UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science
Software Engineering

Erich Erstu

Fluid Morphing for 2D Animations

A thesis submitted for the degree of

Master of Science in Software Engineering (30 ECTS)

Supervisor: Benson Muite

Tartu 2014

Acknowledgements

This thesis would have remained a dream had it not been for my brother Aleksander who immediately recognized this as a great idea. His artist's mind and ambitions as an animator made him an excellent companion for the journey of implementing Fluid Morphing. I would like to thank my good friends Indrek Sünter and Tauri Vahar for their philosophical support and early interest in my hypothesis. Finally, this thesis would not happen to be possible unless Professor Marlon Dumas took the initiative to kick start my research by finding me a supervisor.

Fluid Morphing for 2D Animations

Creation of professional animations is expensive and time-consuming, especially for the independent game developers. Therefore, it is rewarding to find a method that would programmatically increase the frame rate of any two-dimensional raster animation. Experimenting with a fluid simulator gave the authors an insight that to achieve visually pleasant and smooth animations, elements from fluid dynamics can be used. As a result, fluid image morphing was developed, allowing the animators to produce more significant frames than they would with the classic methods. The authors believe that this discovery could reintroduce hand drawn animations to modern computer games.

Key words: *image morphing*, *fluid simulation*, *automated inbetweening*, *point cloud morphing*, *blob detection*, *blob matching*

Voolav muundumine kahemõõtmelistele animatsioonidele

Magistritöö (30 EAP)
Erich Erstu
Resümee

Professionaalsel tasemel animeerimine on aeganõudev ja kulukas tegevus. Seda eriti sõltumatule arvutimängude tegijale. Siit tulenevalt osutub kasulikuks leida meetodeid, mis võimaldaks programmaatiliselt suurendada kaadrite arvu igas kahemõõtmelises raster animatsioonis. Vedeliku simulaatoriga eksperimenteerimine andis käesoleva töö autoritele idee, kuidas saavutada visuaalselt meeldiv kaadrite üleminek, kasutades selleks vedeliku dünaamikat. Tulemusena valmis programm, mis võib animaatori efektiivsust tõsta lausa mitmeid kordi. Autorid usuvad, et see avastus võib viia kahemõõtmeliste animatsioonide uuele võidukäigule — näiteks kaasaegsete arvutimängude kontekstis.

Märksõnad: *pildi muundamine*, *vedeliku simulatsioon*, *automeeritav võtmekaadrite kiilumine*, *punktpilvede muundamine*, *laikude avastamine*, *laikude sobitamine*

Contents

List of Figures

Chapter 1

Introduction

Image morphing[1] performed by computer software was first introduced in the early 1990s. The method is widely used in filmmaking to achieve shapeshifting and artificial slow motion effects. The common approach has been distorting the image at the same time that it is fading into another (see figure 1.1). [80, p. 360]

Figure 1.1: Frames of a morph between Erich Erstu and Kalevipoeg[2] where green dots indicate manually assigned correspondences

However, the distortion step requires a dense correspondence which is problematic because of the need for a manual annotation and even then unnatural artefacts are often created where the correspondence does not exist [66, p. 1]. Because of that, a lot of research has been done both in the academic community and in the movie industry [66, p. 1]. Novel image morphing techniques such as [6, 7, 66] attempt to enhance the conventional methods by reducing the need for human assistance. Unfortunately, none of these state of the art methods satisfy the needs of a video game developer.

This motivated the authors to come up with a novel idea of how to improve the area of image morphing specifically for 2-dimensional computer games [20]. In this work a new approach is proposed — *fluid morphing*, in which painted puddles of fluid reposition in 2D space to produce the morph.

[1]Morphing is a special effect in motion pictures that generates a sequence of inbetween images in which an image gradually changes into another image over time [48, p. 1].

[2]Kalevipoeg is the Estonian national epic and also the protagonist of a short adventure game Sohni - Second Visit to the Underworld [19, 45].

In 2D animations the artist has to draw pictures in a way that when presented fast enough an illusion of unbroken connection between the images appears [79, p. 13]. The smoother the animation the more frames there has to be. More frames means more time to be spent by the artist to finish the animation. To reduce the amount of work the animator could only draw the *key frames*[3] and then, based on the preferred frame rate[4], make the computer find all the in-between images [42, p. 125].

Unfortunately, the procedure of finding these missing frames is so complicated that human intervention is still needed. The principal difficulty becomes apparent when the drawings are just two-dimensional projections of the three-dimensional characters as visualized by the animator, hence information is lost [14, p. 350]. One could argue that this is just a problem of image recognition. However, that would only be the case under the assumption that the images share similarities [59, 63, 85, 23]. If the key frames present absolutely random data an algorithm is needed that would introduce *artificial imagination*.

While human intervention can potentially give the best results, it is not always the preferred method for image morphing. Sometimes it is only needed that the final animation looks fluid even if it comes with the cost of anomalies appearing on the intermediate images. As long as these anomalies appear and disappear fluidly they can be tolerated. Other times the images contain so many changes in details that it quickly becomes irrational for the human user to predefine all these deformations.

In the context of graphical computer games, it is common that *sprites* include mask colour in their *palette*. The mask colour usually indicates unimportant pixels that are to be skipped when drawing a masked sprite [77]. Having said that, fluid morphing is intended to cope with images that clearly distinguish between significant and unimportant pixels. The authors believe that such assumption allows further optimization of the morphing procedure which would not be possible for rectangular images where all pixels are equally important.

One notorious problem of image morphing is the *ghosting effect*. It becomes particularly visible when morphing *subimages* that contain anything but uniformly low image gradients[5]. Extra care must be taken to minimize the visibility of such artefacts. This could be done by adding automatic blob detection[6] to the algorithm so that the best matching subimages could be first morphed separately and then merged in the final morph.

[3]**Terms with dotted underline are explained in appendix A and can be clicked on.**
[4]Frame rate is the frequency at which an imaging device produces consecutive images [61].
[5]Image gradient is a directional change of colour in an image [2, p. 2].
[6]Blob detection is the detection of small, compact image primitives ("blobs") [32, p. 1].

In early video games frame rate was rather low so sprites had fewer frames. With the proposed methods it would be possible to enhance all these old games by rendering their sprites again for a higher frame rate. Lately revived classics such as Doom by its *Brutal Doom Mod*[7] (see video 8 and figure 1.2) could make a great use of fluid morphing.

Figure 1.2: *Brutal Doom* on *Zandronum*[8] engine, showing *id Software's DOOM.WAD*

Basic research gives away that there are no advanced image morphing libraries available as open source. The only such library is *libmorph* [30] and it does not satisfy many of the requirements that have been defined so far. That said, the aim of this work is to develop such a library and distribute it as free software. The results of this work would nourish the low budget artists that cannot afford commercial software.

In the next chapter the reader can find an overview of different solutions to related problems. The chapter for background describes all the technical building blocks that the authors needed in order to develop fluid morphing. Finally, chapter 4 is dedicated to describing the essence of the proposed solution and its implementation. To sum it all up, the authors present a conclusion that includes the most notable results of this thesis and list problems that were left unresolved.

[7]See http://www.moddb.com/mods/brutal-doom for details about the *Brutal Doom Mod*.
[8]See https://zandronum.com/ for details about *Zandronum*.

Chapter 2

Related Work

In the context of image morphing lies the notorious problem of ghosting artefacts. The authors propose that this problem can be solved with the aid of blob detection. Thus, a pixel-level image segmentation algorithm is needed that would distinguish between uniformly coloured blobs. That said, in the next section an overview of recent advances in image segmentation is given. Following that, in section 2.2 image morphing in general is reviewed.

2.1 Image Segmentation

Pixel clustering is one of the basic tasks for a system that needs to understand the content of images. Clusters of nearby pixels that are sometimes called "blobs" are the desired objects to detect. They are used for higher level reasoning to extract meaningful content from an image.

In 1923, Max Wertheimer [78] noticed the importance of grouping atomic pieces of a visual signal by their perceptual similarity, proximity and seamless continuation. Nevertheless, new research is still being done as many of the computational issues remain unresolved [67, p. 888]. Image segmentation is an endless research topic, driven by the concrete needs of every unique problem it tries to solve [57, p. 1278].

The huge number of approaches developed can roughly be grouped into template matching, watershed detection, structure tensor analysis and scale-space analysis methods [32]. With the emergence of discrete optimization, many computer vision problems are solved with energy minimization algorithms such as graph cuts [9, 41], tree-reweighted message passing [40, 76] and belief propagation [52, 83].

Next, to cover some of the related work, the authors review a set of papers they found interesting to examine. Unfortunately, most of them turn out to be rather unrelated to the method the authors themselves had to develop.

2.1.1 2005: Subpixel Precise Blob Detection

Fast and Subpixel Precise Blob Detection and Attribution introduces an algorithm for blob detection based on differential geometry. It starts with the extraction of potential center points of blobs in subpixel precision. Then, boundaries around such points are reconstructed. As a final step, various geometric and radiometric attributes are calculated [32, p. 2].

The method is reasonable for the extraction of elliptic blobs from grey scale images as shown in figure 2.1. It is a perfect example of a situation where a specific problem has driven the development of the image segmentation method. For advanced blob detection that has to differ between colours, this algorithm is completely useless. It is well hidden into mathematical obscurity that the given method just finds local extremums after blurring the original image a lot.

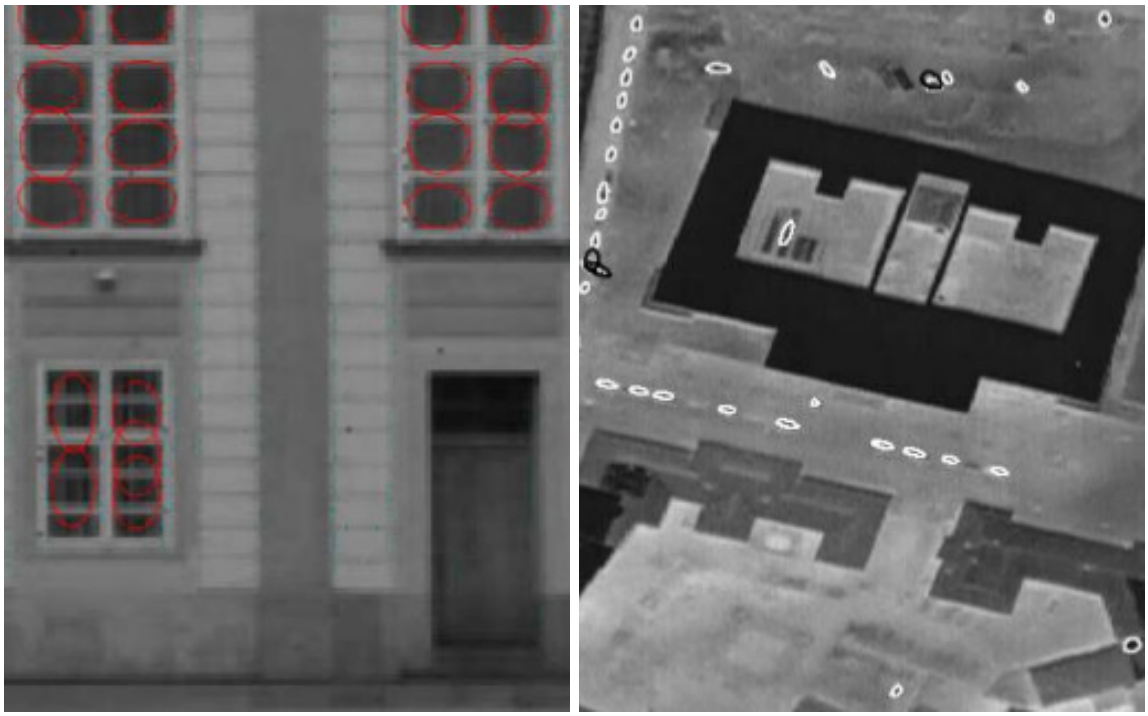


Figure 2.1: Blobs overlayed on original images [32, p. 4]

It is commonly known that image smoothing is a computationally expensive operation, thus there is nothing extraordinarily fast in this algorithm. However, due to the subjective nature of its results, it is difficult to say whether the paper provides a good solution in their local search space.

2.1.2 2010: Adaptive Regularization for Graph Cuts

Graph cut minimization formulates the segmentation problem as an energy function that consists of a data term and spatial coherency. The latter is the boundary length according to the contrast in the image, therefore minimizing the energy with this term leads to shorter boundaries. This technique is popular for interactive image segmentation but it fails to segment thin structures. [75]

Adaptive Regularization Parameter for Graph Cut Segmentation [13] proposes a method which arranges the effect of the regularization parameter on different parts of the image. The procedure first finds the edge probability maps with the Canny edge detector. By running the named algorithm at different hysteresis threshold levels, a linear average of these maps can be found. Edge pixels are determined from that combined probability map (see figure 2.2).

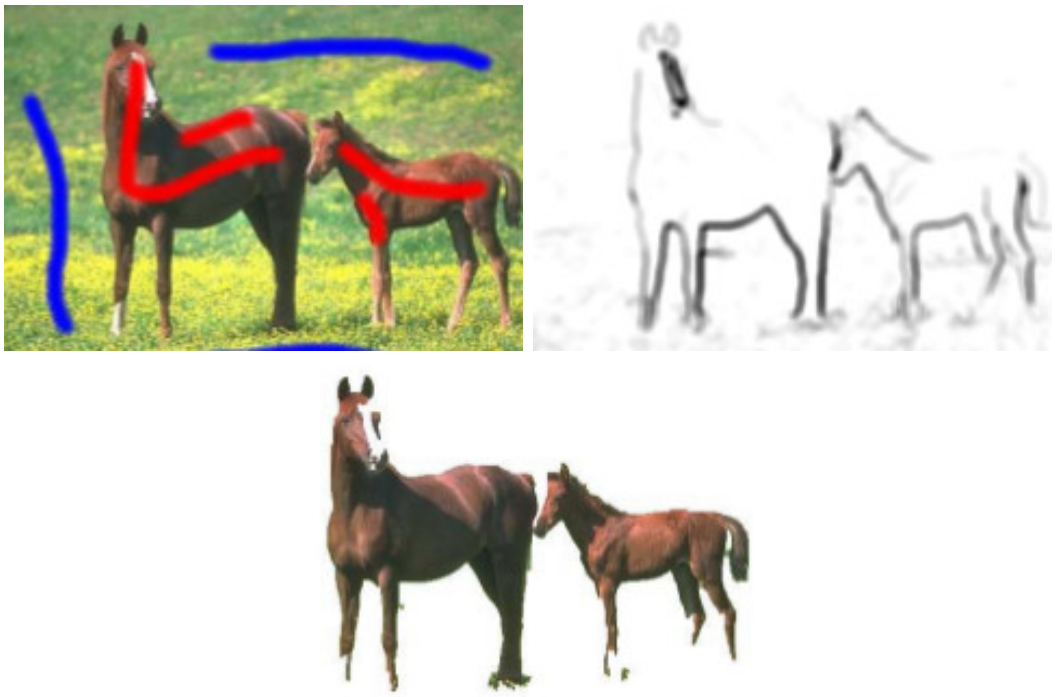


Figure 2.2: Probability calculation of each pixel [13, p. 5, 6]

The novelty of this approach is the idea that adaptively adjusting the regularization parameter protects the thin parts of the foreground from being over-smoothed [13, p. 6]. However, the proposed technique relies heavily on user input and only distinguishes colours by their intensity [13, p. 8]. Although it is a certain enhancement to graph cuts, it is unusable for fully automated computer vision. What is more, it assumes that there are distinguishable edges in the images, making it impractical for detecting blobs in extremely blurred images.

2.1.3 2012: Multiclass Pixel Labeling

The aim of Multiclass Pixel Labeling with Non-Local Matching Constraints is to provide segmentation of the image where each pixel is assigned a label from a predefined set of classes such as *sky*, *road* or *tree*. The given model is motivated by the idea that similar appearance of disjoint image regions suggests similar semantic meaning for pairs of corresponding pixels in the regions. [29, p. 1]

First, they capture long-range similarities between image regions as soft constraints [29, p. 1]. In their experiments, they find matching regions by densely sampling rectangular patches of size 32×32 to 96×96 in 16 pixel increments [29, p. 5]. Then, the resulting energy function is minimized using a graph-cut construction [29, p. 1]. To optimize the minimization of the energy function, a move-making algorithm[1] is used [29, p. 3]. A constraint is set so that corresponding pixels between two matching regions in the image would agree on their label [29, p. 2]. Experimental results are shown in figure 2.3.

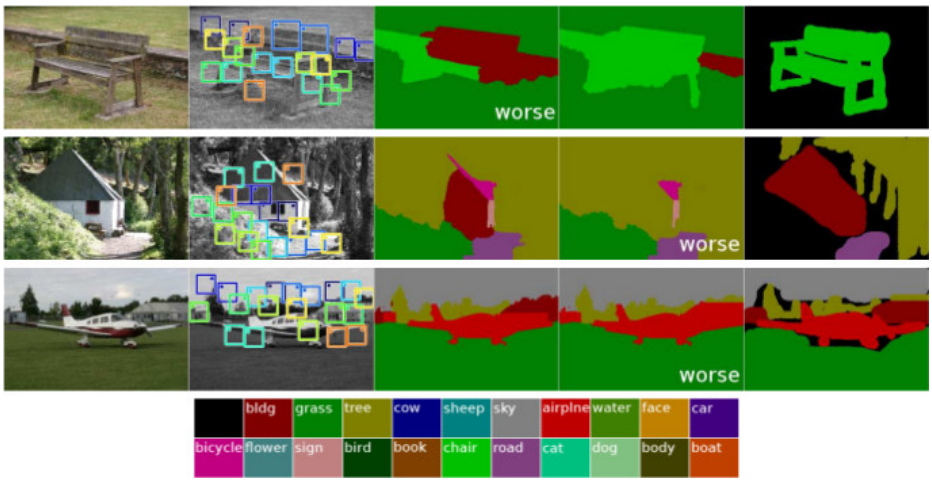


Figure 2.3: Example results of the Multiclass Pixel Labeling experiment [29, p. 7]

The method is only capable of detecting similar regions of the image when there is one-to-one pixel mapping — it cannot handle differently scaled regions nor it is able to search matches from more than one image [29, p. 6]. What is more, the paper lacks of explicit description for detection of specific objects which is ironically the most difficult part in object recognition. The reader could imagine that the plane in figure 2.3 was defined solely by red and white colours. If that is the case, the given research is really poor and the illustrative material provided is misleading.

[1] Move making algorithms minimize an energy function by starting from an initial labelling and making a series of changes (moves) which decrease the energy iteratively [38].

2.1.4 2013: Tensor-based Semantic Segmentation

In [55] a non-parametric approach for semantic segmentation is proposed. By using high-order semantic relations, a method to transfer meaning of known images to unlabelled images is proposed.

First, they define semantic tensors representing high-order relations of objects. Then, semantic relation transfer problem is formulated as semi-supervised learning. Based on the predicted high-order semantic relations they are able to segment several challenging datasets and assign labels to blobs (see figure 2.4).

Figure 2.4: Example results [55, p. 3073]

Although the method lacks any such references, it is much like [29] as it too assigns labels to blobs by their relative locations and known properties. The method can be applicable to various computer vision problems including object detection, scene classification and total scene understanding [55, p. 3079].

2.1.5 2013: Clustering on the Grassmann Manifold

The authors of [34] have developed an efficient and accurate Sparse Grassmann Clusterin method. They claim that the overlapping circles in figure 2.5 cannot be clustered into geometric models correctly using standard methods that measure distances between points. The described algorithm is designed to do it.

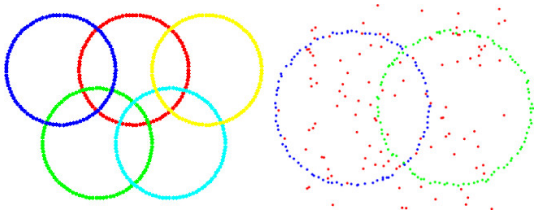


Figure 2.5: Overlapping circles [34, p. 3512, 3516]

The proposed method is computationally efficient and has a low memory requirement. It is scalable and can solve large-scale clustering problems, achieving results comparable to the state of the art [34, p. 3518]. However, it is important to note that this algorithm is meant to find clusters that form geometric primitives such as circles and lines. Hence, it probably fails to detect clusters of more complex shapes.

2.1.6 2014: Game Theory for Segmentation

Currently the latest development in blob detection is [49]. It proposes a segmentation algorithm within the framework of evolutionary game theory, where the Public Goods Game is employed. Its authors go even as far as to claim that their method outperforms the state of the art (see figure 2.6) [49, p. 14].

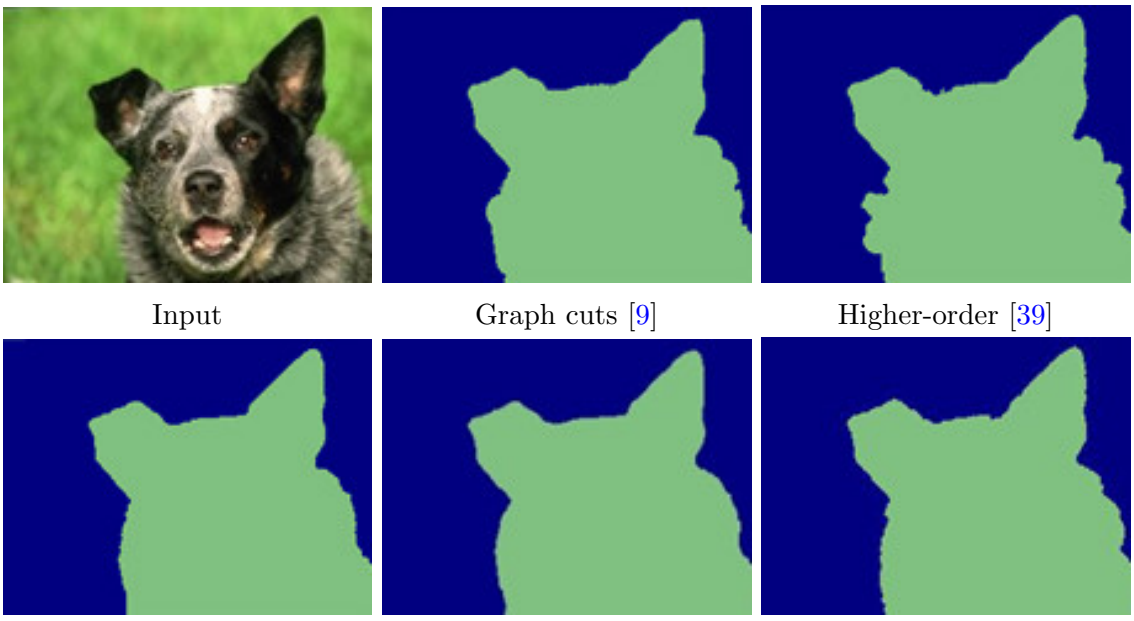


Figure 2.6: Comparison results on Segmentation Evaluation Database [51]

The method is comparable to [37] in a sense that it iteratively optimizes pixel labelling and clique[2] potentials. However, the difference lies in the fact that in this method, each pixel is related to multiple cliques, whereas in [37] each pixel is linked to one specified region [49, p. 3].

Similarity of neighbouring cliques is defined by Euclidean distance between their average colours [49, p. 6]. The feature-based probability of a pixel is calculated by following the procedure of K-means [49, p. 6]. For each superpixel, 3D CIE Lab colour is extracted [49, p. 11]. The latter is important to note because it mimics the non-linear response of a human eye to different colours.

In their problem statement it is emphasized that for an m label problem, they would like to partition the input image into m non-overlapping parts [49, p. 4]. This is a naive presumption because in the real world it is often not known how many objects need to be detected in the first place.

[2]Usually the clique is a set of pixels [49, p. 3].

