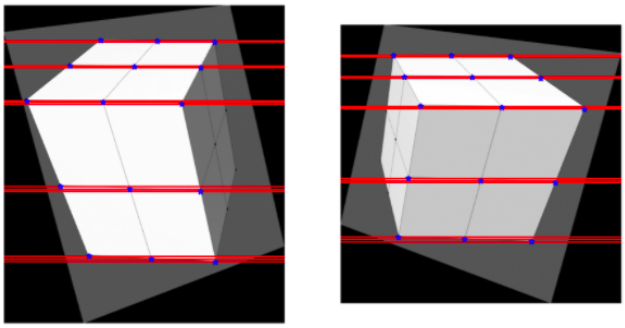

Figure 13. Rectified large Blender cube images with epipolar lines marked.

To see if the non-3D-rules aspect of the cube was the main factor in these poor results, I tried a subdivided cube made in Blender for a more precise and realistic comparison model.

#### 4.5.2 Blender cube

The hope with the Blender cube was to test my implementation on an image that followed real life 3D rules, and see if this differed from my results for the hand drawn cube.

For this set, I first tried 15 correspondence and 15 feature lines, which are marked in Figure 12.

The rectification gave results that were more in line with what I expected from a rectified cube; as you can see in Figure 13, the edges running parallel are horizontal.

However, the end result with the morphing is relatively similar to the hand drawn cube; the sides of the cube still seem to bulge during its transition, in both the rectified images in Figure 14 and the plain Beier-Neely morphs in Figure 15.

Since the rectification went as expected this time, this lead me to believe that the poor results are mainly related

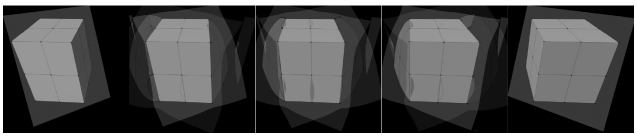

Figure 14. Morphing Blender cube with rectification, where $t = 0, 0.3, 0.5, 0.7, 1.0$.

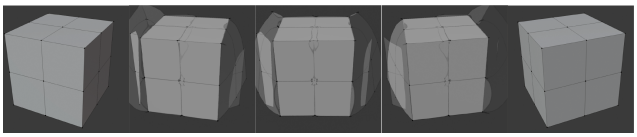

Figure 15. Morphing Blender cube **without** rectification, where $t = 0, 0.3, 0.5, 0.7, 1.0$.

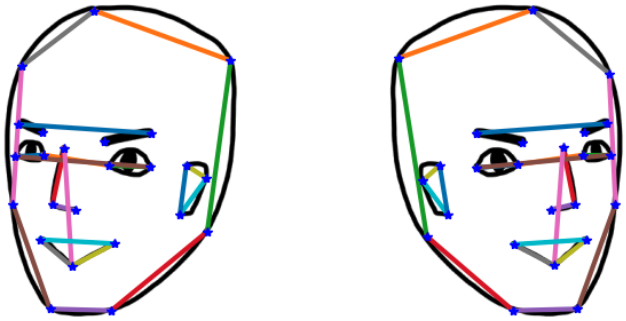

Figure 16. Faces with feature lines for Beier-Neely morphing plotted.

to noisy and too few correspondences, and too few feature lines selected.

#### 4.5.3 Face

The final image set that I tested my implementation on was a simple hand drawn face. I used 24 correspondences and 21 feature lines, which can be seen in Figure 16.

The final results for both the rectified and simple Beier-Neely morphs were terrible. As you can see in the plain Beier-Neely morphing outcome in Figure 18, there are intense artifacts and the features pinch in an unappealing way. In Figure 17 the rectified version is even worse. Somehow the rectification transformed the left-facing face to be right-facing, which was unexpected and could be a source of the poor quality.

While I initially thought 21 feature lines would be a moderate amount of feature lines for morphing, it appears that quite a few more would be required to get a desirable result.

Figure 17. Morphing hand drawn face with rectification, where $t = 0, 0.3, 0.5, 0.7, 1.0$.

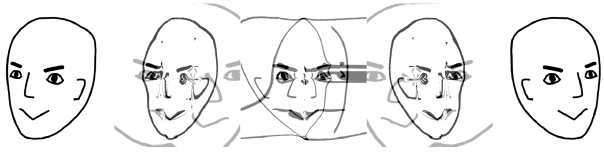

Figure 18. Morphing hand drawn face **without** rectification, where $t = 0, 0.3, 0.5, 0.7, 1.0$.