2.2 Image Morphing

Image morphing deals with the metamorphosis of an image to another image, generating a sequence of intermediate images in which the image gradually changes into another over time. The practical problem of image morphing dates back to early 1990s and was mostly present in the context of motion pictures. For example, the technique has been widely used in the creation of special effects for movies and music videos such as Michael Jackson's *Black or White* (see video 7 starting at 3:35). [48]

2.2.1 1957: Classic Animations

First animations date back to more than 35 000 years. It can be seen in the ancient cave paintings that sometimes animals were drawn with four pairs of legs to show motion (see figure 2.7). However, the animation business truly got off in 1906 when the cartoonist James Stuart Blackton and the inventor Thomas Edison publicly released Humorous Phases of Funny Faces (see video 4). Their novelty was an instant hit and today they are known as the forefathers of the animated cartoon. [79, p. 15]

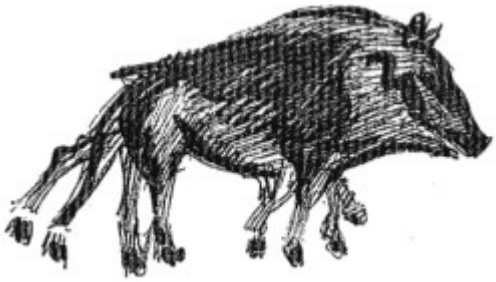


Figure 2.7: Early cave painting that displays motion [79, p. 11]

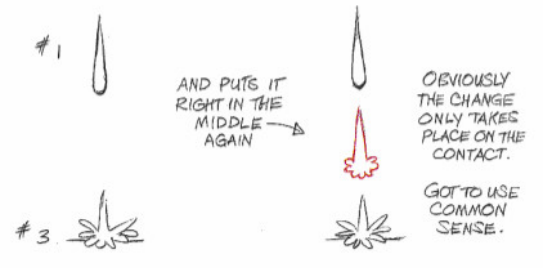


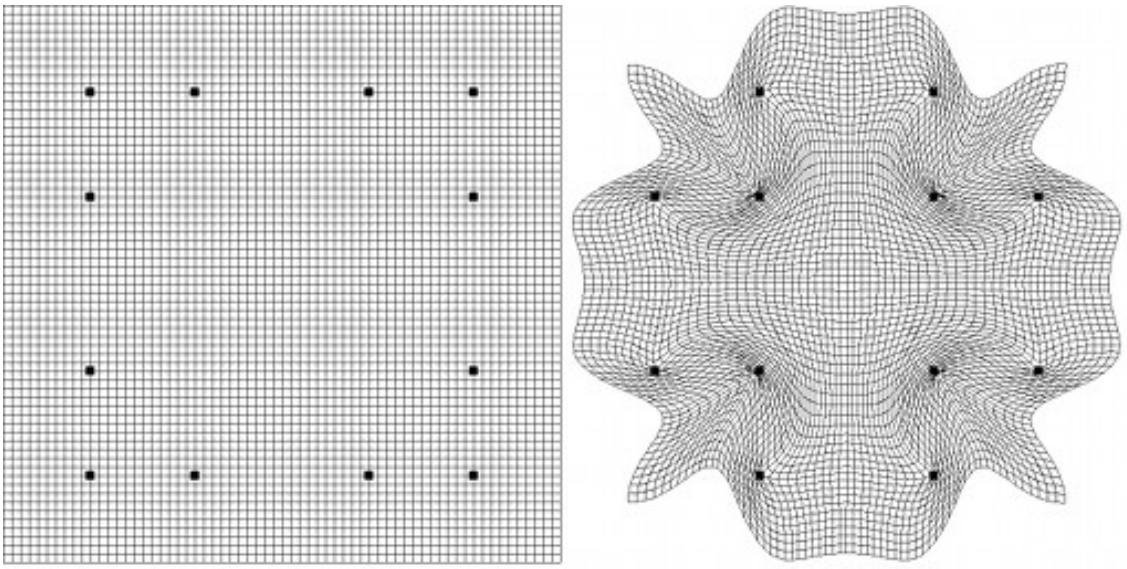
Figure 2.8: Example of a classic inbetweening mistake [79, p. 88]

Although computers can aid the animators by removing the dull work, their assistance is still limited to a certain level. For example, turns out that there are aspects of inbetweening[3] that require the inbetweener to understand what is happening on the animation scene. In a situation shown in figure 2.8 there is no reasonable way for a computer to get the morph right. It is rather the responsibility of the animator to prepare the key frames for the computer in a way that such mistakes would not happen. In the end, the outcome is still dependent on the adeptness of the animator and special algorithms can only eliminate the dull work.

[3]Inbetweening or tweening is the process of finding the correspondence between images so that an interpolated image that is dependent on the correspondence could be produced [14, p. 350].

2.2.2 1996: Conventional Image Morphing

Historically image morphing has been done by warping[4] and cross-dissolving the images that have a set of common predefined feature points. Visually this technique can be described by figure 2.9 where the disposition of feature points determines the transformation of the whole image. Same method was used to generate figure 1.1.



(a) The undeformed shape (b) A deformation of the plate

Figure 2.9: A deformation example [48, p. 7]

The blending of warped images in each frame is prone to produce blurring and ghosting during the process of morphing [81, p. 312]. To solve this problem, a new method is proposed in [81]. It suggests blending an individual pixel only after it has reached the best match with the destination image. Similarly to fluid morphing (see section 4.3.2), an energy function minimization takes place.

Nevertheless, the most tedious part of such image morphing is to establish the correspondence of features between images [50, p. 4]. When having to morph images with absolutely no common features then warping becomes completely irrational. This means extra work for the animator so that an obvious need arises for more advanced morphing techniques.

[4] A warp is a two-dimensional geometric transformation that generates a distorted image when it is applied to an image [48, p. 1].

2.2.3 2005: Two-Dimensional Discrete Morphing

In essence, the previously described image morphing techniques are very different from fluid morphing. However, two-dimensional discrete morphing takes a rather similar route by looking more into individual pixels than trying to generalize them into patches or polygons.

The proposed method uses a distance function [62] associating to each point of the object the distance of the nearest pixel of the background [8, p. 411] (see figure 2.10). A rigid transformation is performed that aligns the shapes to decrease geometrical differences between them [8, p. 409]. By iteratively adding or suppressing pixels, a transformation from one object to another is found [8, p. 409]. This results in a linear algorithm for computing an intermediate shape between two binary shapes [8, p. 418]. The same authors have also proposed a method for computing the discrete average of n two-dimensional shapes [7].

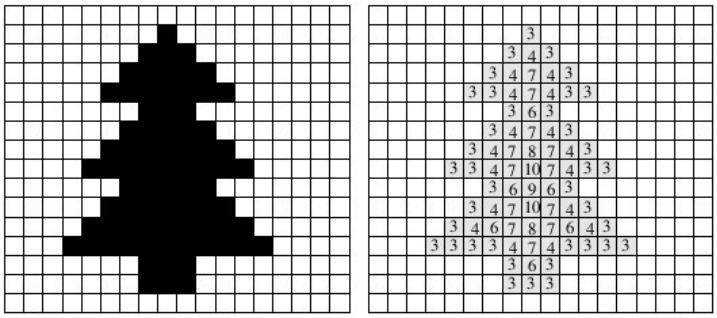


Figure 2.10: An example of a distance transformation [8, p. 411]

Regrettably, the algorithm is capable of morphing just binary shapes making it impractical for cases where textured objects have to be morphed. The resulting inbetweens seem to display disturbingly many rough edges (see figure 2.11) and tend to become unintuitive if the morphed objects are too different [8, p. 418].

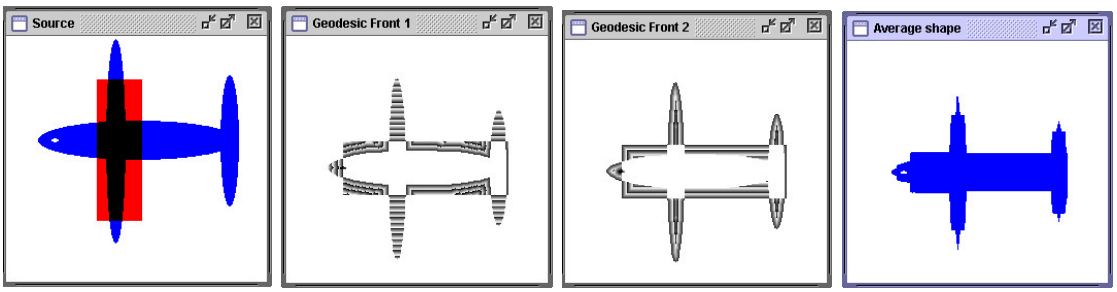


Figure 2.11: Discrete deformation of various shapes [8, p. 419]

2.2.4 2009: N-way Image Morphing

N-way Image Morphing offers efficient N-way interpolation technique that preserves rigidity and improves extrapolation capabilities (see figure 2.12 and video 5). The method makes use of As-Rigid-As-Possible Shape Manipulation technique proposed in [33] but offers improvements to interpolate between a set of input images. [6, p. 6]

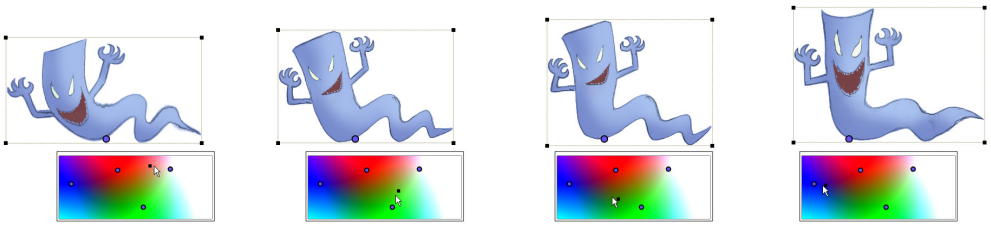


Figure 2.12: Shapes generated with N-way Image Morphing [6, p. 22]

However, N-way Image Morphing faces difficulties when given anything but simple polygons as input. Animations produced this way are limited to motions in the camera plane, making it inconvenient to use drawings with overlapping parts, such as an arm drawn in front of the chest [6, p. 17]. To further exacerbate the situation, it makes no attempt to warp textures in order to improve blending. Thus, this method is prone to ghosting effects (see figure 2.13).

Figure 2.13: Ghosting effect in N-way Image Morphing (see video 5) [6]

The authors of this work were able to capture the above image by pausing the provided video at the very right moment. Although one could argue that this could be just a video compression artefact, it is unlikely because the original paper [6] does not mention anything about the prevention of the ghosting effects.

Although n-way morphing produces visually pleasant and definitely interesting results, it is unfortunate that at this point of time, there is no source code nor demo application publicly available. Fluid morphing, on the contrary, provides a solution to all the problems mentioned previously. Theoretically, it would even be possible to enhance fluid morphing to support n-way interpolation (see section 5.1).

2.2.5 2010: Regenerative Morphing

Regenerative morphing (see video 6) does not assume any similarities in the key frames and does not attempt to detect rigid shapes in images (see figure 2.14). It is built upon [68] (Bidirectional Similarity Method) so that the morph sequence is directly synthesized from local regions of the two sources, making it temporally coherent and locally similar to the sources [66, p. 6].

Figure 2.14: Example results of Regenerative Morphing [66, p. 1]

Even though regenerative morphing is useful for inbetweening unrelated images, it produces a lot of blurring effects and obvious ghosting (see figure 2.15). Also, the algorithm only counts on the next and the previous key frame when producing inbetweens. In contrast, fluid morphing is capable of taking all the key frames into consideration (see figure 4.23).

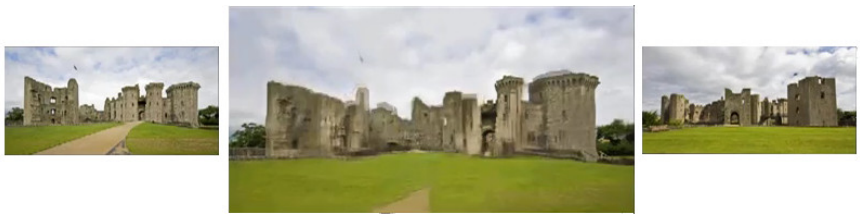


Figure 2.15: Ghosting effect in Regenerative Morphing [66]

Similarly to [6], the authors of regenerative morphing have not made the source code publicly available. There have been attempts to implement regenerative morphing as stated in [46] but its authors conclude that the implementation was not efficient compared to the results achieved in [66]. Namely, the developed Matlab implementation was terribly slow and produced a lot of blur, making the whole algorithm seem useless. The results would probably have been better if the implementors directly used GPU capabilities as it was suggested in the original paper [46, p. 6].

Chapter 3

Background

In this chapter some of the building blocks essential to the construction of the proposed algorithm are reviewed. To fully understand this thesis, it is recommended to be familiar with the subsequent terminology.

3.1 Artificial Intelligence

Artificial Intelligence as a formal discipline is intended to make computers do things, that when done by people, are described as having indicated intelligence [10, p. 1]. It sometimes deals with the problems that cannot be solved by trying all possibilities due to limits set by time and memory. Even when it is not known exactly how to solve a certain problem, it may be possible to program a machine that searches through a large space of solution attempts [53, p. 9].

Hill climbing is the simplest algorithm for artificial intelligence. It is a fundamental technique that is always used in the background of more complex systems [53, p. 10]. The heuristic undertakes to progress unidirectionally from their starting point to a local optimum [28, p. 191]. Its great virtue is that the sampling effort grows only linearly with the number of parameters, so the addition of more parameters of the same kind ought not cause an inordinate increase in difficulty [53, p. 10]. Moreover, it requires only a limited amount of memory and if one or more solutions exist in the search space it can be surprisingly efficient at finding it [65, p. 334].

However, the limitation of a hill climbing procedure is that the local optimum obtained at its stopping point may not be a global optimum [28, p. 191]. When dealing with an optimization problem the local search cannot be used to determine whether the solution found is globally optimal [65, p. 334]. The authors chose this method because it is easy to implement and gives plausible results. If needed, it is trivially upgradable to a more advanced method such as simulated annealing.

3.2 Catmull-Rom Spline

Catmull-Rom splines [15] are a family of C^1 continuous cubic interpolating splines that allow local control and interpolation [74, p. 1]. The named spline was developed for computer graphics purposes, having its initial use for the design of curves and surfaces [35, p. 1]. In practical applications these curves are often used to interpolate the control points of animations and trajectories [84, p. 19].

In appendix D a C++ implementation of the Catmull-Rom Spline is given. The authors had to modify the original implementation[1] (see listing 3.1) so that *closed splines* could be produced.

Listing 3.1: Original Implementation That Does Not Produce Closed Splines

```
#define BOUNDS(pp) { if (pp < 0) pp = 0; else if (pp >= (int)vp.size()-1) pp = vp.size() - 1; }
Vec3D CRSpline::GetInterpolatedSplinePoint(float t) {
    // Find out in which interval we are on the spline
    int p = (int)(t / delta_t);
    // Compute local control point indices
    int p0 = p - 1;     BOUNDS(p0);
    int p1 = p;         BOUNDS(p1);
    int p2 = p + 1;     BOUNDS(p2);
    int p3 = p + 2;     BOUNDS(p3);
    // Relative (local) time
    float lt = (t - delta_t*(float)p) / delta_t;
    // Interpolate
    return CRSpline::Eq(lt, vp[p0], vp[p1], vp[p2], vp[p3]);
}
```

Assuming that most animations used in video games are meant to run in a loop, the splines used in the context of this work need to be closed as shown in figure 3.1. By doing so, seamlessly repeatable morphs could be achieved.

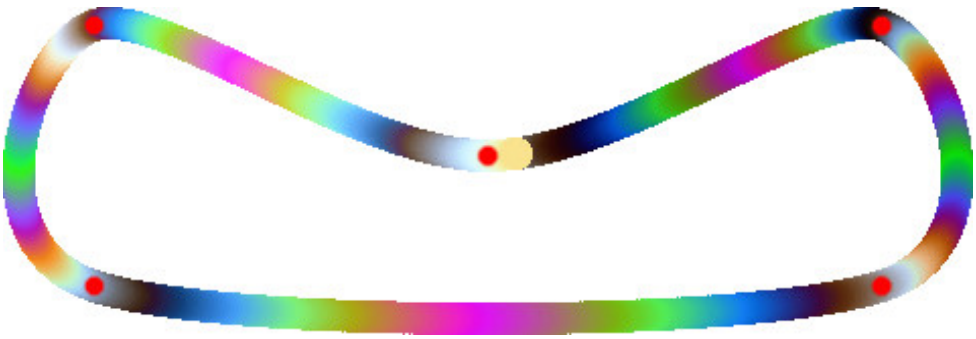


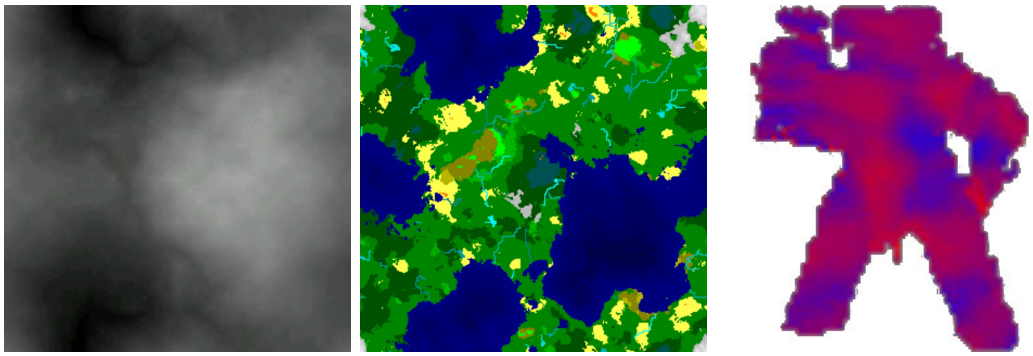
Figure 3.1: A closed Catmull-Rom spline

Although there are other types of splines with the required attributes, the authors chose the Catmull-Rom spline mainly because it was suggested in the *Allegro.cc Forums*[2] and it is known as the most commonly used interpolating spline [4, p. 377]. Most other splines are inconvenient for the use in image morphing because they do not allow setting the exact control points to pass through.

[1] http://svn.lam.fr/repos/glnemo2/trunk/src/catmull_rom_spline.cc
[2] https://www.allegro.cc/forums/thread/612242

3.3 Perlin Noise

Since its introduction [58], Ken Perlin's noise function has found a wide variety of uses. The idea is to create a basis function that varies randomly to serve as a building block for procedural textures [4, p. 396]. By summing the basic functions on different frequencies in a way that higher frequency samples have less weight, textures shown in figure 3.2 can be achieved.



(a) Marble texture [22] (b) World map [18, p. 27] (c) Morphing

Figure 3.2: Pictures produced with Perlin noise

In figure 3.2a the most common use for Perlin noise is shown. The method produces natural textures and because it is a procedural method it can be stored on a hard drive in a very compact way — needing just the seed for the random number generator that allows full reconstruction of such images.

Figure 3.2b displays a more sophisticated use of Perlin noise. It is used in [18] to produce random yet plausible world maps. The reader might notice that both of these textures appear as seamless. This is another nice property of such textures because they can be tiled to cover much larger surfaces, consuming less memory and computing power.

In the context of this thesis, Perlin noise is used for quite a different purpose. In figure 3.2c there is an interpolated image, having one of its sources coloured completely red and the other one fully blue. The intermediate frame displays a texture similar to figure 3.2a. That is because some parts of the image morph gradually faster than other parts. The local morphing speed is determined by Perlin noise (see video 2).

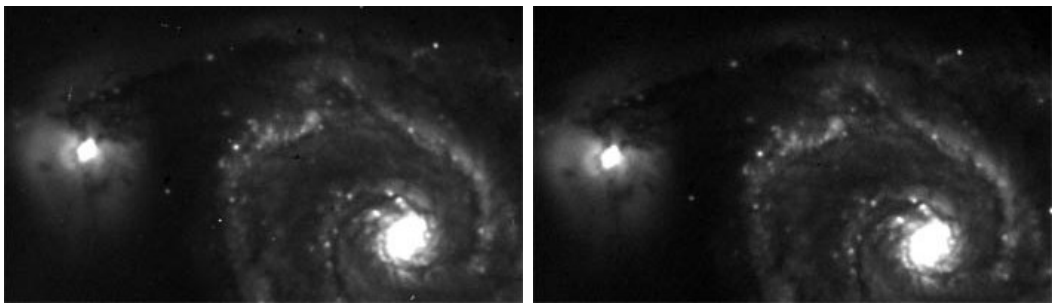
In appendix E a C++ implementation of the Perlin noise function is given. The authors included it to serve a proof of concept rather than present a fully optimized method for Perlin noise dependent image morphing. Consequently, the rendering speed with it is left far from its potential optimum.

3.4 Noise Reduction

Fluid morphing being a meshfree[3] method is prone to producing noise. This happens because it is not trivial to decide the colour of a pixel if many atoms lie on it. To solve this problem one could use a nonlinear filter such as the median filter that has proven to be useful for noise reduction [11, p. 116].

However, the authors of this work see it as fighting with the symptoms and thus consider it inadvisable. Instead, the problem should be dealt with on much lower level. The noise produced by locally chaotic particles has one virtue — changing the seed of a random number generator also changes the noise. This opens up a possibility to reduce noise by the same means it is done in astronomy.

Stacking multiple single exposures of the same part of the sky (see figure 3.3) turns out to be an effective way to eliminate cosmic rays, satellite tracks, ghost images and capturing device imperfections [31, p. 2]. The method is easily applied to real time image morphing by running many morphs in parallel and stacking them to produce a single combined morph.



(a) Average (b) Median

Figure 3.3: Average (3.3a) and median combine (3.3b) [1]

Mean stacking or averaging (see figure 3.3a) can be done with two or more images. The method is fast compared to other stacking techniques but produces slight blur. Radical outliers in any of the input images notably change the result. Thus, mean stacks will contain a density of artefacts that increases with the number of frames entering the stack [31, p. 2].

Median stacks are more resistant to outliers (see figure 3.3b) when a large enough number of overlapping exposures is available [31, p. 3]. Extreme pixel values have no effect since only the middle value of the sorted list of candidates is taken [1]. However, for outlier free images, mean combining carries a better signal to noise ratio [56, 31].

[3]Meshfree methods approximate partial differential equations only based on a set of nodes without the need for an additional mesh [25, p. 4].

3.5 Fluid Simulation

The history of computational fluid dynamics dates back to 19th century when Claude Navier and George Stokes formulated the equations that describe the dynamics of fluids [27, 54, 73]. These equations have now been accepted as a very good model for fluid flow [69, p. 121]. However, new algorithms are constantly being developed in order to optimize the simulation process for the latest hardware and practical needs.

Conventionally, two categories of simulation methods exist: Eulerian grids and Lagrangian particles [16, 47, 54]. Although pure particle methods trivially guarantee mass conservation and provide a conceptually simple simulation framework [16, p. 219], they often use explicit schemes such as the forward Euler method, which can easily become unstable [47, p. 17]. Instability leads to numerical simulations that "blow up", setting serious limits on speed and interactivity [69, p. 121]. To improve stability, the semi-Lagrangian method is used [47, p. 17].

For example, a hybrid Eulerian/Lagrangian Material Point Method (MPM)[4] has demonstrated itself as a computationally effective particle method [70, 71]. MPM outperforms purely Eulerian methods by the ability to track parameters such as mass and momentum while still using a Cartesian grid to keep the nearest neighbour queries fast [71, p. 104]. To solve the equations of motion, particle data is projected to a background grid on which the equations of motion are solved [70, p. 924].

In aerodynamics and other fields where computational accuracy is important, fluid is simulated off-line and then visualized in a second step [54, p. 154]. In computer graphics, on the other hand, the shape and behaviour of the fluid are of primary interest, while physical accuracy is secondary [69, p. 121]. It is self-evident that image morphing does not need extreme physical accuracy. Therefore, techniques optimized for the use in interactive systems such as computer games [47] should be preferred.

The authors have decided to use Grant Kot's implementation [43] of the Material Point Method because they did not find anything better when searching for usable C++ code. It stands out by the fact that it uses the quadratic B-spline presented in [70] for interpolation and cubic interpolation method [36] to minimize grid artefacts. Although the original implementation's source code is unavailable, Xueqiao Xu has provided poorly documented C++ and Python versions [82] under the MIT License. It is strongly advised for the reader to see the original implementation [43] as it includes an interactive application to demonstrate the method.

[4] In MPM the terms particle and material point can be used interchangeably [5, p. 479].

Chapter 4

Fluid Morphing

Previous attempts to achieve convincing image morphing often try to produce the morph in a continuous fashion with the use of polygons. This is sometimes rational when correspondences between feature points have already been manually defined. However, in this paper all manual preparation is considered undesirable.

Fluid morphing only makes the assumption that there could be unimportant pixels in the input images. Because of that, shape contours here are equal to feature points in other image morphing techniques. What is more, most computer graphics is stored in raster format. Thus, discrete approaches have the advantage of not having to convert the images into vector graphics prior to morphing.

In this chapter, the authors present their novel image morphing method — fluid morphing. First, uniformly coloured blobs are detected and matched across all the key frames. *Blob chains* acquired that way are then separately morphed to generate a set of attractors later to be used in the fluid simulator. Finally, fluid particles are forced to gravitate towards their individual attractors, generating a fluid morph.

4.1 Blob Detection

In this section an agglomerative hierarchical blob detection method is presented. The technique is specifically designed to prepare images for morphing. However, it can also be used for other purposes such as artistic image processing. The implementation is written in C++ because the authors believe it is the best programming language for computationally heavy and practical uses.

The algorithm contains two major steps that are both described in separate subsections. The *blobifying* step is meant to do the core work by clustering the input image into intuitive filled sectors. After that, the unifying phase carries the purpose of clustering outliers into separate units by their distances.

4.1.1 Blobifying the Image

The concept of the presented procedure is to imagine that the image to be blobified is actually a puddle of multi coloured liquids. Because the fluid particles are in a microscopic but constant movement, the colours would blend over time. Therefore, if one would care to wait long enough, such a puddle would end up as a uniformly coloured blend (see figure 4.1).

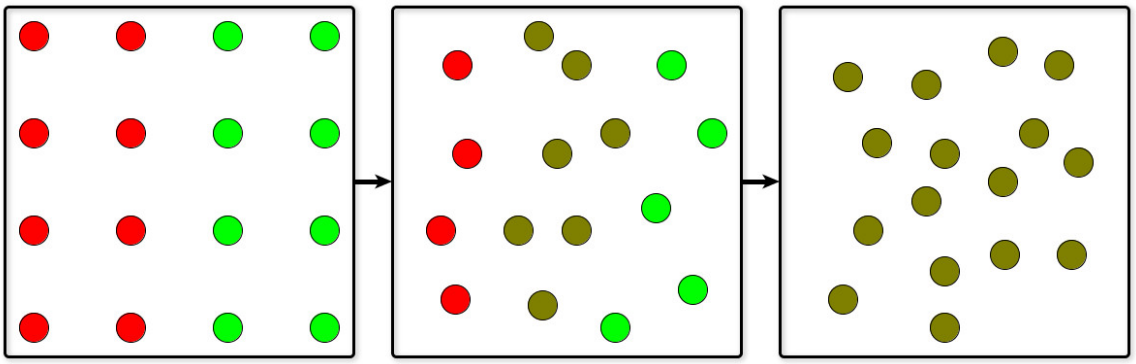


Figure 4.1: Colours blend over time

To achieve this behaviour programmatically on discrete data such as the pixels of an image, every pixel is treated as an atomic blob. Each blob contains at least one pixel and has a representative pixel which is the average of all of its pixels. Before the atomic blobs could be iteratively merged, two requirements must be met: *chaos* and *equality*.

Chaos can be achieved by shuffling the list of atomic blobs so that the order by which they start blending would be random. This has to be done only once in the beginning of the whole procedure. Later, whenever iterating over all the blobs, it is safe to assume that the next blob could be located anywhere within the borders of the input image.

Equality constraint makes sure that in a single iteration every blob can expand only once at maximum. Equality is further enforced by the chaos attribute defined previously, because only in a shuffled list of atomic blobs every blob has an equal chance of being the first one to expand. For images that contain large areas of the same colour, equality makes the blob borders less artificial looking.

An image is converted into a set of atomic blobs by iterating over every one of its pixels. Fully transparent pixels are to be skipped because the presented method is specifically designed for video game sprites. The reader should note that unlike the input image, the set of atomic blobs acquired is not a 2-dimensional array. Instead, it is a special *frame* structure (see listing 4.1).

Listing 4.1: Frame Structure

```
typedef struct key_frame {
    std::vector<blob*>      blobs;  // Vector of blobs.
    std::map<size_t, pixel> pixels; // Map of pixels by position.
    std::map<size_t, blob*> owners; // Map of blobs by position.
    double                  x,y,r,g,b,a;
    size_t index          =0; // Position in the nest container.
    size_t first_expansion=0; // Blobs before this index cannot expand.
    size_t first_dust     =0; // First under-sized blob to be unified.
} frame;
```

The frame structure stores all the pixels directly in a map container for fast and optimal access. A pixel's key in that map is the 1-dimensional position p in its 2-dimensional *(x, y)* coordinate space (see equation 4.1). It is self-evident that such a set up restricts the application to images that are not wider than *UINT16_MAX* pixels.

$$p = 2^{16} \cdot y + x \qquad x = p \bmod 2^{16} \qquad y = \left\lfloor \frac{p}{2^{16}} \right\rfloor \tag{4.1}$$

However, because most of the video game sprites are typically 32, 64, 128 or 256 pixels wide, the defined maximum width is vastly more than enough. By limiting the pixel's x and y coordinates to *UINT16_MAX* (2 byte) range, a memory and speed optimization possibility becomes available on 64-bit architecture. Namely, the pixel structure is defined to consume no more than 8 bytes (4 for coordinates and 4 for RGBA), making it rational to pass it by value instead of a pointer or reference (see listings 4.2 and 4.3).

Listing 4.2: Pixel Structure

```
typedef struct pixel {
    uint16_t x;
    uint16_t y;

    color c;
} pixel;
```

Listing 4.3: Colour Structure

```
typedef struct color {
    uint8_t r;
    uint8_t g;
    uint8_t b;
    uint8_t a;
} color;
```

Having said that, the *blob* structure is defined in listing 4.4. Such blob instances are created when initially iterating over all of the input image's pixels. For each significant pixel, a blob is created and put into the frame structure. The blob's surface will be exactly its only pixel's position and the border will be its 4 neighbouring positions.

Listing 4.4: Blob Structure

```
typedef struct blob {
    size_t index;              // Position in the nest vector.
    std::set<size_t> surface; // Pixel positions for surface.
    std::set<size_t> border;  // Pixel positions around surface.
    size_t group  =0;
    bool   unified=false;
    double x,y,r,g,b,a;
} blob;
```

Figure 4.2 shows example iterations of the blobifying algorithm. Arrows point to a blob's neighbours. Thus, each arrow indicates a position in the blob's border. Empty sockets are not stored in memory at all — unimportant pixels are not mapped. The reader can see that the sum of all example colour values is 1774 which divided by 14 (the number of initial atomic blobs) gives 126. Therefore, it is clear that the proposed agglomerative hierarchical clustering results in a single blob that is coloured as the average of the initial blobs.

Every step the next blob gets to expand as shown in the given figure. It chooses a neighbouring blob to merge with, preferring the one that is closest to its average colour. The borders of the merged blobs will be unified and the average colour recalculated according to the proportion of surfaces. The process is repeated until a preferred number of blobs remains or none of the blobs could expand any more.

In spite of the fact that it is not illustrated in figure 4.2, for the best results the colour distance formula must not be the Euclidean distance between the RGB colour vectors. Although RGB is very common and easy to implement, it is non-linear with visual perception [24, p. 6]. Instead, a more advanced colour space should be considered — the one that appreciates human perception to colour differences (CIELuv and CIELab [24, p. 7]).

For a device independent colour space, the authors have chosen the HSP[1] colour model, informally proposed by Darel Rex Finley in 2006 [44, p. 218]. The named colour model gives very good results in comparison to RGB and XYZ. What is more, Finley has provided the C source code for the HSP colour conversions in his web page.

[1]HSP — hue, saturation and perception (http://alienryderflex.com/hsp.html).

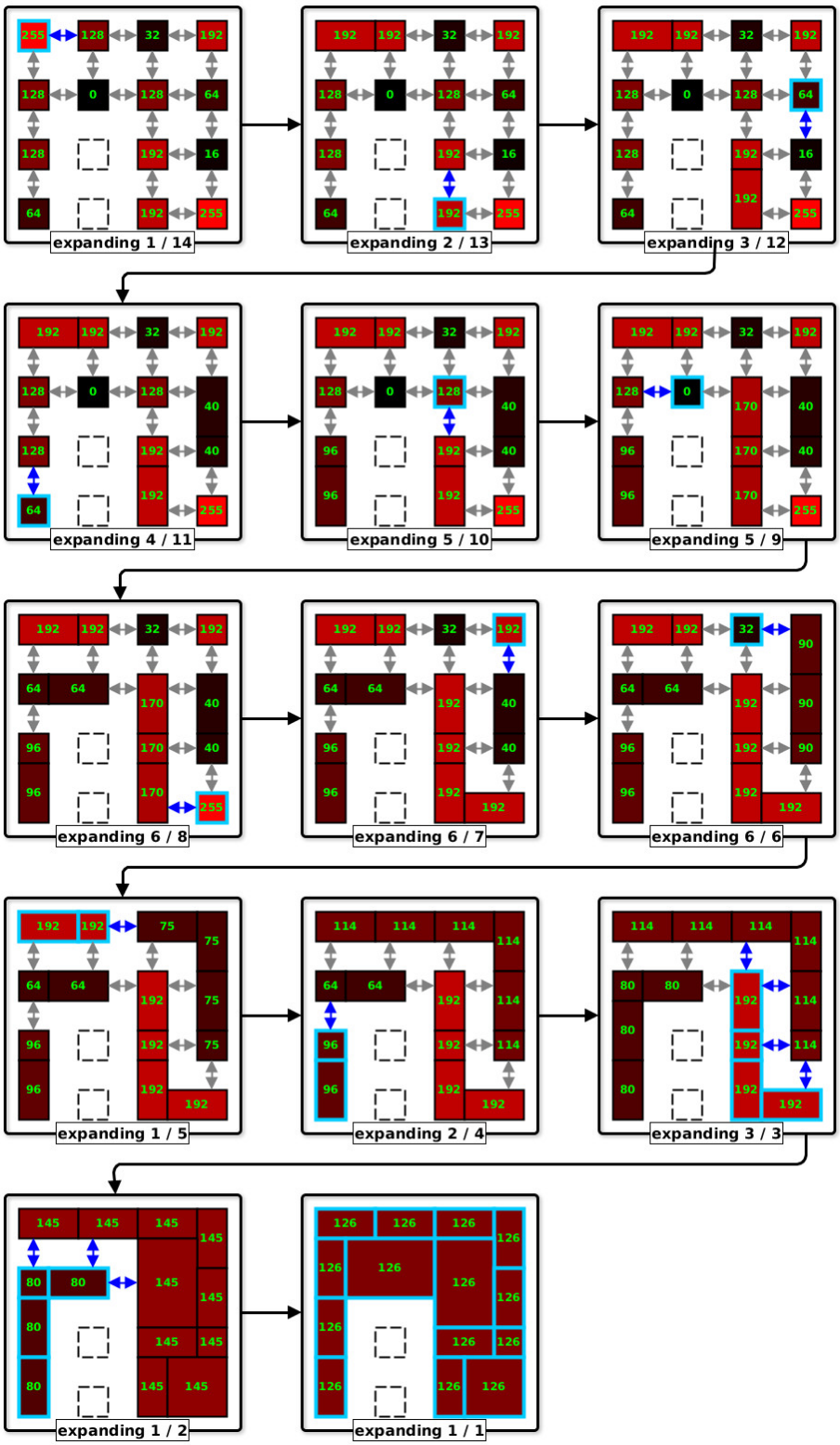


Figure 4.2: Example blob formation

4.1.2 Unifying the Outliers

Under-sized blobs are outliers. For some images, the number of such outliers may be unpleasantly large. An algorithm is needed to unify the under-sized blobs into larger clusters. In figure 4.3, a probabilistic method for fast clustering is presented. It requires that the blobs would be shuffled. For simplicity, the given figure illustrates the unifying process of 1-dimensional blobs.

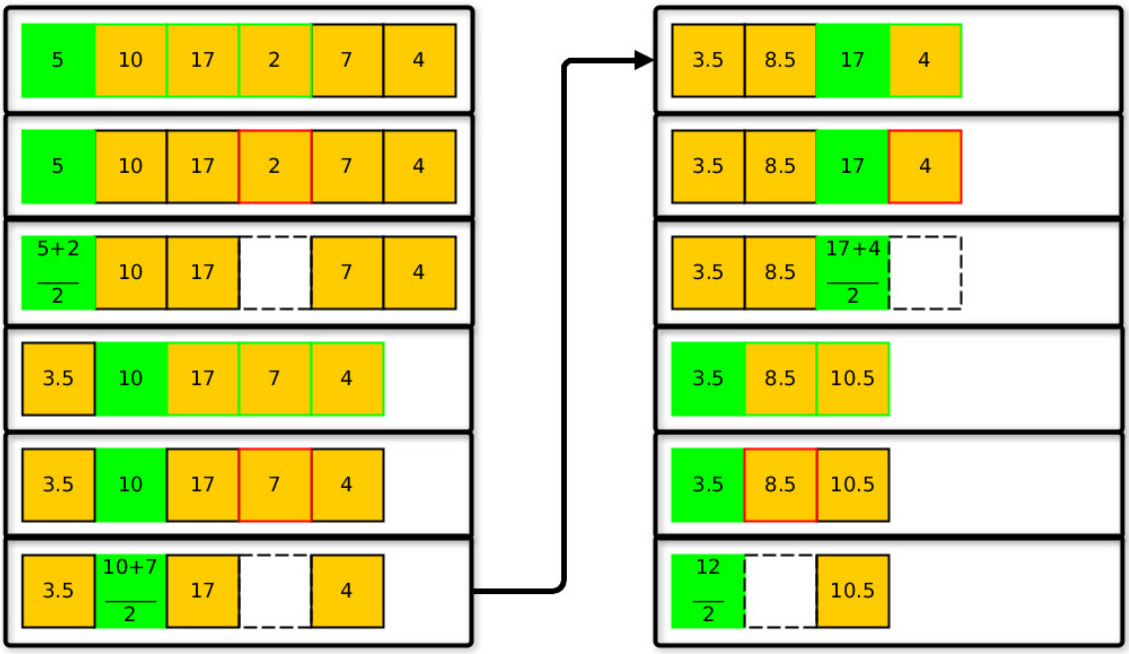


Figure 4.3: Unifying outliers

The method for clustering outliers is probabilistic and hierarchical. A parameter can be given to force such clusters into local regions of the image. When this parameter is defined, only the outliers within the specified vicinity can be unified. However, that neighbourhood is not circular but instead square shaped for faster lookups. Because the cluster borders are fuzzy anyway, it can be argued that for vicinity queries, using a box instead of calculating Euclidean distance is more effective.

Clustering the outliers should be considered as an exceptional procedure. For that reason, effort is not made to differentiate the colours of outliers when unifying them. For special purposes such logic can be easily added on top of the proposed algorithm.

4.1.3 Results

The authors have implemented a simple yet powerful blob detection algorithm. Although it is specifically designed for the use in image morphing, it is also applicable for colour count reduction. Example results can be seen in figure 4.4. Namely, figure 4.4b shows an undefined number of blobs in a distinctive manner while figure 4.4c shows these same blobs by their average colours.

(a) Input image (b) Distinct blobs (c) Coloured by average

Figure 4.4: Blobs of a dog

The above figures were generated by calling the AtoMorph demo program with the parameters shown in listing 4.5. If the reader is interested in running this program (see *AtoMorph Library*), it is recommended to first start it with the *help* flag.

Listing 4.5: Blobifying the Image of the Dog

```
$ ./atomorph --help
$ ./atomorph --file distinct dog.png --blobs-as-distinct --blobs-max-size 128
$ ./atomorph --file average dog.png --blobs-as-average --blobs-max-size 128
```

Whether the proposed technique could detect just the two most outstanding blobs similarly to figure 2.6 is found out by conducting a series of experiments. Turns out that detecting a strictly specified number of blobs in an intuitive manner is not trivially possible as seen in figure 4.5. However, on some occasions the algorithm performs incredibly well, which provides a basis for future research (see section 5.1). The below figure was generated using the commands given in listing 4.6.

(a) Seed 1 (b) Seed 2 (c) Seed 3 (d) Seed 4 (e) Seed 5

Figure 4.5: Detected blobs at the moment when just the last two blobs remain, having given different seeds for the random number generator

Listing 4.6: Attempts to Detect Exactly 2 Blobs

```
$ ./atomorph --file 2_blobs_1 dog.png --blobs-as-average --blobs 2 --seed 1
$ ./atomorph --file 2_blobs_2 dog.png --blobs-as-average --blobs 2 --seed 2
$ ./atomorph --file 2_blobs_3 dog.png --blobs-as-average --blobs 2 --seed 3
$ ./atomorph --file 2_blobs_4 dog.png --blobs-as-average --blobs 2 --seed 4
$ ./atomorph --file 2_blobs_5 dog.png --blobs-as-average --blobs 2 --seed 5
```

Figures 4.6 – 4.8 display the intended use of the proposed algorithm in the context of video games and image morphing. The reader can see that distinct features such as the eyes are successfully detected as separate blobs. The images were generated with the commands given in listings 4.7 – 4.9. It took ∼90 milliseconds to complete that process on a single core of a computer with AMD Phenom(tm) II X4 955 processor and 3.9 GiB memory, having a 64-bit Linux Mint 14 for the operating system.

(a) Input image (b) Distinct blobs (c) Coloured by average

Figure 4.6: Blobs of the Battle Lord sprite from *Duke Nukem 3D (1996, 3D Realms)*

(a) Input image (b) Distinct blobs (c) Coloured by average

Figure 4.7: Blobs of the orc from *Dungeon Keeper 1 (1997, Bullfrog Productions)*

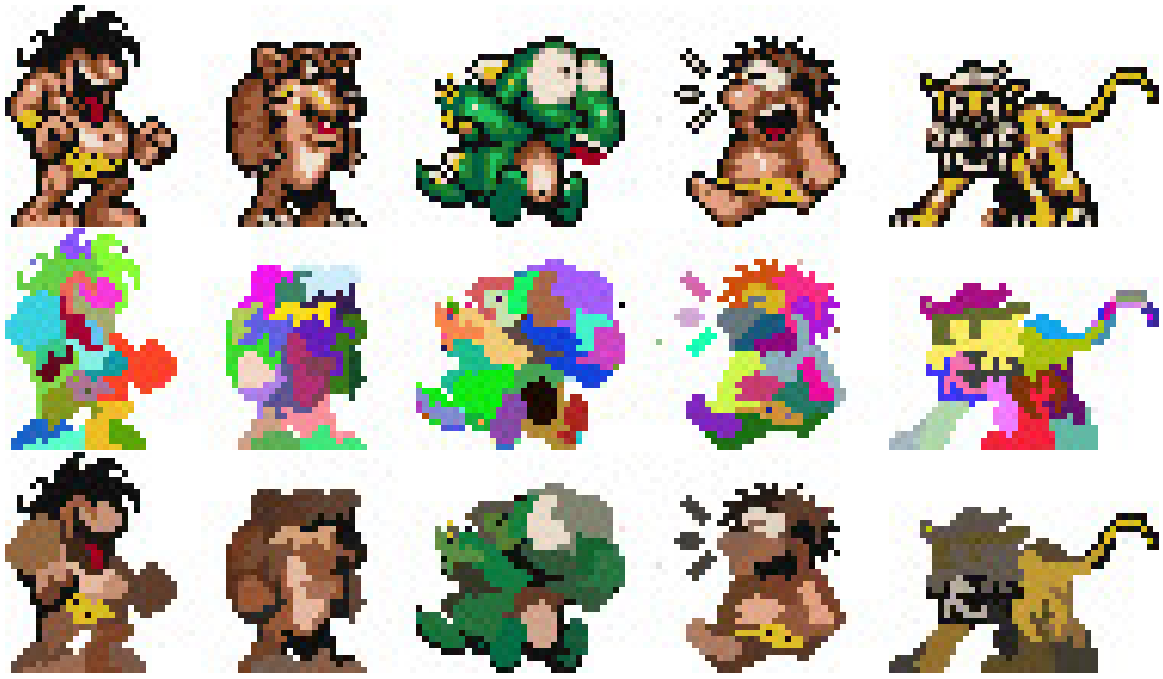


Figure 4.8: Blobs of the sprites from *Prehistorik 2 (1993, Titus Interactive)*

Listing 4.7: Blobifying the Battle Lord

```
./atomorph -f bl_dst lord.png --blobs-as-distinct -t 128 -m 3 -B 128 -S -1 -g 16
./atomorph -f bl_ave lord.png --blobs-as-average -t 128 -m 3 -B 128 -S -1 -g 16
```

Listing 4.8: Blobifying an Orc

```
./atomorph -f dk1orc_dst orc.png --blobs-as-distinct -t 30 -B 64
./atomorph -f dk1orc_ave orc.png --blobs-as-average -t 30 -B 64
```

Listing 4.9: Blobifying Sprites from Prehistorik 2

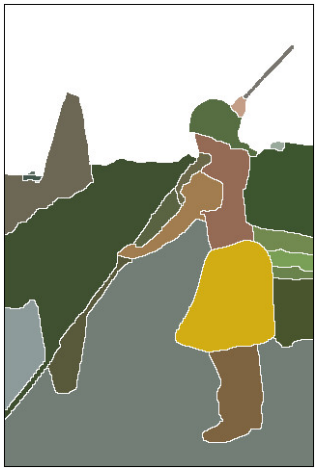
```
./atomorph -f pre2_dst pre2.png --blobs-as-distinct -t 128 -B 16
./atomorph -f pre2_ave pre2.png --blobs-as-average -t 128 -B 16
```

As seen in the above images, the proposed blob detection algorithm does its job really well, especially when considering its conceptual simplicity. Such results are definitely plausible for the use in image morphing where the fast movements during the morphing procedure make any incorrectly detected blobs almost unnoticeable.

The authors wish to emphasize that the sole purpose of the proposed technique is to detect definite blobs at first priority and leave everything else to chance. In the context of this work, such approach is justified by the fact that fluid morphing does not need to generate logical transitions — all it has to do is to generate smooth and "ghost" free transitions.

To give the reader an impression of how the developed blob detection algorithm really compares to other well known image segmentation methods, figure 4.9 shows the blobs of an image taken from the Berkeley Segmentation Data Set [3]. It is evident, that the authors' method gives somewhat less intuitive results. However, it provides a good basis for future research, which is intricately discussed in section 5.1.

Input image



Berkeley's algorithm

Authors' distinctive method

Authors' descriptive method

Figure 4.9: Comparison with the Berkeley's generic segmentation algorithm

The authors' method is optimized for speed and relies heavily on luck. The reason for this is the fact that the proposed image morphing method is not designed to understand the content of the images but instead provide a quick and simple means for the detection of uniformly coloured image patches. Having detected such patches in all of the input images, chains of similar blobs can be found. The latter is also known as blob matching, which is the essence of the next section.

4.2 Blob Matching

The previous section presents a simple and intuitive blob detection algorithm. Unfortunately, that alone does not help much at finding desired morphs. The problem is that after blob detection every key frame may contain a different number of blobs. What is more, these blobs know nothing about their best correspondences across all the key frames. This section is dedicated to provide a solution to these problems.

4.2.1 Concept

During the procedure of blob detection, some valuable attributes are stored in every blob's structure. These attributes are *size* (s), *average colour* (r, g, b, a) and *the center of mass* (x, y). The authors deem it sufficient to use just these three simple vectors when finding chains of best matching blobs.