## 5. Conclusion

### 5.1. Interpretation of results

Though the evaluation of results was based on subjective visual appearance, I believe most viewers would rate the results negatively if they were to appear in the Sunday morning cartoons.

It is necessary to keep in mind that I had not implemented the third and final post-morph step in any of these results. However, I don't believe an inverse homography could help much in terms of straightening warped lines or erasing heavy artifacts. While it would be interesting to see what results I have with the third step in place, I am not sure that it would significantly enhance my results.

As I have mentioned briefly in the Results section, I believe these poor results are mainly attributable to too few and too noisy correspondences, as well as too few feature lines. I initially believed the non-3D-world aspect of line drawings was the main culprit. However, after testing my implementation on the 3D-world-abiding Blender cube, and seeing that rectification worked as expected but we still got poor results, I thought that most of the issues must be coming from correspondences and feature lines.

When initially selecting correspondences and feature lines, my goal was to select few enough that the process would run quickly, but enough to give some decent results. Evidently, I pushed too hard for speed and my results suffered.

That being said, even with as few as 21 feature lines for the face, creating 8 intermediate frames still took at least an hour due to the slow implementation of Beier-Neely morphing. Since every pixel must be compared to every feature line, computation time easily builds up, especially for larger images.

Selecting correspondences and feature lines manually through OpenGL was also time intensive and bothersome. I

had to focus and take my time to click precisely, and could not edit my selections afterwards. This also contributed to the noisy and imprecise nature of my correspondences.

### 5.2. View morphing for animation

Let us revisit the idea of making animation easier through this view morphing process; considering how laborious manually selecting correspondences and feature lines was, and how long Beier-Neely morphing takes to run, this is definitely not a viable option for animators (at least not with my implementation). Like Catmull mentioned, the extra time to provide the information (correspondences and feature lines) required for this pipeline to run would greatly overshadow any convenience of automatically generated inbetweens.

In hindsight, as 2D animation doesn't need to follow the 3D world rules that view morphing aims to preserve, other applications such as interpolating curves could be much more useful and expressive. Ultimately, animators want to have the control to create expressive and aesthetically pleasing animations, and perhaps the best way to do that is simply by hand.

### 5.3. Future ideas

Beyond improving this implementation by selecting denser correspondences and more feature lines, I could find ways to make the Beier-Neely morph faster, or research newer field morphing alternatives. In a similar vein, to make the whole process easier and smoother, it would be helpful to find correspondences automatically, perhaps through a CNN like Ji et al implemented [5].

In thinking of animators and how to streamline their process, it would also be good to consider interfaces with which to use an implementation like this one. How can we keep making things as efficient and easy as possible for animators, so that they can keep creating wonderful animations for the world to watch?

## References


[1] Cacani features. `https://cacani.sg/cacani-features/?v=7516fd43adaa`, 2020.

[2] T. Beier and S. Neely. Feature-based image metamorphosis, 1992.

[3] L. Carvalho, R. Marroquim, and E. V. Brazil. Dilight: Digital light table - inbetweening for 2d animations using guidelines. *Computers and Graphics*, 65:31–44, 2017.

[4] E. Catmull. The problems of computer-assisted animation. *SIGGRAPH Computer Graphics*, 12(3):348–53, 1978.

[5] D. Ji, J. Kwon, M. McFarland, and S. Savarese. Deep view morphing, 2017.

[6] T. Miura, J. Iwata, and J. Tsuda. An application of hybrid curve generation - cartoon animation by electronic computers, 1967.

[7] S. M. Seitz and C. R. Dyer. Toward image-based scene representation using view morphing, 1995.

[8] S. M. Seitz and C. R. Dyer. View morphing, 1996.

[9] N. Sun, T. Ayabe, and K. Okumura. An animation engine with the cubic spline interpolation. *2008 International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 109–112, 2008.

[10] J. Xing, L.-Y. Wei, T. Shiratori, and K. Yatani. Autocomplete hand-drawn animations. *ACM Transactions on Graphics*, 34(6), 2015.


## 6. Appendix

**GitHub repository:**
`https://github.com/CynthiaXJia/view-morphing-2d-animation`

**Animations of results:**
`https://docs.google.com/presentation/d/1Nu4PXLqI2rPEUc3EiESsuVsCLoDG-wVDRiFIHFumRew/edit?usp=sharing`