The process of blob matching can be seen as an energy minimization task. In equation 4.2, the system's energy E for h key frames is the sum of *blob chain* lengths where w is the number of blobs per key frame. Because key frames may initially have different number of blobs, *volatile blobs* should be added to the frames that lack blobs. After that, every key frame contains exactly w blobs, making it possible to find the system's energy with the given formula.

$$
\begin{aligned}
E &= \sum_{i=1}^{w}\sum_{j=1}^{h}(W_1 \cdot D_1 + W_2 \cdot D_2 + W_3 \cdot D_3) \qquad r, g, b, a, D_1, D_2, D_3 \in [0, 1] \\
D_1 &= \frac{\sqrt{(x_{i,j} - x_{i,k})^2 + (y_{i,j} - y_{i,k})^2}}{d} \qquad D_2 = \frac{|s_{i,j} - s_{i,k}|}{s_{i,j} + s_{i,k}} \\
D_3 &= \frac{\sqrt{(r_{i,j} - r_{i,k})^2 + (g_{i,j} - g_{i,k})^2 + (b_{i,j} - b_{i,k})^2 + (a_{i,j} - a_{i,k})^2}}{2}
\end{aligned}
\tag{4.2}
$$

In the above equation, $k = 1 + j \bmod h$ — referring to the next blob in the closed chain of blobs — and d is the diagonal length of the minimal bounding box that surrounds the pixel set of the entire system. The latter is needed to normalize the locational distances to a compact range.

The length of a blob chain is the sum of distances between every sequential pair of blobs in that chain. Because there are three vectors contributing to the distance of two blobs, that distance is actually a weighted average of distances D_1, D_2 and D_3 where W_1, W_2 and W_3 are the weights respectively. The numeric values of these weights should be calibrated accordingly to the exact practical needs.

4.2.2 Algorithm

For the iterative minimization of the system's energy, a move making algorithm is used. The atomic move there is a swap of two blobs in the row of blobs that share the same key frame. The system's state is defined by a two-dimensional array of pointers to blobs. In that array, columns indicate blob chains and rows mark the key frames.

As said in the previous section, initially key frames may contain unequal number of blobs. This is taken care of by adding volatile blobs where needed. When the system's state is initialized, its absolute energy is computed once and stored for later use (see listings 4.10 and 4.11).

Listing 4.10: Initializing the System's State

```
// Find the frame with the largest number of blobs.
std::map<size_t, frame>::iterator it;
size_t blob_count = 0, frame_count= 0;
for (it=frames.begin(); it!=frames.end(); ++it) {
    frame *f = &(it->second);
    if (blob_count < f->blobs.size()) {
        blob_count = f->blobs.size();
    }
    frame_count++;
}

blob_map_w = blob_count; blob_map_h = frame_count;
size_t i,j,f=0;
// To save space on paper, memory is assumed not to run out here.
blob_map = (blob ***) malloc( blob_map_w * sizeof(blob **) );
for (i = 0 ; i < blob_map_w ; ++i ) {
    blob_map[i] = (blob **) malloc( blob_map_h * sizeof(blob *) );
}

// Fill blob map with pointers to real blobs.
for (it=frames.begin(); it!=frames.end(); ++it) {
    frame *fp = &(it->second);
    // Add empty blobs if needed.
    while (fp->blobs.size() < blob_count) {
        blob* new_blob    = new blob;
        new_blob->unified = true;
        new_blob->index   = fp->blobs.size();
        fp->blobs.push_back(new_blob);
    }
    std::shuffle(fp->blobs.begin(), fp->blobs.end(), e1);
    size_t blobs = fp->blobs.size();
    for (size_t b=0; b<blobs; ++b) {
        blob_map[b][f] = fp->blobs[b];
        fp->blobs[b]->group = b;
    }
    ++f;
}
blob_map_e = get_energy(blob_map);
```

Listing 4.11: Used Functions

```
double thread::get_energy(struct blob ***map) {
    if (blob_map_h == 0
    ||  blob_map_w == 0) return 0.0;

    blob *pframe_blob;
    blob *cframe_blob;
    double e=0.0;

    for (size_t i=0; i<blob_map_w; ++i) {
        pframe_blob = map[i][blob_map_h-1];

        for (size_t j=0; j<blob_map_h; ++j) {
            cframe_blob = map[i][j];
            e += blob_distance(pframe_blob, cframe_blob);
            pframe_blob = cframe_blob;
        }
    }
    return e;
}

double thread::blob_distance(const blob *b1, const blob *b2) {
    size_t sz1 = b1->surface.size();
    size_t sz2 = b2->surface.size();
    size_t szs = sz1+sz2;
    double pix_dist = 0.0;
    double col_dist = 0.0;
    double siz_dist = 0.0;

    if (szs > 0) {
        siz_dist = fabs(double(sz1)-sz2)/double(szs));
    }

    if (sz1 > 0 && sz2 > 0) {
        pixel p1,p2;
        p1 = create_pixel(b1->x,b1->y,b1->r*255,b1->g*255,b1->b*255,b1->a*255);
        p2 = create_pixel(b2->x,b2->y,b2->r*255,b2->g*255,b2->b*255,b2->a*255);
        pix_dist = sqrt(double(pixel_distance(p1, p2))/bbox_d);
        col_dist = color_distance(p1.c, p2.c);
    }

    return (blob_xy_weight   * pix_dist +
            blob_rgba_weight * col_dist +
            blob_size_weight * siz_dist);
}
```

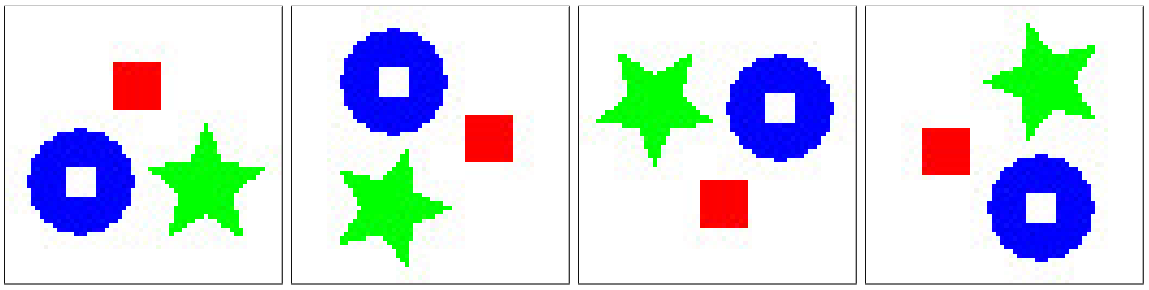
For the actual blob matching, a hill climbing algorithm on the defined energy function was implemented. In essence, it tries swapping randomly chosen blobs in a randomly chosen key frame and accepts only the swaps that decrease the system's energy. After each swap, the energy does not need to be recalculated but instead decreased by the local change in value (see listing 4.12).

Listing 4.12: Matching Blobs with Hill Climbing

```
std::uniform_int_distribution<size_t> uniform_dist_x(0, blob_map_w - 1);
std::uniform_int_distribution<size_t> uniform_dist_y(0, blob_map_h - 1);
size_t x1,x2;
size_t y      = uniform_dist_y(e1);
size_t y_next = (y+1) % blob_map_h;
size_t y_prev = (y>0 ? y-1 : blob_map_h - 1);
x1 = uniform_dist_x(e1);
do {
    x2 = uniform_dist_x(e1);
} while (x1 == x2);

blob* x1_y_prev   = blob_map[x1][y_prev];
blob* x2_y_prev   = blob_map[x2][y_prev];
blob* x1_y        = blob_map[x1][y     ];
blob* x2_y        = blob_map[x2][y     ];
blob* x1_y_next   = blob_map[x1][y_next];
blob* x2_y_next   = blob_map[x2][y_next];
bool  x1_volatile = x1_y->surface.empty();
bool  x2_volatile = x2_y->surface.empty();

if (x1_volatile && x2_volatile) {
    return false; // No point in swapping empty blobs.
}

double x1_e_before, x2_e_before, x1_e_after, x2_e_after;
x1_e_before = blob_distance(x1_y_prev, x1_y) + blob_distance(x1_y, x1_y_next);
x2_e_before = blob_distance(x2_y_prev, x2_y) + blob_distance(x2_y, x2_y_next);
x1_e_after  = blob_distance(x2_y_prev, x1_y) + blob_distance(x1_y, x2_y_next);
x2_e_after  = blob_distance(x1_y_prev, x2_y) + blob_distance(x2_y, x1_y_next);
double c1 = x1_e_before + x2_e_before;
double c2 = x1_e_after  + x2_e_after;

if (c1 > c2) {
    blob *buf       = blob_map[x2][y];
    blob_map[x2][y] = blob_map[x1][y];
    blob_map[x1][y] = buf;
    blob_map[x1][y]->group = x1;
    blob_map[x2][y]->group = x2;
    double gain = c1 - c2;
    if (blob_map_e >= gain) blob_map_e -= gain;
    else                    blob_map_e  = 0.0;

    if (x1_volatile) {
        x1_y->x = (x2_y_prev->x + x2_y_next->x)/2.0;
        x1_y->y = (x2_y_prev->y + x2_y_next->y)/2.0;
    }
    if (x2_volatile) {
        x2_y->x = (x1_y_prev->x + x1_y_next->x)/2.0;
        x2_y->y = (x1_y_prev->y + x1_y_next->y)/2.0;
    }
}
```

4.2.3 Results

To test the developed blob matching algorithm, a simple series of images was drawn (see figure 4.10). Then, these images were given as an input to the AtoMorph demo program with different command line parameters as shown in listing 4.13.



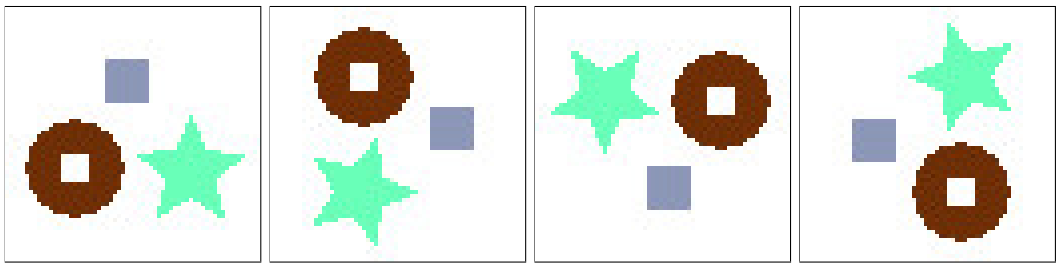
(a) Key frame 1 (b) Key frame 2 (c) Key frame 3 (d) Key frame 4

Figure 4.10: Primitive shapes with clearly distinguishable features

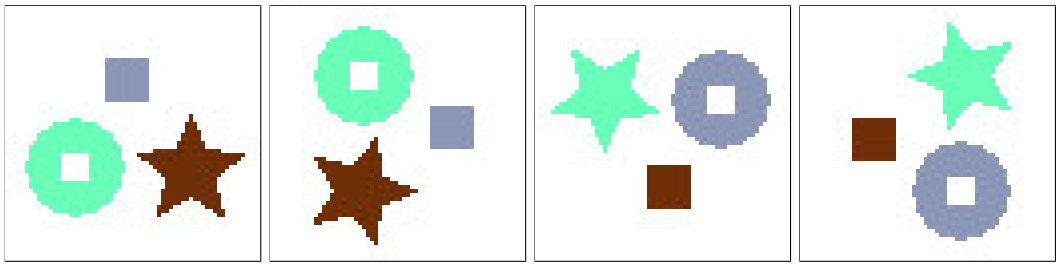
Listing 4.13: Matching the Blobs of Simple Shapes by Colour

```
./atomorph --blobs-as-distinct RGB_1.png RGB_2.png RGB_3.png RGB_4.png -F 4 -c 1 -z 0 -p 0 -s 0
./atomorph --blobs-as-distinct RGB_1.png RGB_2.png RGB_3.png RGB_4.png -F 4 -c 1 -z 0 -p 0 -s 7
```

The reader can see that the program is first launched with *seed 0* and then with *seed 7* for the last parameter. Turns out, that the hill climbing algorithm gets stuck in a local optimum when launched with *seed 7*. In figure 4.11, the generated blob chains are shown in both cases, distinguishable by their colour.



(a) Global optimum achieved with *seed 0*



(b) Stuck in local optimum with *seed 7*

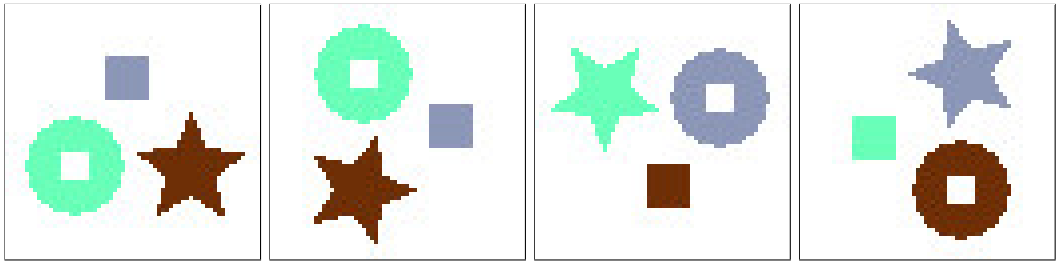
Figure 4.11: Blob chains found with different seeds

Hence, it is clear that for such an energy minimization task, mere hill climbing is not enough. The reader can see that if the system gets to a state shown in table 4.1, the move making algorithm will never find a way out because all swaps would increase the system's energy.

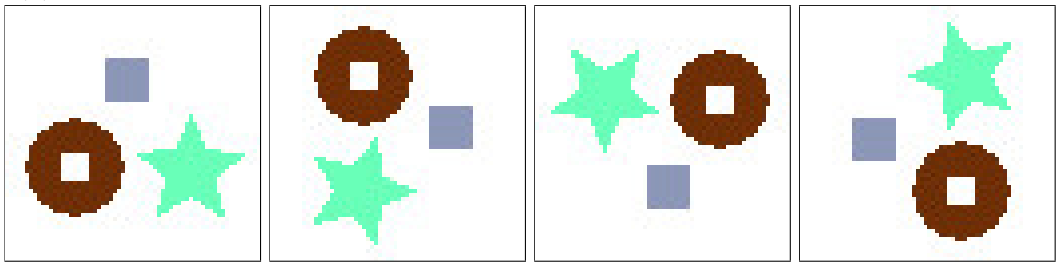
Table 4.1: State of the Local Optimum

	Blobs		
Key frame 0	R255; G000; B000	R000; G000; B255	R000; G255; B000
Key frame 1	R255; G000; B000	R000; G000; B255	R000; G255; B000
Key frame 2	R000; G000; B255	R000; G255; B000	R255; G000; B000
Key frame 3	R000; G000; B255	R000; G255; B000	R255; G000; B000

To overcome this problem, one could implement a metaheuristic such as the simulated annealing as supposed in section 3.1. However, by trying different seeds and monitoring the energy of the achieved state, it is also possible to discover the global optimum with the standard hill climbing optimization technique. In figure 4.12 blobs are matched by location (4.12a) and size (4.12b).



(a) Global optimum achieved with *seed 6* when matching blobs solely by location



(b) Global optimum achieved with *seed 0* when matching blobs solely by size

Figure 4.12: Blobs matched by location and size

The above results are just a proof of concept, showing that the proposed move making algorithm is suitable for blob matching. For any practical applications, the used hill climbing technique should be replaced by a more sophisticated method because manually trying different seeds for the random number generator is naturally an unwanted activity.

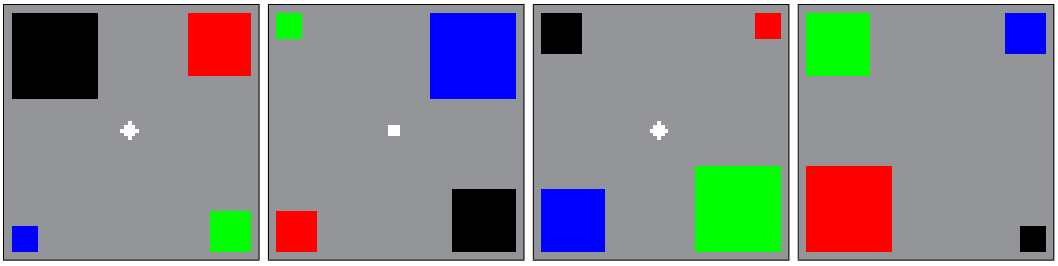
As this thesis is devoted to deliver a real practical application, the problem of local optimums is almost mandatory for the authors to solve. Although simulated annealing could be used here, the authors deem it a rather too general solution. What is more, simulated annealing comes with unintuitive initial parameters such as the system's temperature that need to be calibrated with trial and error.

Instead of implementing something complex, a simple enhancement is made to the hill climbing procedure. The best found solution is stored and occasionally the move making algorithm is allowed to make steps that increase the system's energy. The degeneration period is defined by the *degenerate* parameter of the demo program. Listing 4.14 shows the changes that were made to the standard hill climbing procedure to avoid getting stuck in local optimums. The reader can compare it with listing 4.12.

Listing 4.14: Enhancement to Hill Climbing

```
if (c1 >= c2 || (degenerate && (counter % degenerate) == 0)) {
    blob *buf       = blob_map[x2][y];
    blob_map[x2][y] = blob_map[x1][y];
    blob_map[x1][y] = buf;
    double gain     = c1 - c2;
    blob_map_e     -= gain;

    if (blob_map_e < 0.0) blob_map_e  = 0.0;
    if (blob_map_e <= best_e) {
        best_e          = blob_map_e;
        best_blob_map_e = best_e;
        counter         = 0;
        if (x1_volatile) {
            x1_y->x = (x2_y_prev->x + x2_y_next->x)/2.0;
            x1_y->y = (x2_y_prev->y + x2_y_next->y)/2.0;
        }
        if (x2_volatile) {
            x2_y->x = (x1_y_prev->x + x1_y_next->x)/2.0;
            x2_y->y = (x1_y_prev->y + x1_y_next->y)/2.0;
        }
        if (deviant) {
            for (size_t j=0; j<blob_map_h; ++j) {
                for (size_t i=0; i<blob_map_w; ++i) {
                    blob_map[i][j]->group = i;
                }
            }
        }
        else {
            blob_map[x1][y]->group = x1;
            blob_map[x2][y]->group = x2;
        }
        deviant = false;
    }
    else deviant = true;
}
```

The enhanced hill climbing procedure was tested using images shown in figure 4.13 as input. The first series of images is specifically designed to test blob matching by different combinations of feature vectors. The second series is chosen to demonstrate a typical use case where the application should match the blobs intuitively and without any problems.



(a) Squares with different features

(b) Attacking Cacodemon from *Doom (1993, id Software)*

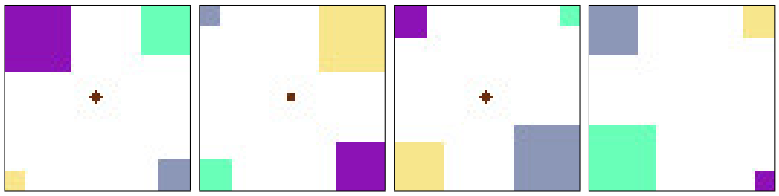
Figure 4.13: Images used to test blob matching

Listing 4.15 shows the arguments used to match the blobs in the previously shown images. It should be noted that the algorithm indeed no longer gets stuck in obvious local optimums. However, the *degeneration* parameter remains somewhat mysterious as changing it may dramatically affect the effectiveness of the algorithm. The authors were unable to find a good method for choosing that parameter automatically.

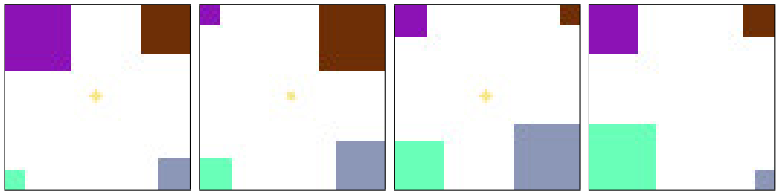
Listing 4.15: Command Line Parameters Used for Testing

```
./atomorph --blobs-as-distinct sq_1.png sq_2.png sq_3.png sq_4.png -F 4 -c 1 -p 0 -z 0
./atomorph --blobs-as-distinct sq_1.png sq_2.png sq_3.png sq_4.png -F 4 -c 0 -p 1 -z 0
./atomorph --blobs-as-distinct sq_1.png sq_2.png sq_3.png sq_4.png -F 4 -c 0 -p 0 -z 1

./atomorph -m 3 -t 80 -B 128 --blobs-as-distinct cd_1.png cd_2.png cd_3.png cd_4.png -F 4 --verbose
Loading data/cd_1.png        ... 80x80 image decoded.
Loading data/cd_2.png        ... 80x80 image decoded.
Loading data/cd_3.png        ... 80x80 image decoded.
Loading data/cd_4.png        ... 80x80 image decoded.
Blobifying 80x80 morph.
Matching blobs, energy is 13.851849.
Matching blobs, energy is 13.771829.
Matching atoms.
All done!
Detected 35 blobs on frame 3.
Process took 4 seconds to finish.
```

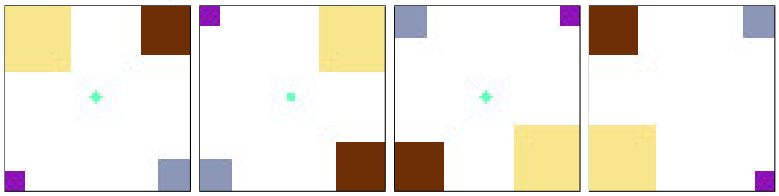
Figure 4.14 shows the resulting blobs gained from figure 4.13a. There are 3 series of output images because each of these was generated using a different feature vector for the distinguishing element. The reader might notice that even though the fourth frame lacks a white coloured blob for the central shape, the algorithm is still able to find the correct matches by having a volatile blob replacing the missing one.



(a) Blobs matched by their colour



(b) Blobs matched by their location



(c) Blobs matched by their size

Figure 4.14: Blobs matched by colour, location and size

Figure 4.15 shows the resulting blob chains gained from figure 4.13b. It is evident that the developed automatic blob matching works. However, for images that represent meaningful content, getting even a single blob wrong may render the algorithm worthless. Perhaps, the developed system could be used in combination with the interactive guidance of a human user. For now, these results are satisfactory and can be considered a proof of concept — to say the least.

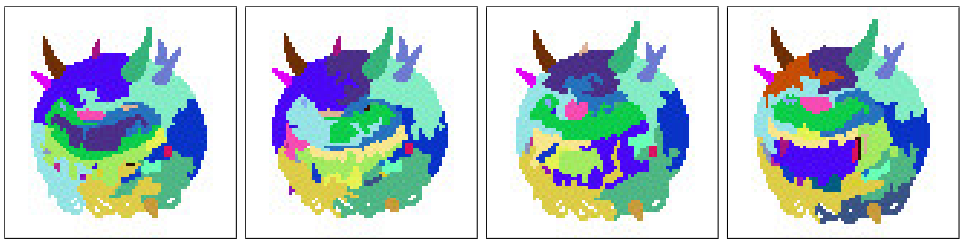


Figure 4.15: Blobs matched by all features

4.3 Atomic Morphing

In this section a novel atom cloud morphing method is revealed. Similarly to blob matching, the problem is formulated as an energy minimization task. Although atomic morphing is designed to be just one component of fluid morphing, its results alone are already very impressive.

4.3.1 Concept

Conventionally, shape morphing has always come to a point where researchers attempt to simplify the problem to edges instead of surfaces. This might be optimal for simple shapes but it is easy to see how edge based methods would terribly fail when having to morph binary images that contain noise as seen in figure 4.16.

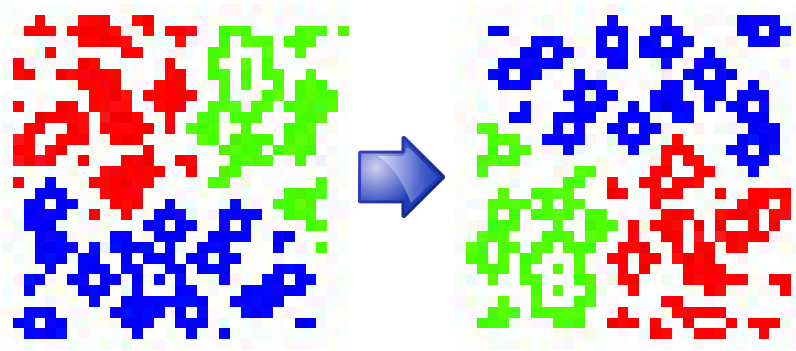


Figure 4.16: A transition is required between the noisy shapes

To solve this problem without the use of edges and polygons, the authors propose to convert 2-dimensional surfaces to point cloud data prior to any morphing. It is safely assumed that if these clouds are dense enough, they can later be rasterized as filled surfaces, thus making the conversion bidirectional (see figure 4.17).

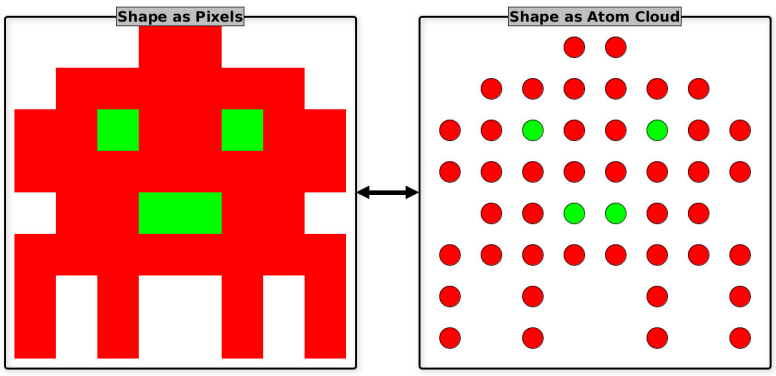


Figure 4.17: Pixels to atoms bidirectional conversion

Having the input images converted into atom clouds, a discrete mapping is needed between these clouds to express the transition. At this point, for computational simplicity, there is no possibility for the atoms to collide. The reader can think of it as the atoms having only the position but no size. To visualize the problem, all the atom clouds are superimposed in the same 2-dimensional space (see figure 4.18).

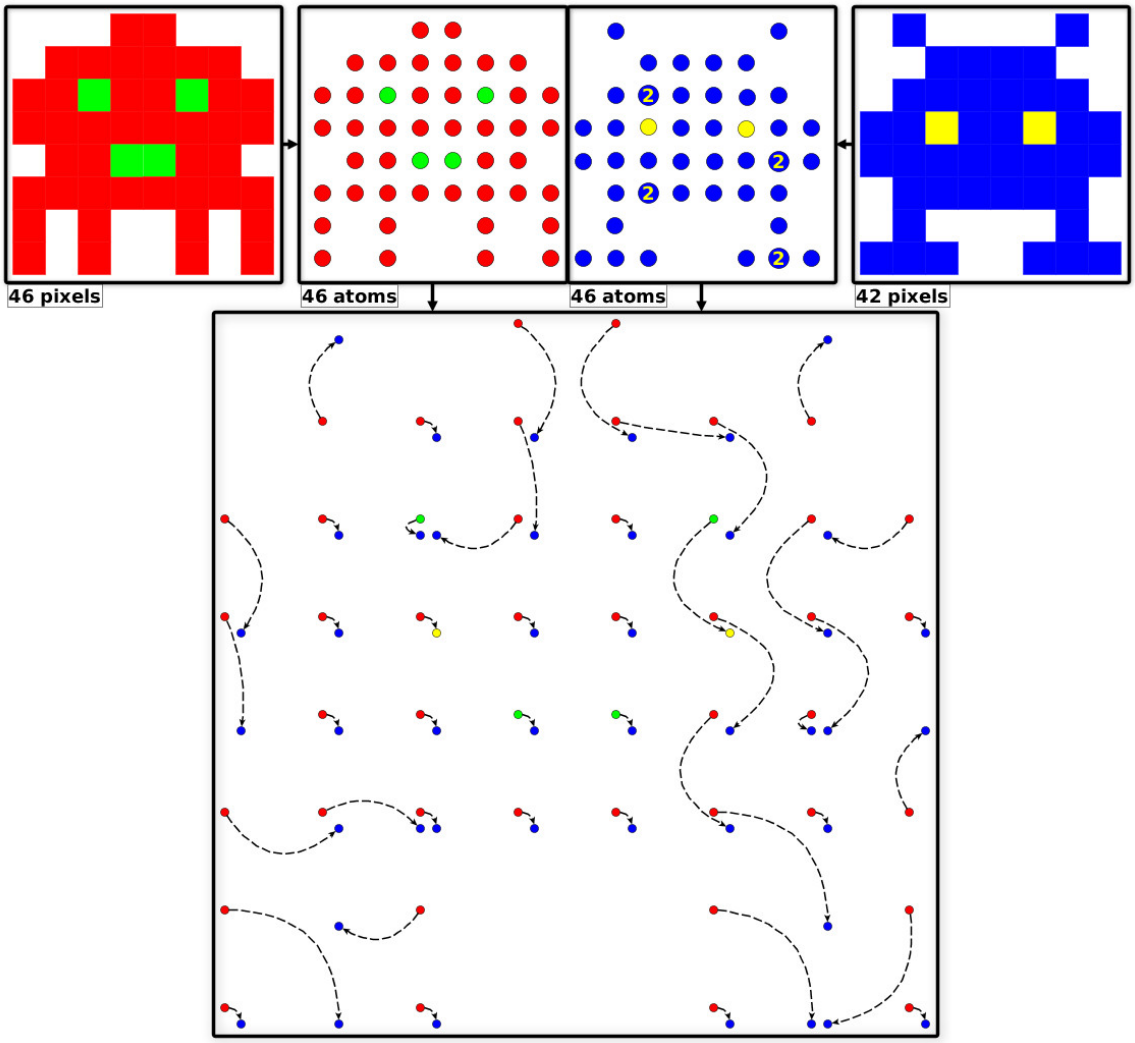


Figure 4.18: Matching the atoms for transition

The reader can see that the number of atoms is made equal for all the input images. The images that have less pixels will have multiple atoms placed on the same position. Such positions are chosen randomly. Thus, the minimal sufficient number of atoms per frame equals to the pixel count of the image with the largest surface area. Having said that, it is more convenient to call these atoms *key points* because input images are already known as *key frames*.

To actually find the best possible mapping between the key points, the problem is formulated as an energy function minimization task. First, it is assured that all the key frames of a morph have the same number of key points, duplicating some if needed. Then, a random mapping between the key points is chosen.

The energy function is defined to return a value that correlates positively to the total sum of distances between the corresponding key points. The energy is minimized with a move making algorithm by swapping randomly chosen correspondences within the same key frame if the swap would decrease the total energy of the morph.

The number of key points per frame is equal to the total number of *attractors* in the morph. Each attractor has exactly one key point defined per key frame. By connecting the best matching key points between different key frames, a trajectory for the attractor is gained. Thus, the interpolated position sets of attractors on their trajectories can be perceived as inbetween frames. In figure 4.19 attractors are shown moving on their interpolated smooth trajectory to visualize the morphing procedure.

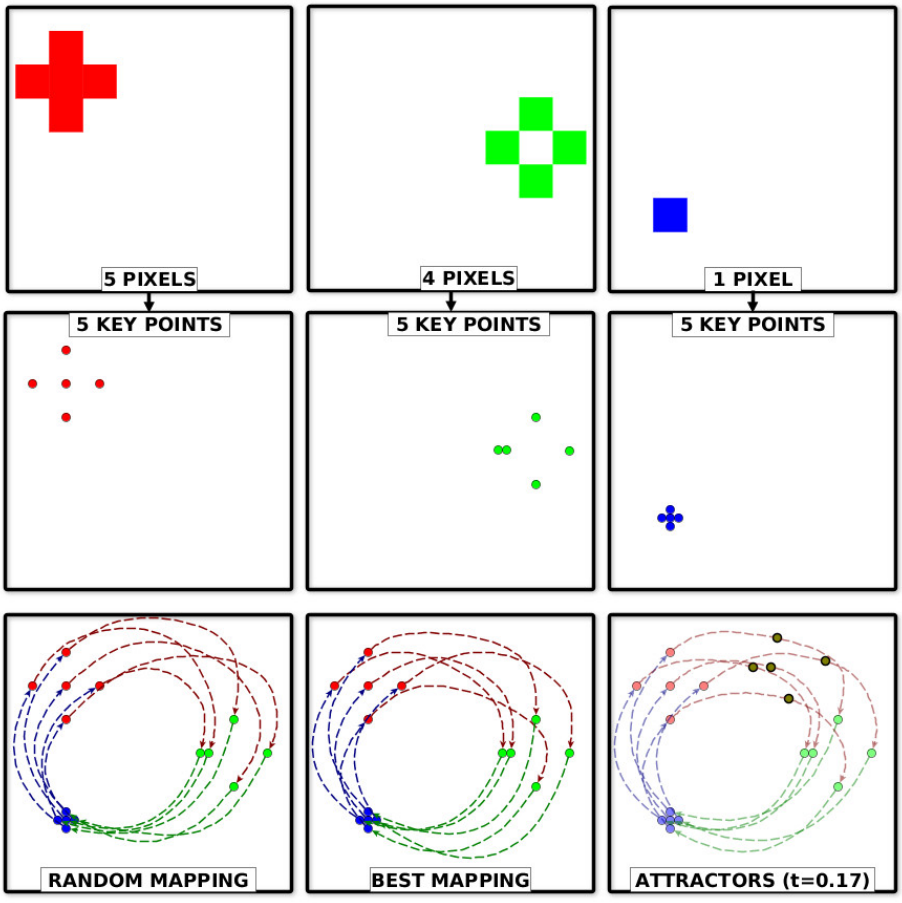


Figure 4.19: Attractors between the 1st and 2nd key frame

4.3.2 Algorithm

The process of matching key points is the most difficult part of atomic morphing. The problem is that the move making algorithm has to distinguish between good and bad swaps when according to the energy function they are considered equal.

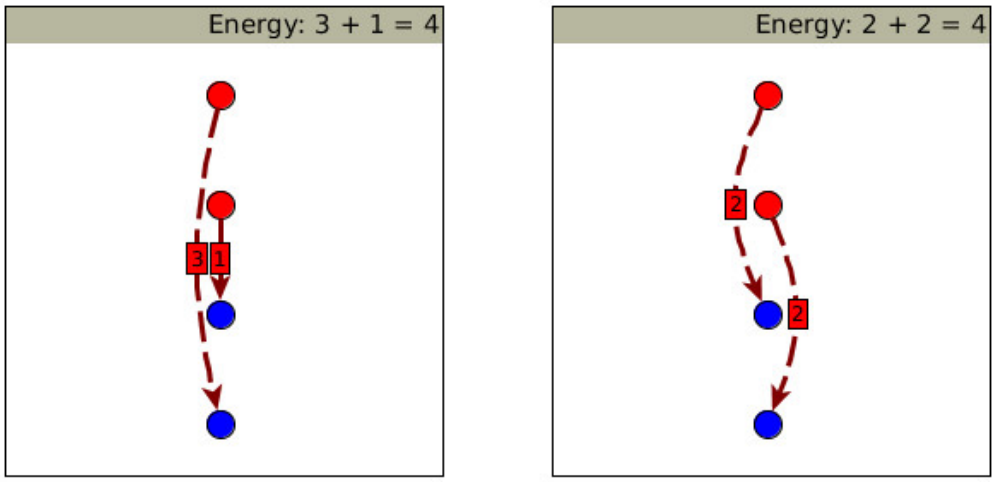


Figure 4.20: Dilemma when matching key points

For instance, in figure 4.20 there are two possible ways to detect corresponding key points. Because the sum of trajectory lengths is in both cases the same and there could be no collision between the attractors, morphs can easily become unintuitive (see figure 4.21).

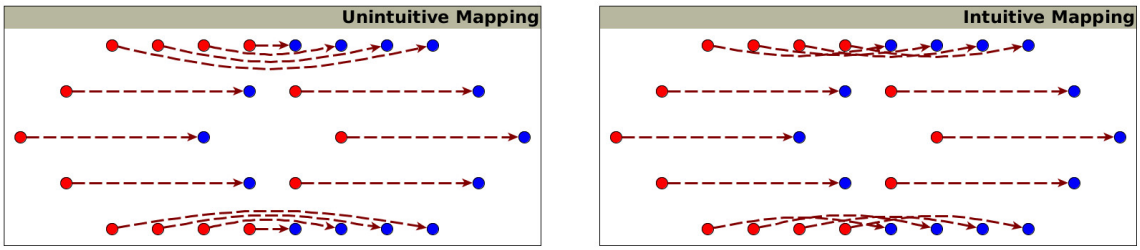


Figure 4.21: Different ways to match key points

The solution to this problem is fairly simple. When calculating the distance between the corresponding key points, a squared Euclidean distance should be used. There are two great things about that metric. It is faster to calculate than the "ordinary" distance and it places progressively greater weight on objects that are farther apart [12, p. 557]. By conducting a series of experiments, the most optimal C++ implementation of such a distance function was discovered (see appendix F.1).

In figure 4.22 a squared Euclidean distance is used to calculate the system's energy. In contrast to the dilemma shown in figure 4.20, this time the move making algorithm can easily determine the most intuitive match. The match that contributes most to the decrease of the system's energy is also the most intuitive one.

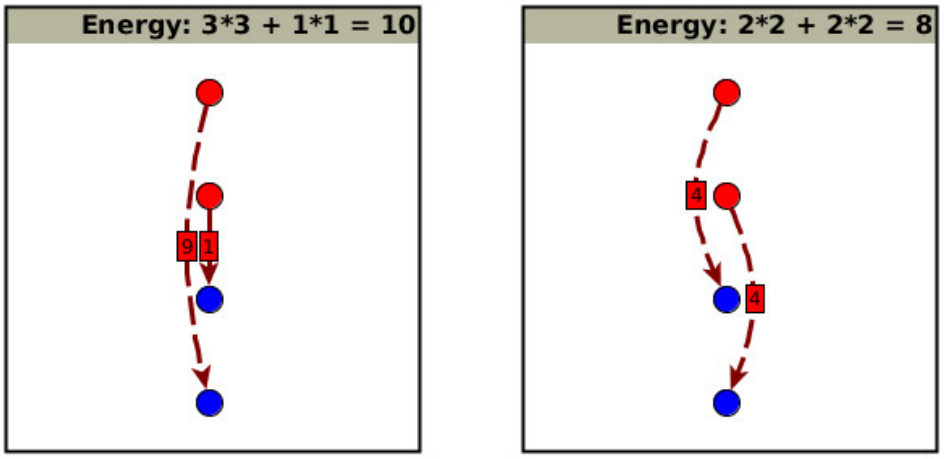


Figure 4.22: Using squared Euclidean distance to match key points

When the suitable mapping has been found, it becomes trivial to find all the inbetween frames. As said earlier, the interpolated positions of attractors on their trajectories captured at the same time can be perceived as an inbetween frame. The reader may ask whether linear interpolation is sufficient for this. It is not. In addition to linear interpolation, the authors have implemented the Catmull-Rom Spline (see section 3.2) because it gives much better results when the proposed morphing technique is used to express global movement (see figure 4.23).

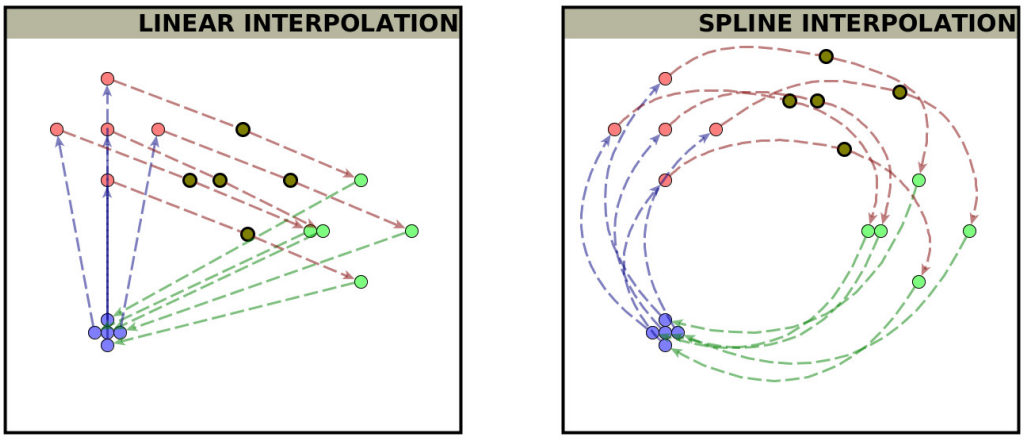


Figure 4.23: Linear vs. spline interpolation

The requirement of having the same number of key points in every key frame allows the solution search space to have a very simple structure. In other words, the key point matching takes place in a table structure (see figure 4.24), where swapping two cells of the same row is an atomic movement.

	KEY POINTS							
1. KEY FRAME:	x=0 y=0	x=1 y=0	x=0 y=1	x=1 y=1	x=10 y=10	x=11 y=10	x=10 y=11	x=11 y=11
2. KEY FRAME:	x=0 y=0	x=2 y=2	x=4 y=4	x=6 y=6	x=8 y=8	x=10 y=10	x=12 y=12	x=14 y=14
3. KEY FRAME:	x=6 y=6	x=7 y=6	x=5 y=6	x=6 y=5	x=6 y=7	x=5 y=5	x=7 y=7	x=7 y=5

Figure 4.24: Key points in a table structure

Now, to optimize this task for parallel computing, one could periodically redistribute the column indexes between the worker threads (see figure 4.25). If the redistribution is done randomly and sufficiently then all the key points will have an equal chance for being matched with all the other key points.

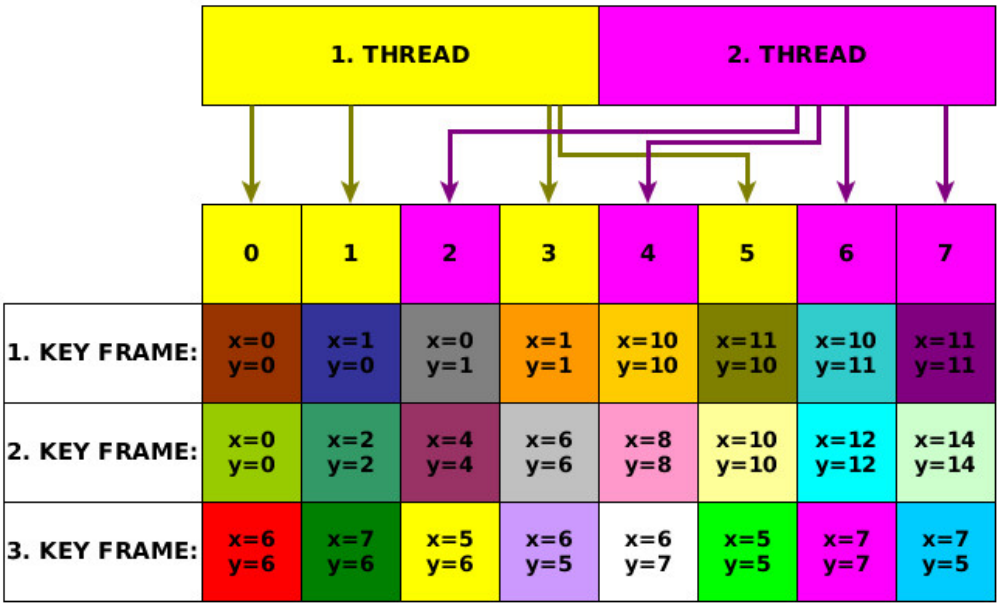


Figure 4.25: Using threads for key point matching

In figure 4.26 an example move is made to minimize the energy of the whole system. The first thread swaps the first and sixth key point in the first key frame and checks whether that move reduces the system's energy. Turns out that in the given example, the swap would make the system less optimal and thus, it must not be made.

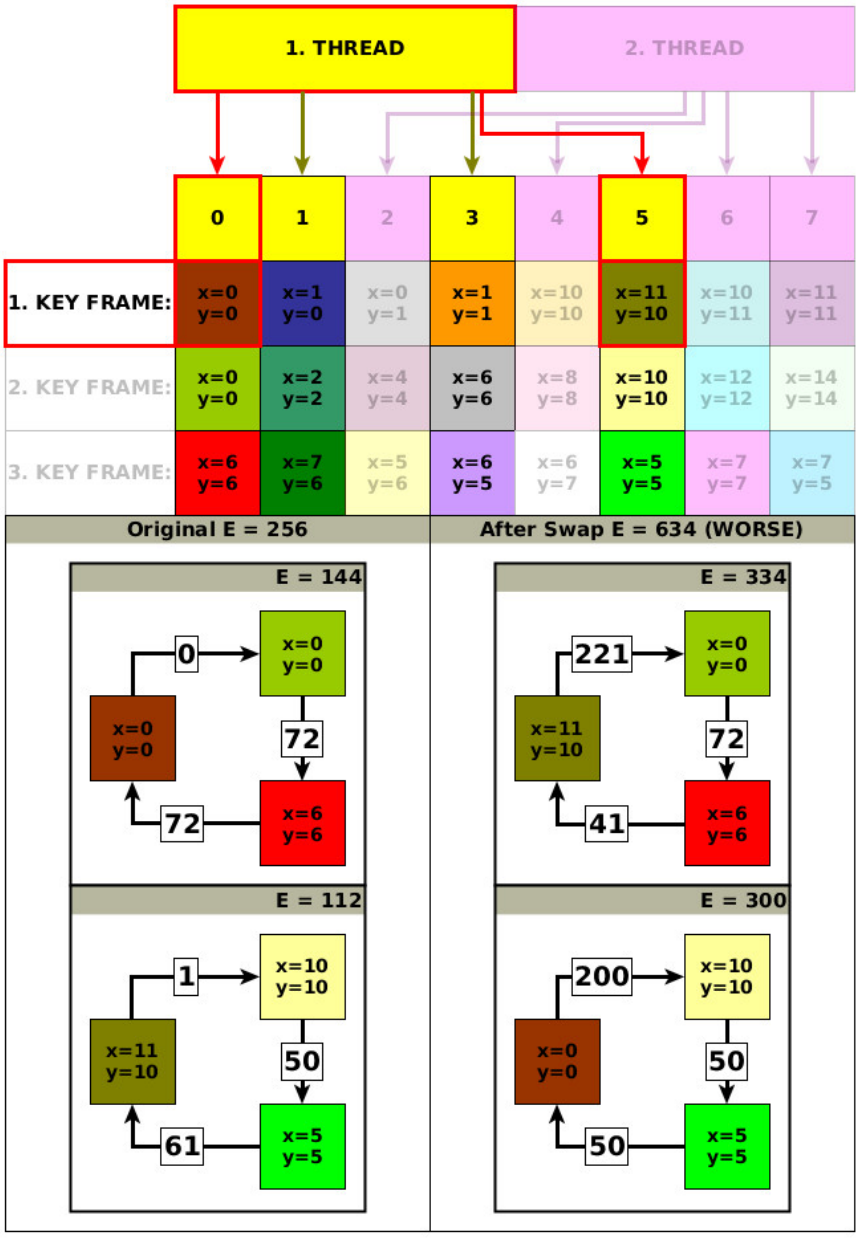


Figure 4.26: Example move for key point matching

4.3.3 Optimizations

To enhance the heuristic used in the hill climbing procedure, the authors came to an idea of using locality sensitive hashing. Instead of choosing a random pair of key points to be swapped, the algorithm would seek for the nearest key point in the succeeding key frame and construct a swap that has naturally a higher probability for lowering the system's energy.

One could argue that locality sensitive hashing carries the overhead of initializing the search tree. However, it is known that in the context of this work, key points never change. Thus, such a hashing would be naturally effective. What is more, atomic morphing does not need to know the absolute nearest neighbour, making it favourable to use an algorithm optimized for approximate nearest neighbour queries.

Searching the Internet revealed two popular C++ libraries designed for finding approximate nearest neighbours: ANN and FLANN. The latter was chosen because it is included in the OpenCV library by default. Also, FLANN (Fast Library for Approximate Nearest Neighbours) boasts about being much faster than any of the previously available approximate nearest neighbour search software.

After some testing on the OpenCV's interface to the FLANN library, a minor bug was encountered. The library accidentally printed some debug information into the standard output, making it irritating for the authors to proceed with the development of the AtoMorph library. Luckily, OpenCV is an open source library so that the authors were able to conveniently fix[2] the named issue.

Because OpenCV is a heavy weight library, it might not be wise to have mandatory dependencies on it. To solve this problem, the authors decided to include a new target in the project's *Makefile* which would enable the use of OpenCV. By default, *libatomorph.a* is compiled without any dependencies, being as portable as possible. However, for these experimental optimizations to take effect, AtoMorph should be compiled having the `ATOMORPH_OPENCV` macro defined.

After integrating OpenCV's FLANN to AtoMorph, the authors were shocked by the fact that it did not give any positive effect at all. The FLANN queries slowed down the move making algorithm more than 100 times when using KD-trees and more than 1000 times using the linear search. Given the same amount of computation time, the results were much better with random guessing than with the proposed FLANN optimizations. Perhaps when morphing high resolution images, FLANN would justify itself, but for now it is of no use.

[2]Contribution can be reviewed at https://github.com/Itseez/opencv/pull/2692.

4.3.4 Results

Videos 1 and 3 show the preliminary results of the AtoMorph library. They belong to a longer list that was originally published with [21]. In these videos, blob detection is not featured. However, in addition to purely atomic morphing, they display the use of median combining for noise reduction. The latter is not included in any of the final algorithms proposed in this thesis because it turned out to be unnecessary.

Figure 4.27 shows an animation[3] gained by morphing only 4 key frames using blob detection, blob matching and atomic morphing. The reader may see that 2 of the key frames completely lack blue pixels so that the algorithm is forced to use volatile blobs there. These volatile blobs — being sequential — are not trivial to position intuitively. However, with the algorithm shown in listing 4.16, the authors were able to solve this problem generally.

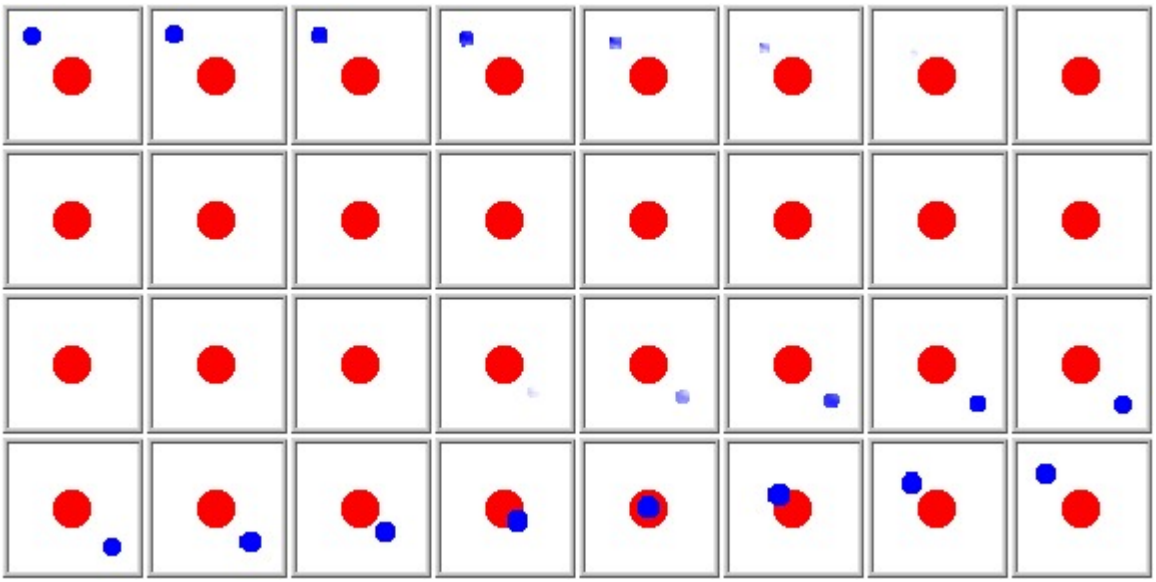


Figure 4.27: A morph of 4 key frames (leftmost column) featuring volatile blobs

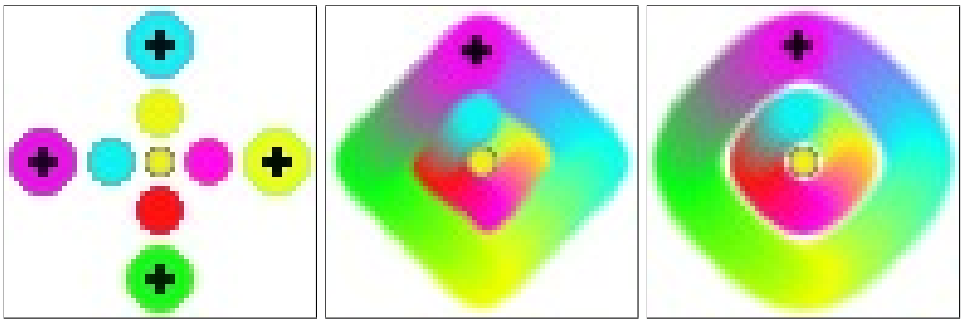
To further dissect the above results, it should be noted that when a blob enters the void, not only it shrinks into nothingness but it also turns transparent in the process. What is more, the sequential volatile blobs used here are automatically positioned at equal distances from each other. The latter makes it seem as if the blue circle disappears exactly at one side of the red circle and then reappears from nothingness right at the opposite side. Without equalizing the distances between volatile blobs, the blue circle would overlap the red one as it enters the void.

[3]See http://atomorph.org/img/test_volatile.gif for the animated version.

Listing 4.16: Positioning the Volatile Blobs

```
void thread::fix_volatiles (std::vector<blob *> *to_be_fixed) {
    size_t i, sz = to_be_fixed->size(); if (sz <= 1) return;
    bool started = false; std::vector<blob *> volatiles;
    blob *first_static = nullptr, *previous_static = nullptr;
    while (1) {
        i = (i+1)%sz;
        blob *bl = to_be_fixed->at(i);
        bool empty = bl->surface.empty();
        if (!started) {
            if (empty) {
                if (i == 0) break;
                continue;
            }
            started = true;
            first_static = bl;
            previous_static = bl;
            continue;
        }
        if (empty) {
            volatiles.push_back(bl);
            continue;
        }
        if (!volatiles.empty()) {
            size_t vsz = volatiles.size();
            for (size_t v=0; v<vsz; ++v) {
                double t = (v+1.0) / double(vsz+1.0);
                volatiles[v]->x = t*bl->x+(1.0-t)*previous_static->x;
                volatiles[v]->y = t*bl->y+(1.0-t)*previous_static->y;
            }
            volatiles.clear();
        }
        previous_static = bl;
        if (previous_static == first_static) break;
    }
}
```

Sharp-sighted reader may have noticed that in figure 4.27 the blue circle seems to be affected by slight inertia although it is not defined by the key frames. This effect originates from the Catmull-Rom splines (see section 3.2) that are used to interpolate the trajectories of the atoms. Different interpolation techniques are emphasized by a special purpose test of which results can be seen in figure 4.28.



No interpolation | Linear interpolation | Spline interpolation[4]

Figure 4.28: Flattened morphs using different interpolation techniques

[4]See http://atomorph.org/img/test_motion_spline.gif for the animated version.

Previously, an example of morphing key frames that have different number of blobs was given. This resulted in the introduction of volatile blobs to the scene. Next, in figure 4.29 the reader can see how AtoMorph handles the situation where some key frames are completely empty. In this case, the empty key frame will contain just the volatile blobs, having them positioned in a way to have the shortest possible trajectories.

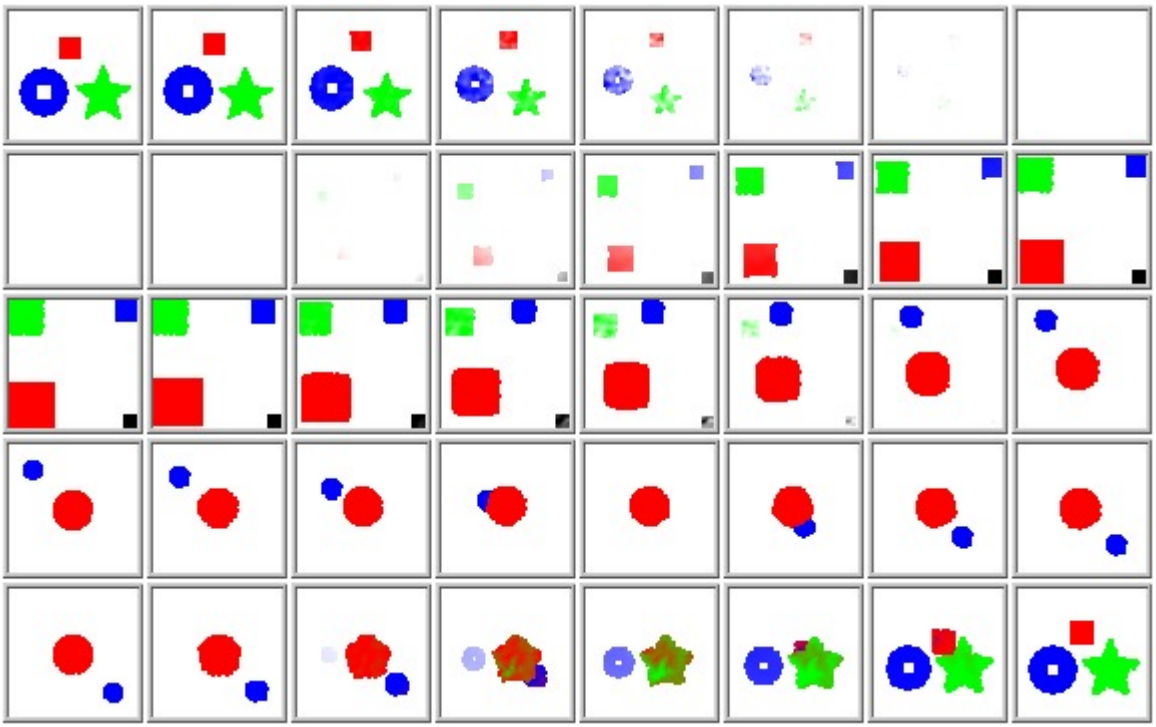


Figure 4.29: A morph[5] of 5 key frames (leftmost column) showing that empty input images are not a problem

Another interesting feature visible in this example is the Perlin noise dependent colour interpolation (see section 3.3 and video 2). This is achieved by first initiating two different Perlin noise functions — `lag_map(x, y)` and `slope_map(x, y)` — and then, when interpolating key point colours, the weight *f(t)* of the next key point is calculated from the equation 4.3, where L is lag and S is slope given by the Perlin noise functions (see figure 4.30).

$$s = \frac{S + 0.1}{1.1} \quad l = L \cdot (1 - s) \quad f(t) = \begin{cases} 0 & : t \leq l \\ 1 & : t \geq 1 + s \\ 0.5 \cdot (1 - \cos \frac{\pi \cdot (t-l)}{s}) & : \text{otherwise} \end{cases} \tag{4.3}$$

[5]See http://atomorph.org/img/test_empty.gif for the animated version.

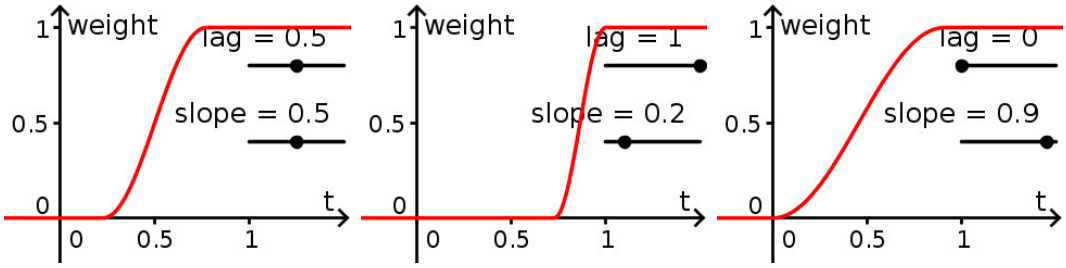


Figure 4.30: Cosine interpolation function at different parameters

To make the effect of Perlin noise dependent colour interpolation even more visible to the reader, figure 4.31 was created using the AtoMorph demo application's test suite. It is evident that for uniformly coloured shapes, Perlin noise definitely improves the morph by making it more artistic. Thus, one could even speculate that this is a perfect example of artificial imagination.

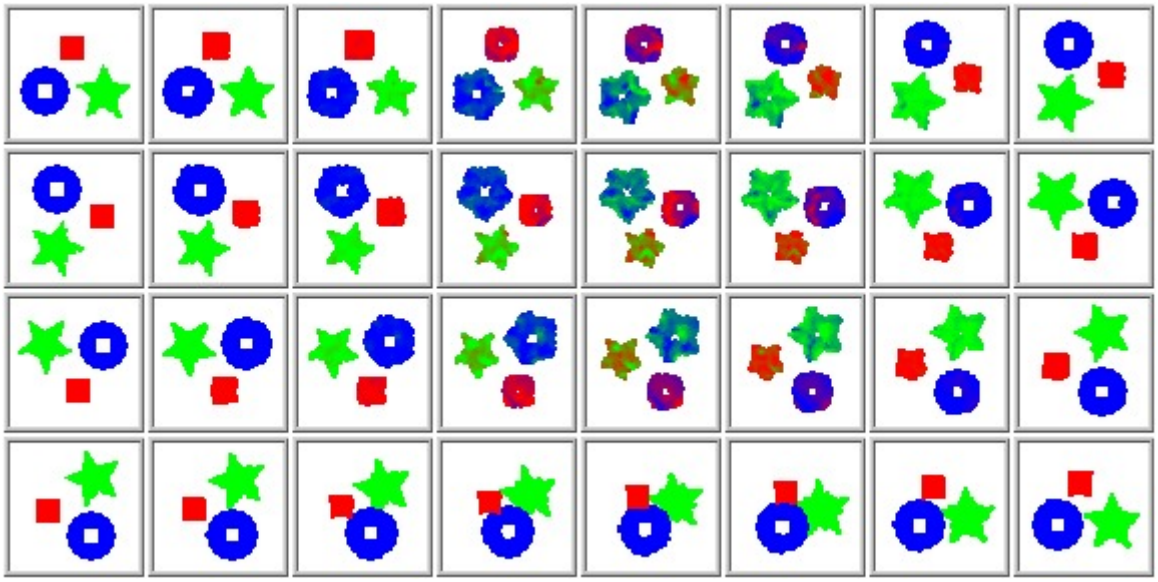


Figure 4.31: A morph[6] of 4 key frames (leftmost column) displaying Perlin noise

Another interesting test to highlight would be the intended rotation of a star (see figure 4.32). Although it seems intuitive that the result of such a morph would be a perfectly rotating green star, instead a strangely anomalous behaviour appears. It remains unknown what the exact reasons for this are, but the authors speculate that the algorithm either gets stuck in a local optimum or according to the defined energy function — the unintuitive morph is the global optimum.

[6]See http://atomorph.org/img/test_perlin.gif for the animated version.
[7]See http://atomorph.org/img/test_rotation.gif for the animated version.

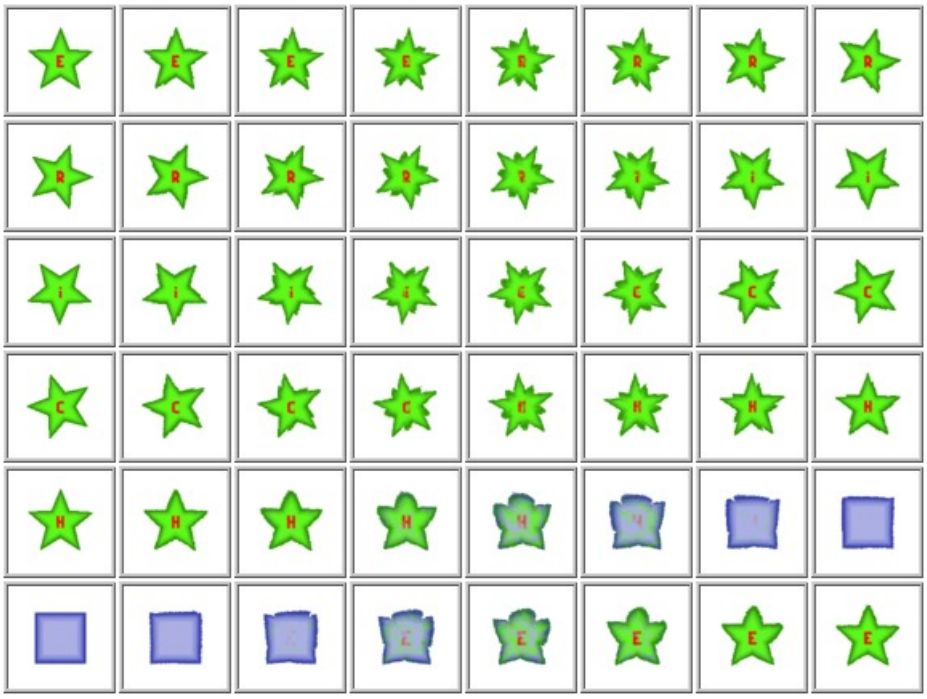


Figure 4.32: A morph[7] of 6 key frames (leftmost column) displaying rotation

Figure 4.33 displays a transformation of a car into a giant robot. The intent of such a morph is to show that AtoMorph is particularly good at finding morphs between unrelated images that share no common features. The authors believe that the human eye is very good at noticing preposterous transformations. Thus, whenever an intuitive morph exists, anything other than that could even anger the human observer.

Figure 4.33: A morph[8] of 2 key frames known from the movie *Transformers (2007)*

[8]See http://atomorph.org/img/test_transformer.gif for the animated version.

Figure 4.34 displays a long morph of 20 key frames. For game developers, such image morphing would allow crafting items by merging some of the existing items together. For example, a player would choose two of their favourite armours and produce a hybrid, having them morphed together in the shown fashion.

Figure 4.34: A morph[9] of 20 armours known from *Diablo (1996, Blizzard North)*

Other uses include enhancements to the interactive elements of an application's graphical user interface. Buttons, switches and similar GUI primitives could make great use of the proposed automatic morphing technique. Perhaps even the developers of ImageMagick would consider embedding AtoMorph in their software.

[9]See `http://atomorph.org/img/test_armors.gif` for the animated version.

Figure 4.35 shows a morph between the monsters of Doom — one of the most influential titles in the history of video games. Interestingly, some of the hybrid monsters look actually pretty artistic. AtoMorph could either be used by artists to discover new ideas for monsters or perhaps — by cross-breeding the corresponding images of existing monsters, procedurally generate a whole new monster.

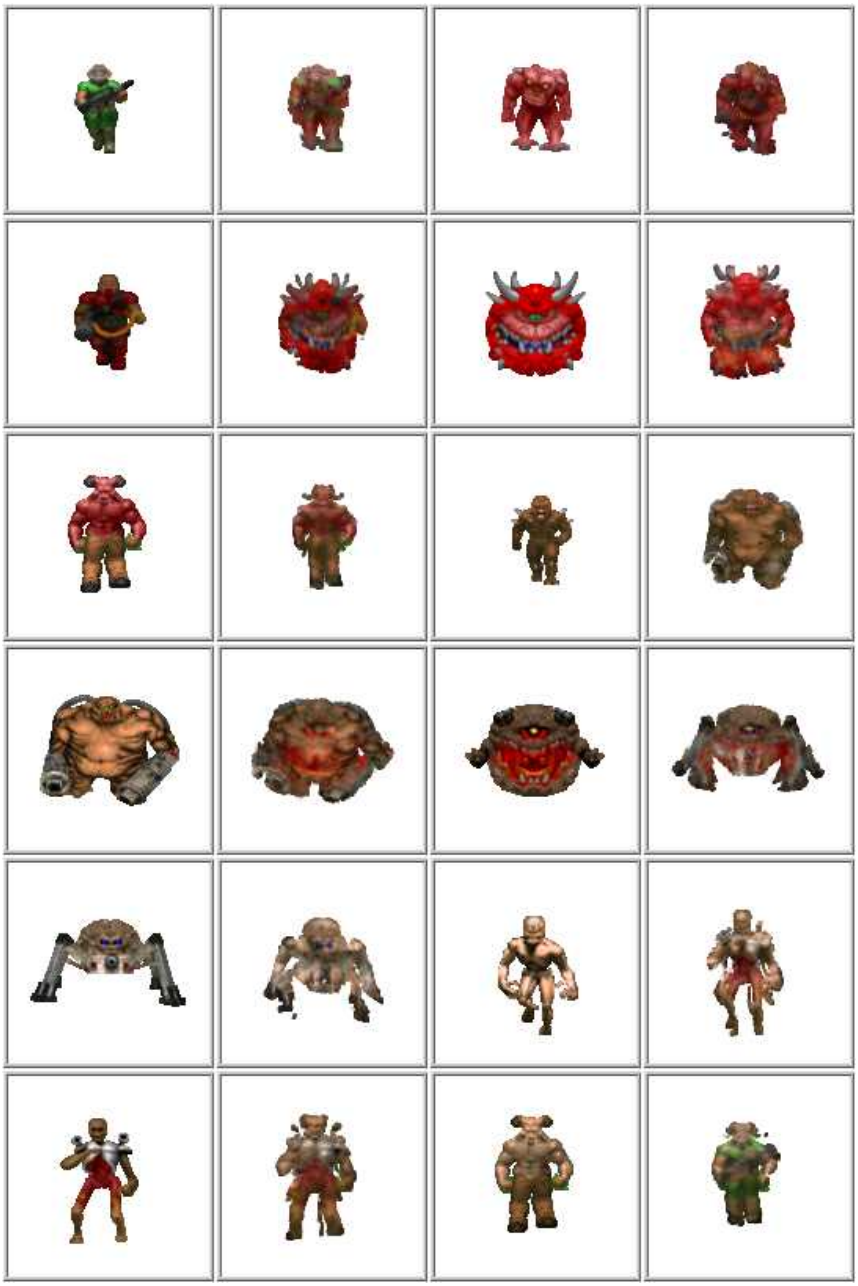


Figure 4.35: A morph[10] of 12 monsters known from *Doom (1993, Id Software)*

[10]See http://atomorph.org/img/test_monsters.gif for the animated version.

Figures 4.36, 4.37, 4.38 and 4.39 show morphs between different types of edibles. It should be noted about this example that it was not created by the authors of this work. The AtoMorph demo application was given to Aleksander Erstu — an independent digital artist — who used his own best judgement when choosing the images for testing out AtoMorph. It is remarkable that a non-programmer was actually able to compile and use AtoMorph, having received just a couple of brief instructions.

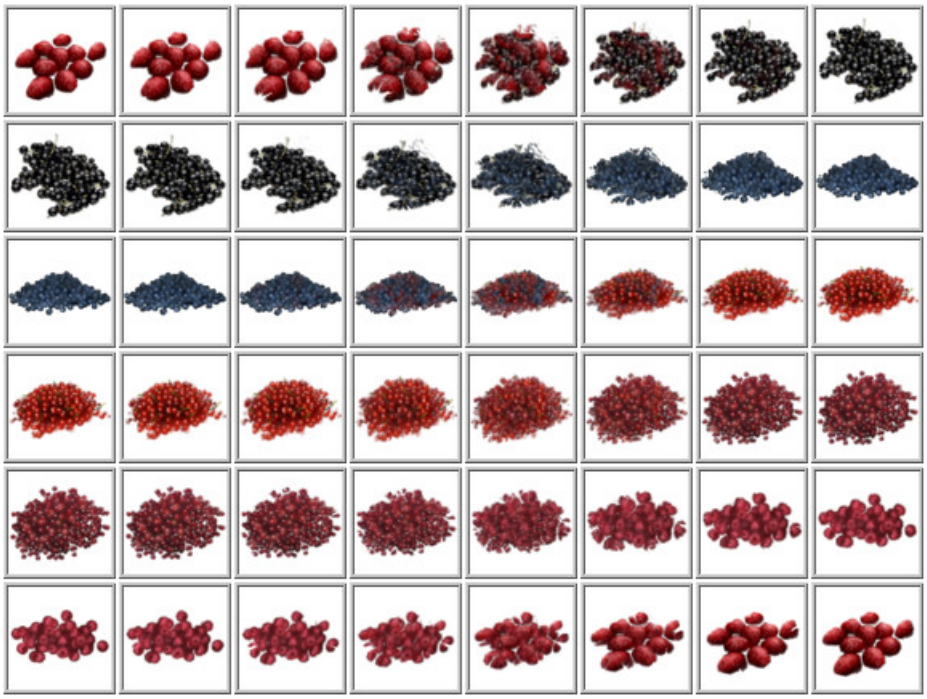


Figure 4.36: A morph[11] of 6 types of berries, the leftmost column being the key frames

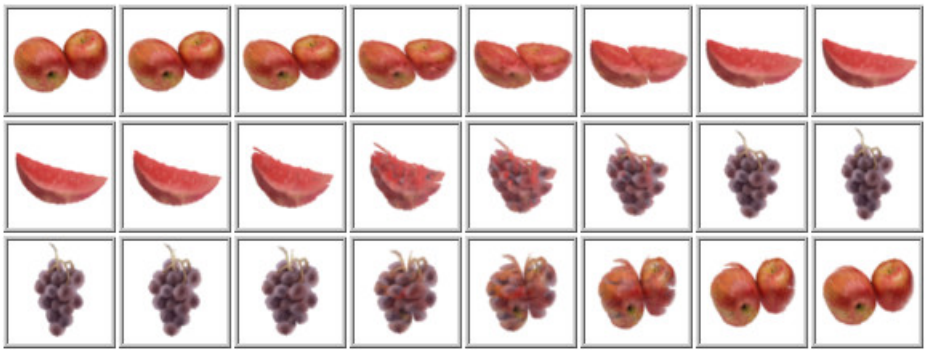


Figure 4.37: A morph[12] of 3 types of fruits, the leftmost column being the key frames

[11] See http://atomorph.org/img/test_berry.gif for the animated version.
[12] See http://atomorph.org/img/test_fruit.gif for the animated version.
[13] See http://atomorph.org/img/test_lettuce.gif for the animated version.

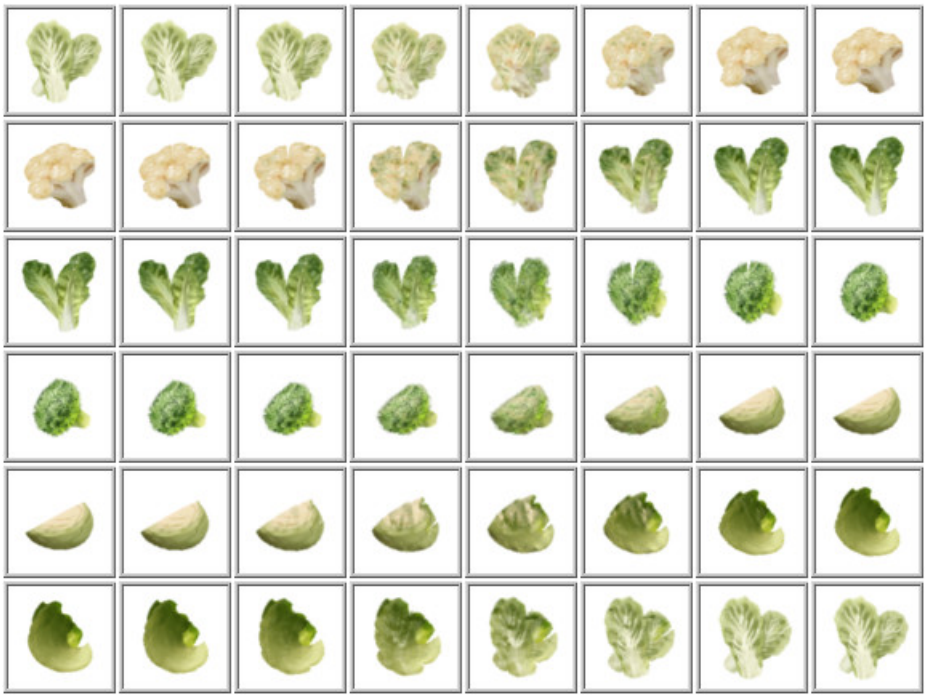


Figure 4.38: A morph[13] of 6 types of lettuces, the leftmost column being the key frames

Figure 4.39: A morph[14] of 6 types of spices, the leftmost column being the key frames

Figure 4.40 demonstrates that AtoMorph can be used to add inbetween frames to animated video game sprites. However, it should be noted that such morphs are not always easily performed. In the given example, the authors had to manually divide the key frames into logical segments that were each morphed separately and later flattened to produce the final animation. Such intervention is inescapable when morphing content that needs to be understood.

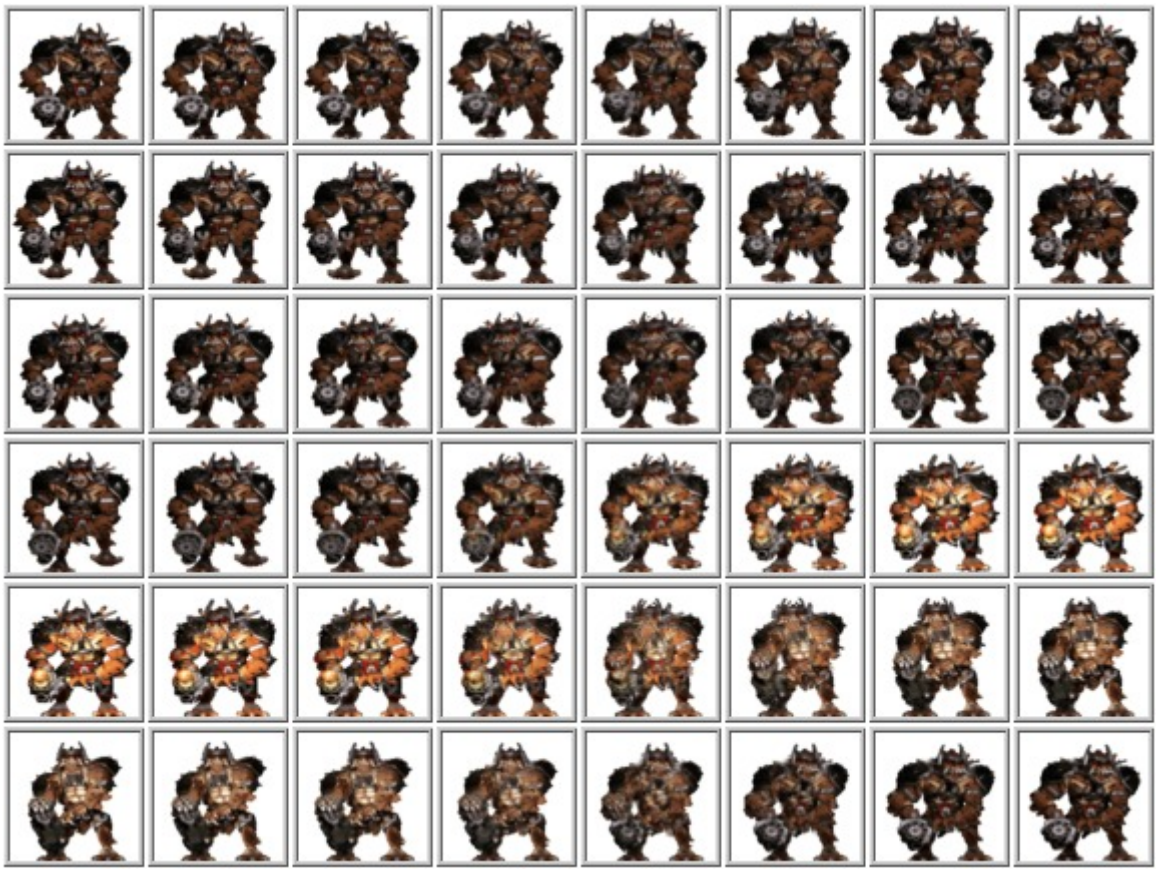


Figure 4.40: A morph[15] of 6 key frames (leftmost column) featuring the Battle Lord from *Duke Nukem 3D (1996, 3D Realms)*

The quality of morphs relies heavily on the matched blobs. Without manual blob matching, AtoMorph tends to produce unintuitive[16] results, whereas the absence of blobs inclines to cause disturbing pixel dust[17]. Neither of these problems is easily solvable in the proposed morphing framework. Nevertheless, the above figure proves the concept that with a bit of human assistance impressive results can still be achieved.

[14]See http://atomorph.org/img/test_spice.gif for the animated version.
[15]See http://atomorph.org/img/test_combine.gif for the animated version.
[16]See http://atomorph.org/img/test_battlelord_fun.gif for the unintuitive results.
[17]See http://atomorph.org/img/test_battlelord.gif for the pixel dust

Last but not least, a morph between two unrelated photos was carried out. In figure 4.41 a bouquet of flowers morphs seamlessly into a female face. The procedure includes blob detection and matching but there is still no collision between blobs. Instead, they are blended together as they overlap. Also, smooth blob edges are achieved by turning them gradually transparent and the occasional empty space between blobs is filled with the pixels from the weighted blend of the source images. The latter is gained by simply fading one image into another over time.

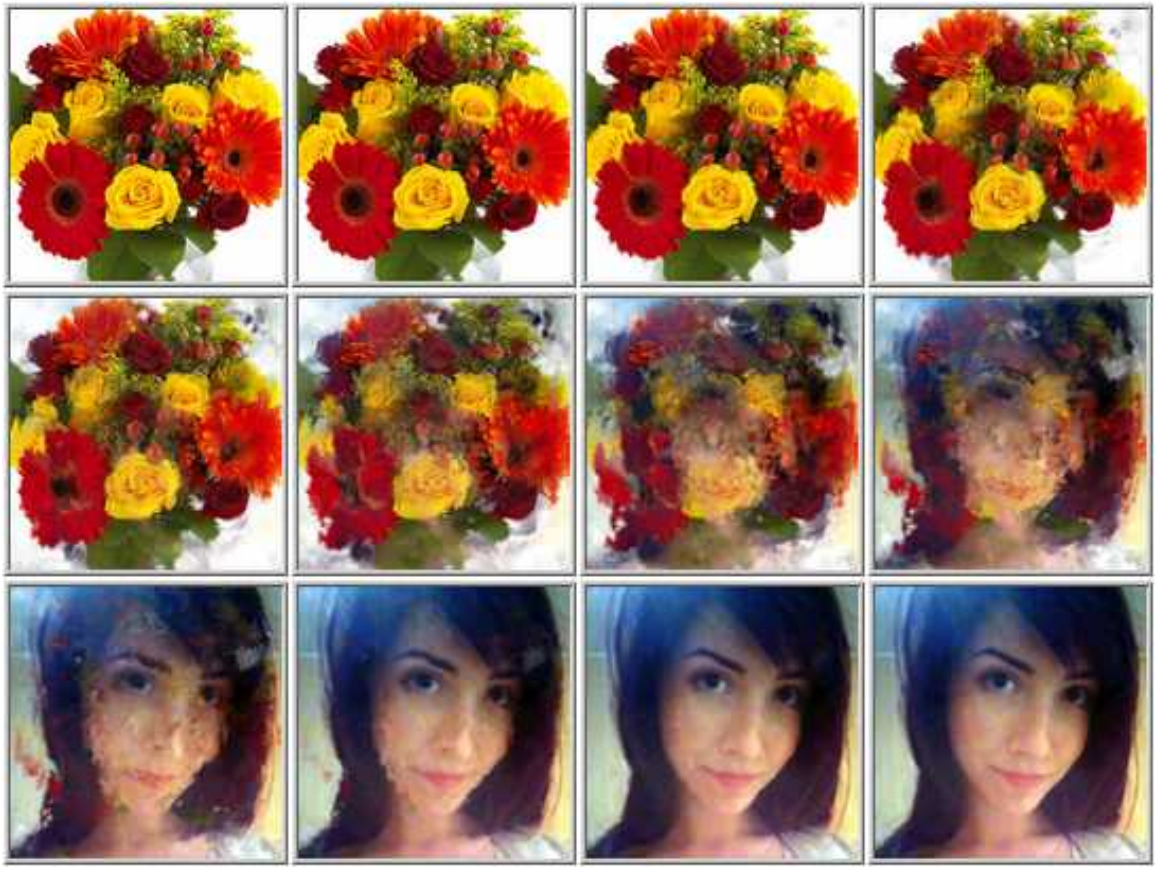


Figure 4.41: A morph[18] of two photos displaying unrelated content

So far, none of the results has really featured fluid simulation but instead mere point cloud morphing, ornate with gems from image processing. The essence of fluid morphing is yet to be discussed. In the next section, the reader is acquainted with the proposed uses of a fluid simulator. However, due to timely constraints, the named topic is rather briefly examined, delivering just the proof of concept.

[18]See http://atomorph.org/img/test_unrelated.gif for the animated version.

4.4 Fluid Simulation

In this section, the actual use of fluid dynamics is explained. It starts with the review of the original hypothesis [20] and some of the most apparent problems that need to be solved. Then, a technical solution to implementing fluid morphing is provided, followed by the amazing results achieved with the developed technique.

4.4.1 Hypothesis

To make any movement seem natural, a reasonable idea is to copy the nature itself. Behaviour of a liquid is compelling to watch. Therefore, it would make sense to use fluid physics when describing motion in computer graphics. By using a certain combination of methods, it could be possible to achieve some control over the simulated liquid.

Namely, Material Point Method — being a particle method — allows to easily track the fluid particles and assign custom variables such as colour to them. The method seems plausible for the task because it is performance oriented — suitable for real time rendering — and much appreciated in recent research such as [71].

To get the first impressions of the whole idea, the liquid in the fluid simulator is painted in a way to represent a typical sprite. Then, the simulator is started having the center of gravity in the center of the screen instead of the bottom. The result of this experiment is shown in figure 4.42.

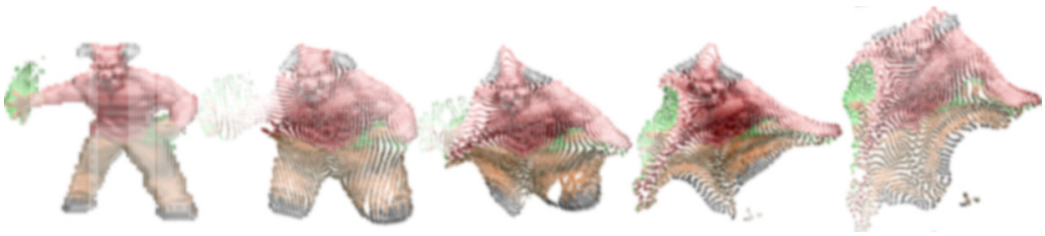


Figure 4.42: Baron of Hell from *Doom (1993, id Software)* as a puddle of liquid[19]

This proof of concept shows that it already is useful to paint the liquid to achieve simple transformations of a raster image. The next step would be for the liquid to flow into a predefined form while preserving the seamless transition of its texture. For a successful morph not only the shape of the image needs to smoothly transform but also the texture of that shape. The fact that fluid particles collide gives an idea that the proposed morphing algorithm could allow much more complex deformations than deemed possible with the conventional methods.

[19] See https://www.youtube.com/watch?v=kzDppwVF3hM for the animated version.

4.4.2 Problems

Variable Surface Areas

The first problem of fluid morphing is the need for the puddle surface area to have a variable size between key frames. For example, if the first form of a liquid is gained from 100 pixels and the next form only represents 50 pixels then during the flow the fluid must either be compressed or the particle count needs to decrease. Moreover, if the next key frame contains no pixels at all the fluid should dry up entirely or be compressed into an infinitely small area.

Flow Control

The second problem is the flow control. Even if the fluid is defined by its texture and placement in the sequential key frames, it remains a question of how exactly the fluid particles should find their individual trajectories for the transition to take place. It is known that there is collision between fluid particles, thus the trajectories cannot be precomputed in a reasonable way.

"Ghosting"

The third problem is the ghosting artefacts. For the fluid simulator, it means that it needs to distinguish between fluid particles that belong to different blobs in the original images. This is also the reason why compression of the fluid is not an appealing option when dealing with variable surface areas — sets of particles that represent different blobs could require different level of compression within the same simulation.

Rasterization

The last problem is the rasterization of a fluid. When drawing the fluid particles as single pixels on the screen, the fluid does not appear continuous by default. Instead, a particle swarm is seen as shown in figure 4.43.

Figure 4.43: Fluid drawn as particle swarm

4.4.3 Solution

To solve the problem of variable surface areas, the authors propose adding and removing fluid particles from the simulation accordingly to the changes in surface size. It is exactly known how many fluid particles should exist in the simulation to represent a key frame in its totality. Therefore, the number of fluid particles during transitions can be interpolated. If the system contains too few particles then new ones should be added. Otherwise, existing particles should disappear as shown in figure 4.44. The authors suggest that particles farthest to their destination should disappear when having excess number of particles in the simulation.

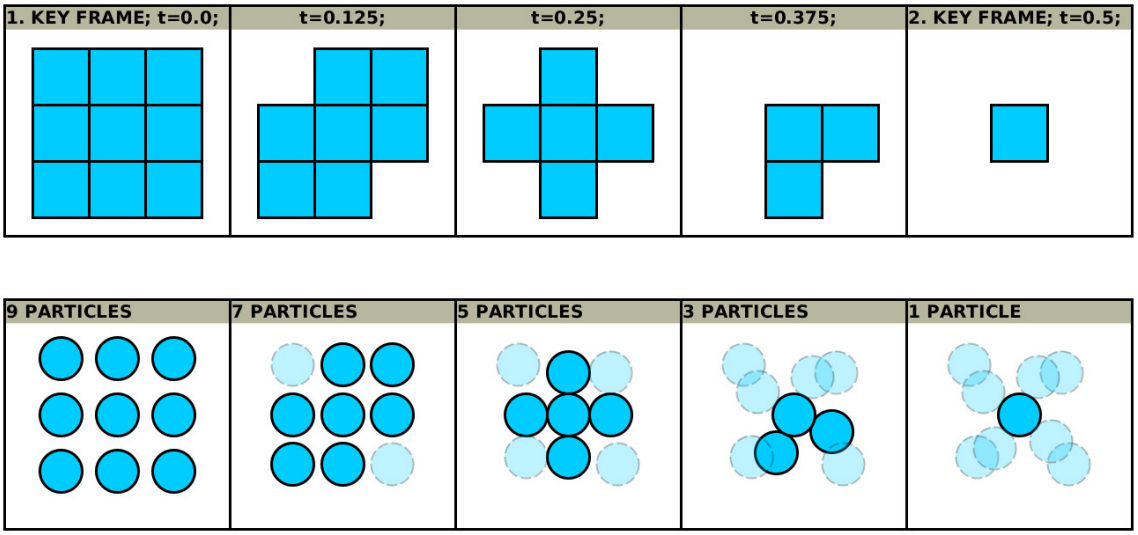


Figure 4.44: Decreasing surface area

Flow control can be achieved with the help of *attractors*. In fact, this is why they are called attractors in the first place — they are the attractors of the fluid particles. Although it can be argued that one attractor may have multiple followers amongst fluid particles, for the sake of simplicity, in this work it is assumed that a single attractor can either have one or zero active followers in the fluid simulation. Typically, an attractor loses its follower when the surface area gets too tight.

It is known that attractors have no collision, thus their position at any point of time can be interpolated from their individual key points. The authors propose that if the fluid particles are forced to gravitate towards their attractors, flow control can be achieved, sufficient enough to allow image morphing as a side product of the simulation. In figure 4.45, magenta arrow indicates the trajectory of an attractor and a red arrow points to the center of gravity for the attracted fluid particle.

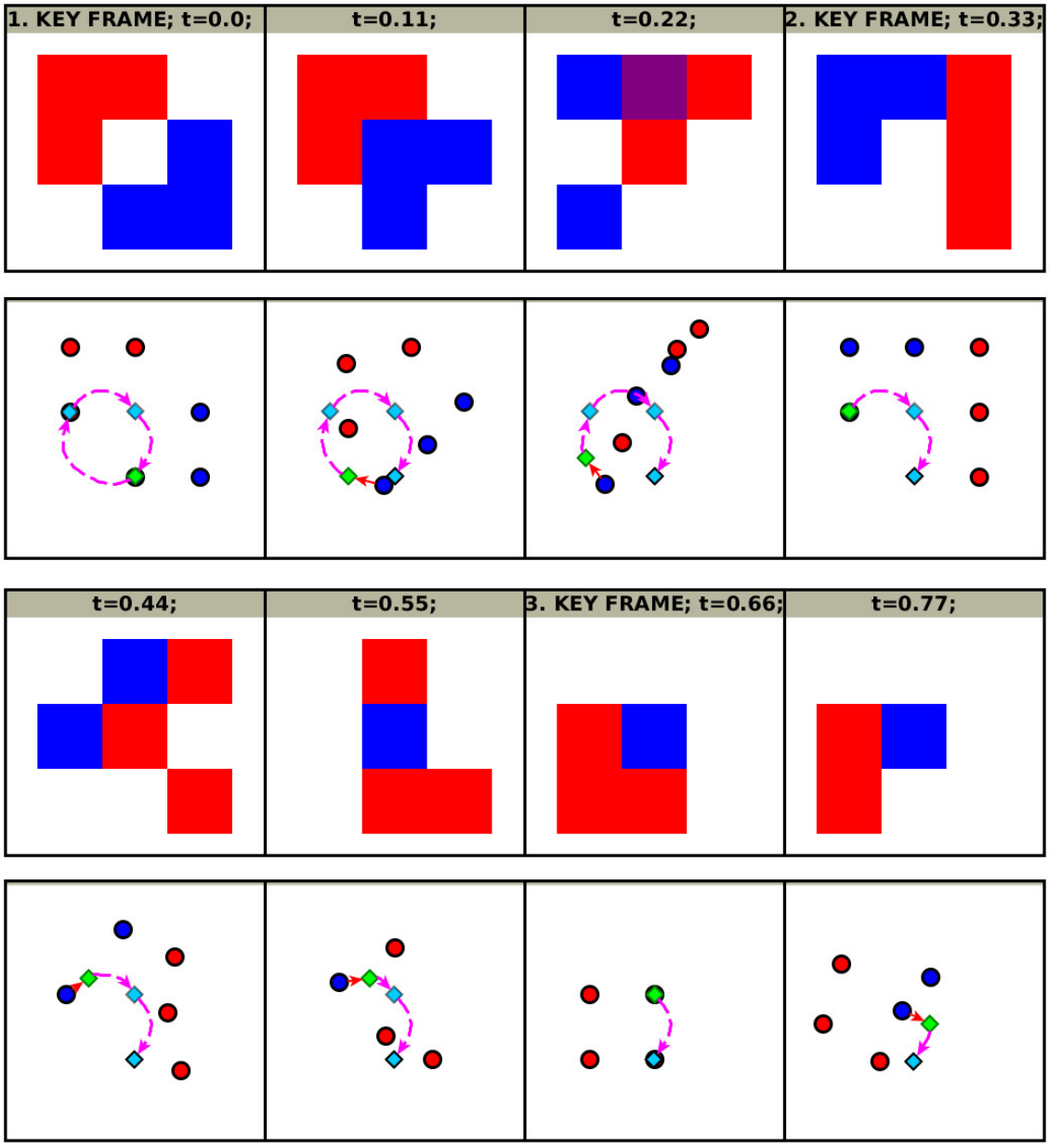


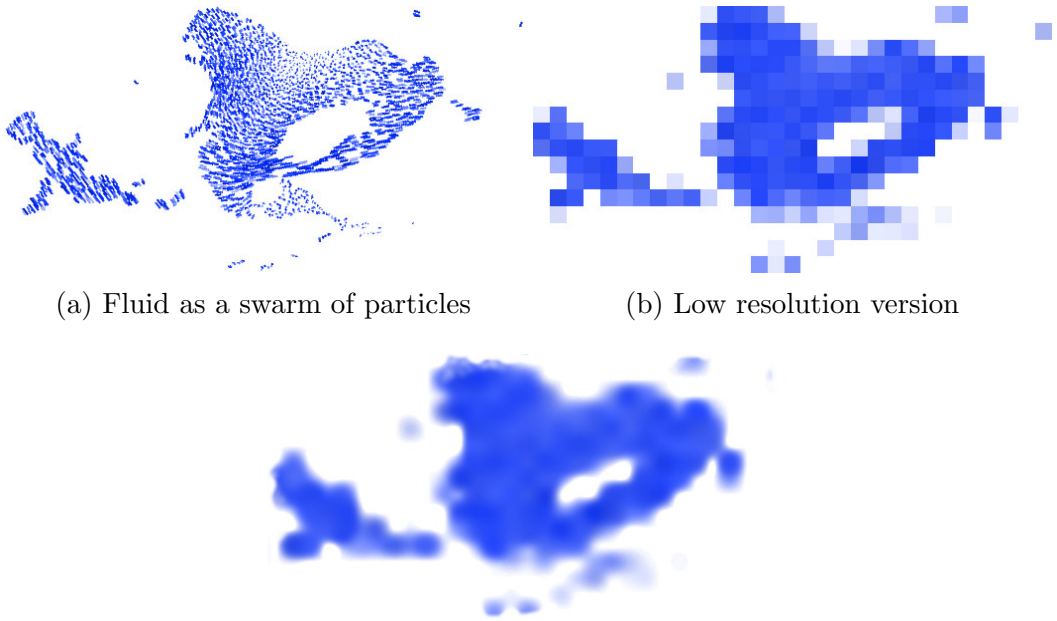
Figure 4.45: Fluid particles gravitating towards their attractors

To avoid the ghosting artefacts, input images are divided into rather uniformly coloured blobs. Similarly to key point matching, these blobs are also matched so that each blob would have a corresponding blob defined in each of the key frames. When matching blobs, their size, position and average colour should be taken into account.

It can happen that some blobs will not have a correspondence in every key frame. In such cases, during a morph towards the key frame where a correspondence is undefined, blobs should shrink into nothingness. In other words, a blob that needs to morph into nothing should simply dry up at its starting position.

Although there are many ways to render fluids, the authors recommend drawing the fluid particles as single pixels on a canvas which has a resolution low enough to result in an image of a continuous puddle. The reason for this is that it is the fastest and most simple approach. However, when speed and complexity is not an issue, one might prefer triangulation instead.

If multiple particles affect the same pixel, their colours should be blended. To have even greater effect, the blending could be weighted according to the fractional position of the fluid particles. For example, this approach is appreciated in [21]. Particles are first drawn to a low resolution bitmap which is then smoothly resized as needed (see figure 4.46).



(a) Fluid as a swarm of particles

(b) Low resolution version

(c) Blurred version

Figure 4.46: Rendering a fluid

4.4.4 Results

Having integrated a fluid simulator with the previously developed morphing primitives, the proposed fluid morphing method is complete and can be illustrated with figure 4.47. The reader should notice that in addition to the fluid like collision effects, the shapes also exchange colour during the interaction. The latter is achieved by blending the colours of the fluid particles that share the same neighbourhood in the Eulerian grid of the Material Point Method.

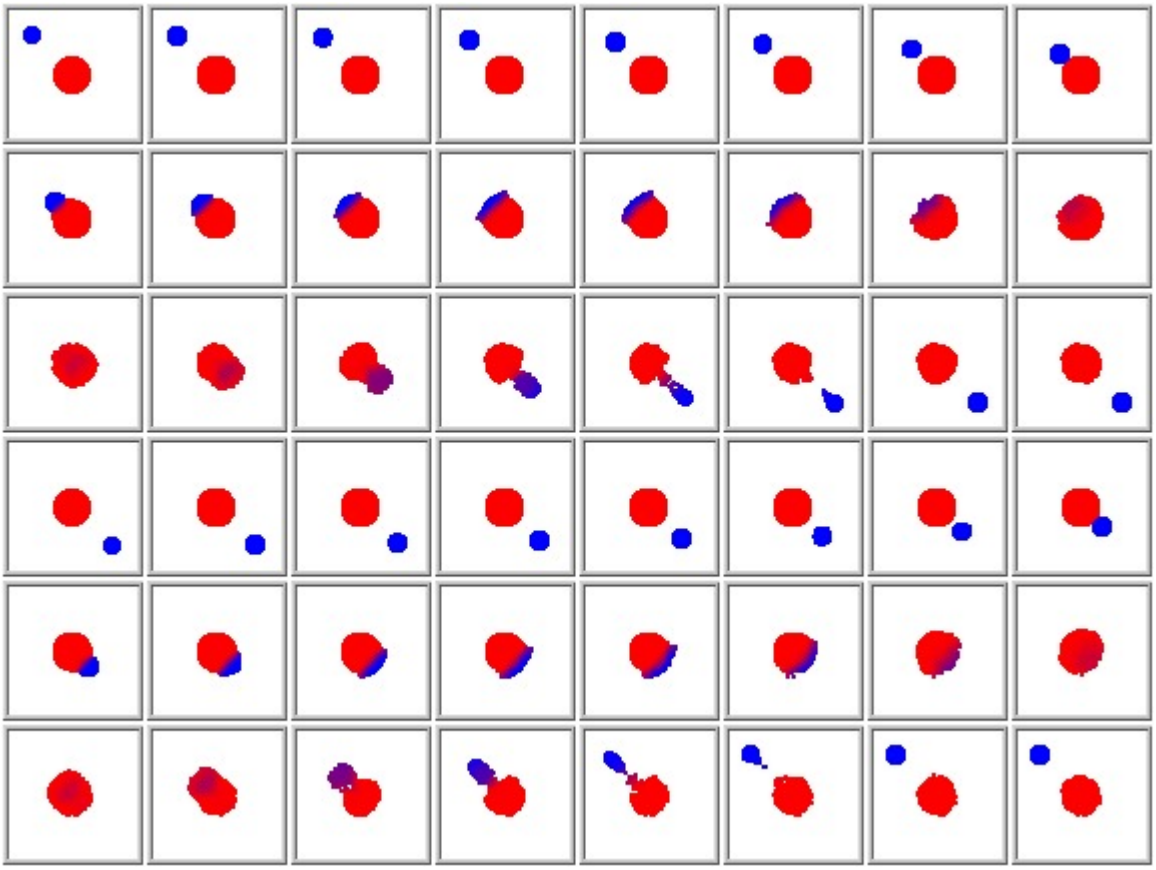


Figure 4.47: A morph[20] of 2 key frames using fluid dynamics

It should be emphasized that the results given in this work are solely a proof of concept. There are simply so many combinations of different parameters for setting up fluid morphing that it would be a research of its own to map these possibilities. For example, the above animation would be vastly different if the blobs were not created equal[21].

[20] See http://atomorph.org/img/test_fluid_simple.gif for the animated version.
[21] See http://atomorph.org/img/fluid_test_008.gif for the animated version.

To see where fluid dynamics really starts to matter, figure 4.41 should be compared with figure 4.49. Both of these examples have their blobs detected and matched the same way, except the latter features fluid dynamics, efficiently eliminating the notorious ghosting effect. It is highly advisable for the reader to see the animated versions of these examples.

For the reader who is eager to compare fluid morphing with the related state of the art methods, figure 4.48 was generated. The input images are taken from [66, p. 1] (see figure 2.14). It is obvious that fluid morphing outperforms Regenerative Morphing by producing less artificial looking results. What is more, fluid morphing has proven itself to be useful for discrete shape morphing, making it a superior method for fully automated morphing.

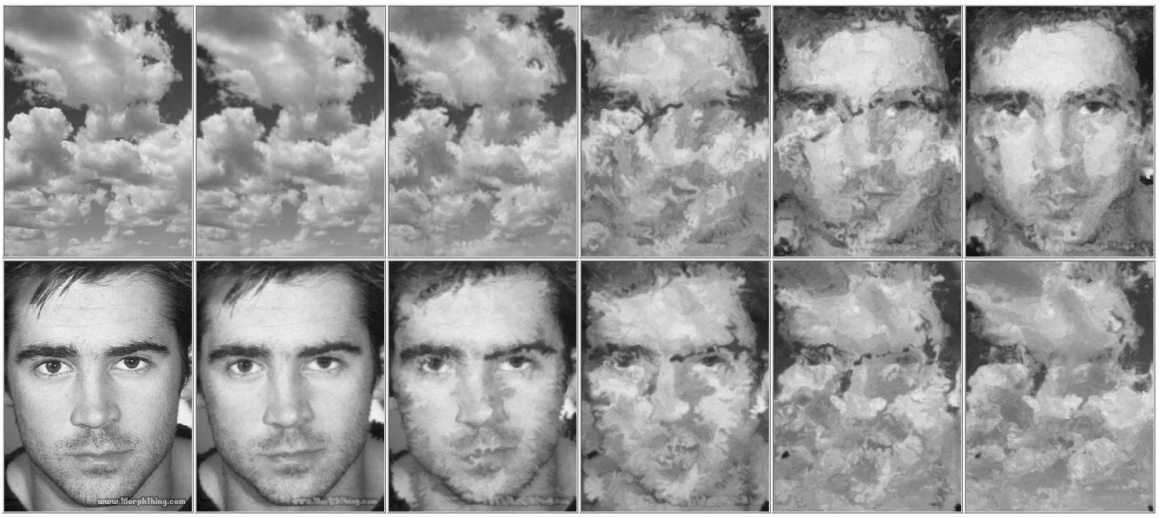


Figure 4.48: A fluid morph[22] of 2 unrelated images that were also used in Regenerative Morphing

That said, it is safe to conclude that the proposed fluid morphing method indeed works as the authors foresaw. The idea is now backed up with solid results and a free to use C++ library. Although much is yet to be tested out, the scope of this work is limited, leaving it for the reader to discover what other possibilities fluid morphing has to offer. In the next chapter, the final conclusion of this thesis is given.

[22]See http://atomorph.org/img/test_regenmorph.gif for the animated version.
[23]See http://atomorph.org/img/test_fluid.gif for the animated version.

Figure 4.49: A seamlessly repetitive fluid morph[23] between flowers and a female face

Chapter 5

Conclusions

In this thesis, a novel image morphing method was introduced. Not only did it explain the solution but also provided a solid C++ implementation. More than 3000 lines of code were written as an essential part of this work, making it highly recommended for the reader to acquaint themselves with appendix B. The tests included in the demo application provide explicit examples of how to use the library.

The authors were successfully able to implement an application capable of fully automated image morphing — something that has not been possible with the conventional methods. Moreover, it outperformed the state of the art by dramatically reducing the occurrence of the ghosting effects. When combined with human assistance, even the most complex morphs became conveniently possible.

The agglomerative hierarchical blob detection method turned out to give rather good results in terms of image segmentation (compare figures 4.5 and 2.6). Unfortunately, due to other priorities, the topic did not receive as much attention as it could have. Because the technique is essentially very simple yet effective, researchers in the field of image segmentation might want to further explore the capabilities of such hierarchical clustering.

By the visual results, atomic morphing was clearly more potent than Discrete Morphing [8]. Not only did AtoMorph generate smoother transformations of a shape but it managed to preserve its texture during the process. The closest competitor to the developed method was Regenerative Morphing [66]. However, it was left far behind Fluid Morphing when a comparison between their results was made — fluid flow is simply more pleasant to observe than what Regenerative Morphing does.

Having said that, the authors can gladly summarize that this thesis turned out to be a great success. Some of the readers might already have their own ideas for improving Fluid Morphing. To elaborate on that matter, suggestions for future research are given in the next section.

5.1 Future Research

Exhaustive Literature Study

A pedantic reader might have noticed that the included related work is not very strongly related to this thesis. It naturally raises some concerns, because after all – image morphing is intuitively a much researched topic, thus there has to be plenty of papers to review.

Yet, the authors, having spent a considerable amount of time on literature study, must conclude with regret that there are only a few of such papers freely available. The reason for this would be the fact that cartoon animation is a profitable business that relies heavily on computer technology. Even though there may be similar approaches already developed, these are likely to be only included in the high end commercial cartoon making software.

Nevertheless, it would not hurt to exhaustively study the literature that could be relevant to image morphing, possibly gathering references to papers that were left out from this thesis. For example, atomic morphing can be seen as optimal transportation of the pixels. Thus, even transportation theory might be worth researching.

Investigating a Bug in C++

Despite of the fact that contributions to the development of other software may not be seen as part of this thesis, the authors cannot deny the time they spent on researching issues that were not directly relevant to image morphing. For example, section F.2 demonstrates a bug that the authors found from the standard C++ library when developing AtoMorph.

The latter is of critical importance to the C++ developers all over the world because everyone assumes that a standard library has no bugs. For instance, having an X-ray generator returning *1.0* instead of *0.0* due to a bug in the standard library may have catastrophic consequences.

Mix of Different Fluids

Fluid particles could be given different parameters, making it possible to generate even more natural looking morphs. For example, if the morphed image sequence displays a rock moving in the water then the rock particles could be given greater stiffness than the water particles. Other properties such as density, viscosity, elasticity and gravity might also be worth exploration.

Graphics Engine Integration

Fluid morphing could be used in games as part of their graphics engine. If a game character is drawn on screen based on the state of its personal fluid sprite, natural looking deformations could occur on collision events between different game objects. It may be possible to develop this system so that the underlying fluid simulator would not have to contain the fluid sprites of all the game's objects concurrently. Instead, only the sprites visible on screen would be included in the fluid simulation.

Blob Detection Enhancements

Figure 4.5 reveals that depending on the seed for the random number generator, the results can sometimes be very intuitive when the number of blobs to detect is predefined. Using that knowledge, it might be possible to enhance the developed algorithm so that the generated hierarchical structure of the blobs would always be intuitive.

Instead of picking a random blob to expand, the algorithm could expand only the blob that would introduce the smallest change to its average colour. By doing this, it would make more sense to define the number of blobs at which the algorithm stops. The latter would make the colour threshold a less important parameter and thus the whole application easier to use.

N-way Fluid Morphing

By finding morphs for all the possible combinations of the key frames, it would be possible to present an interactive pose space of fluid sprites. That would allow pose space discovery similarly to N-way Image Morphing [6]. It would require finding a 2-way morph between all the key frames so that for n key frames there would be $\frac{(n^2-n)}{2}$ 2-way morphs. The inbetween frame in that pose space would then be simply a weighted average of all the generated 2-way morphs.

Aligning the Input Images

The developed blob matching technique considers the center of mass when matching blobs by location. However, key frames may have that vector vastly varying which sometimes results in an unwanted shift between sequentially matched blobs when superimposed. A theoretical fix for that would be taking the key frame's center of mass for the origin of its blobs' coordinate space. The latter would be somewhat similar to what [8] does.

Appendix A

Definition of Terms

Artificial imagination

Artificial simulation of human imagination by general or special purpose [64]. In the context of this work it represents means to interpolate animation frames in a way similar to a human artist's.

Attractor

An attractor is a moving point on the trajectory that passes through all the corresponding *key points* in the series of *key frames*. The purpose of attractors is to attract particles that may collide during their movement. For fluid morphing, these particles are the fluid particles. There is no collision defined between the attractors themselves.

Blobification

The procedure of blob detection applied to an image. In the context of this work, blobifying an image divides the image into uniformly coloured segments of nearby pixels.

Blob chain

A circular sequence of image patches across all the key frames, having exactly one representative blob per key frame. The length of a blob chain is the length of a closed path passing through the center of mass of each subsequent blob in the chain.

Closed spline

A spline that has a single point indicating its end and also its beginning. An example of a closed spline is given in figure 3.1.

Ghosting effect

Image artefact that originates from badly constructed deformations for the purpose of image morphing. Translucent parts of either of the source images appear and disappear during the morph in a fading effect (see figure A.1).

Figure A.1: Simple intensity blending generates a ghosting effect [66, p. 4]

Key frame

A key frame in animation and filmmaking is a drawing that defines the starting and ending points of any smooth transition. The drawings are called "frames" because their position in time is measured in frames on a strip of film. [17, p. 148]

Key point

A fixed and known point on the trajectory of a particle or set of particles. A trajectory goes through all of its key points. A key point in a frame is such a point that has exactly one corresponding key point in every remaining frame of the same animation.

Palette

A palette (colour palette) is given by a finite set of colours for the management of digital images [60].

Sprite

In computer graphics, a sprite is a two-dimensional image or animation that is integrated into a larger scene. Sprites are graphical objects that are handled separately from the memory bitmap of a video display. [26]

Subimage

A part of an image that is to be treated as a single object. In the context of this work a subimage is also a blob or a set of blobs that contain nearby pixels.

Volatile blob

A volatile blob is an empty blob that can have its location and average colour changed in order to reduce the energy of the blob matching system.

Appendix B

Source Code

This software was developed and tested only on Linux Mint 14 and 15 operating systems. Therefore, any problems that arise on other operating systems are left for the reader to solve. The reason for that would be the fact that the C++11 standard has not yet become widely spread amongst all the platforms.

AtoMorph Library

svn://ats.cs.ut.ee/u/amc/lib/atomorph

See listing B.1 for the suggested terminal commands.

Listing B.1: Downloading AtoMorph

```
$ sudo apt-get install subversion
$ svn checkout svn://ats.cs.ut.ee/u/amc/lib/atomorph@3089
```

After successfully checking out the revision 3089, the included *README* files should be viewed for build instructions and other comments.

Alternatively, AtoMorph Library can be downloaded from http://atomorph.org/atomorph-1.0-linux.tar.gz. To verify the archive's integrity, its `SHA256` hash should be the same as shown in listing B.2.

Listing B.2: Verifying the Integrity

```
$ sha256sum atomorph-1.0-linux.tar.gz
04f24cec5d0345b2832049ba7588e8bd0bbe84afc0dc0e223cfb843fd84a0901  atomorph-1.0-linux.tar.gz
```

Appendix C

Video Material

1. AtoMorph v0.1 (Morph of a cat):
   http://www.youtube.com/watch?v=YhanXnjEfaU
2. AtoMorph v0.5 (Perlin noise dependent interpolation):
   http://www.youtube.com/watch?v=ITBt3ev2MMk
3. AtoMorph v0.5 (Morph of a video game sprite):
   http://www.youtube.com/watch?v=6Ok_4ldcbHY
4. Humorous Phases of Funny Faces:
   http://www.youtube.com/watch?v=_Tn5sgHYQSc
5. N-way Image Morphing Demonstration:
   http://www.youtube.com/watch?v=cXU5zdJIRgc
6. Regenerative Image Morphing Demonstration:
   http://www.youtube.com/watch?v=NFsnVXSc1hg
7. Michael Jackson - Black Or White Official Music Video:
   http://www.youtube.com/watch?v=YpTNvUlwjFk
8. Brutal Doom v19 Trailer:
   http://www.youtube.com/watch?v=89iszJNcKQw

Appendix D

Catmull-Rom Spline

Listing D.1: spline.h

```
// ============================================================================
// Copyright Jean-Charles LAMBERT - 2007-2013
// e-mail:   Jean-Charles.Lambert@oamp.fr
// address:  Dynamique des galaxies
//           Laboratoire d'Astrophysique de Marseille
//           Ple de l'Etoile, site de Chteau-Gombert
//           38, rue Frdric Joliot-Curie
//           13388 Marseille cedex 13 France
//           CNRS U.M.R 7326
// ============================================================================
// See the complete license in LICENSE and/or "http://www.cecill.info".
// ============================================================================
#ifndef CATMULL_ROM_SPLINE_H
#define CATMULL_ROM_SPLINE_H
#include "vec3d.h"
#include <vector>

namespace glnemo {

class CRSpline
{
    public:
    CRSpline();
    CRSpline(const CRSpline&);
    ~CRSpline();

    void AddSplinePoint(const Vec3D& v);
    Vec3D GetInterpolatedSplinePoint(double t);  // t = 0...1; 0=vp[0] ... 1=vp[max]
    int GetNumPoints();
    Vec3D& GetNthPoint(int n);

    // Static method for computing the Catmull-Rom parametric equation
    // given a time (t) and a vector quadruple (p1,p2,p3,p4).
    static Vec3D Eq(double t, const Vec3D& p1, const Vec3D& p2, const Vec3D& p3, const Vec3D& p4);

    // Clear ctrl points
    void clearCPoints() { vp.clear();}

    private:
    std::vector<Vec3D> vp;
    double delta_t;
};
}
#endif
```

Listing D.2: spline.cpp

```
// ==========================================================================
// Copyright Jean-Charles LAMBERT - 2007-2013
// e-mail:   Jean-Charles.Lambert@oamp.fr
// address:  Dynamique des galaxies
//           Laboratoire d'Astrophysique de Marseille
//           Ple de l'Etoile, site de Chteau-Gombert
//           38, rue Frdric Joliot-Curie
//           13388 Marseille cedex 13 France
//           CNRS U.M.R 7326
// ==========================================================================
// See the complete license in LICENSE and/or "http://www.cecill.info".
// ==========================================================================
#include "spline.h"

namespace glnemo {

CRSpline::CRSpline() : vp(), delta_t(0.0) {}

CRSpline::CRSpline(const CRSpline& s) {
    for (int i = 0; i < (int)s.vp.size(); i++)
        vp.push_back(s.vp[i]);

    delta_t = s.delta_t;
}

CRSpline::~CRSpline() {}

// Solve the Catmull-Rom parametric equation for a given time(t) and vector quadruple (p1,p2,p3,p4)
Vec3D CRSpline::Eq(double t, const Vec3D& p1, const Vec3D& p2, const Vec3D& p3, const Vec3D& p4) {
    double t2 = t  * t;
    double t3 = t2 * t;
    double b1 = 0.5 * (    -t3 + 2.0*t2 -   t);
    double b2 = 0.5 * ( 3.0*t3 - 5.0*t2 + 2.0);
    double b3 = 0.5 * (-3.0*t3 + 4.0*t2 +   t);
    double b4 = 0.5 * (     t3 -     t2     );

    return (p1*b1 + p2*b2 + p3*b3 + p4*b4);
}

void CRSpline::AddSplinePoint(const Vec3D& v) {
    vp.push_back(v);
    delta_t = 1.0 / vp.size();
}

Vec3D CRSpline::GetInterpolatedSplinePoint(double t) {
    // Find out in which interval we are on the spline
    int p = (int)(t / delta_t);
    // Compute local control point indices
    int p0 = p - 1; p0 = (p0 < 0 ? vp.size()-1 : (p0 >= (int)vp.size() ? p0 - (int)vp.size() : p0));
    int p1 = p;     p1 = (p1 < 0 ? vp.size()-1 : (p1 >= (int)vp.size() ? p1 - (int)vp.size() : p1));
    int p2 = p + 1; p2 = (p2 < 0 ? vp.size()-1 : (p2 >= (int)vp.size() ? p2 - (int)vp.size() : p2));
    int p3 = p + 2; p3 = (p3 < 0 ? vp.size()-1 : (p3 >= (int)vp.size() ? p3 - (int)vp.size() : p3));
    // Relative (local) time
    double lt = (t - delta_t*(double)p) / delta_t;
    // Interpolate
    return CRSpline::Eq(lt, vp[p0], vp[p1], vp[p2], vp[p3]);
}

int CRSpline::GetNumPoints() {
    return vp.size();
}

Vec3D& CRSpline::GetNthPoint(int n) {
    return vp[n];
}

}
```

Listing D.3: vec3d.h

```
// ================================================================================
// Copyright Jean-Charles LAMBERT - 2007-2013
// e-mail:   Jean-Charles.Lambert@oamp.fr
// address:  Dynamique des galaxies
//           Laboratoire d'Astrophysique de Marseille
//           Ple de l'Etoile, site de Chteau-Gombert
//           38, rue Frdric Joliot-Curie
//           13388 Marseille cedex 13 France
//           CNRS U.M.R 7326
// ================================================================================
// See the complete license in LICENSE and/or "http://www.cecill.info".
// ================================================================================
#ifndef GLNEMOVEC3D_H
#define GLNEMOVEC3D_H
#include <math.h>
#include <iostream>
/**
    @author Jean-Charles Lambert <jean-charles.lambert@oamp.fr>
*/
namespace glnemo {

class Vec3D{
    public:
    ~Vec3D() {};

    double x, y, z;
    Vec3D( double InX, double InY, double InZ ) : x( InX ), y( InY ), z( InZ ) {}
    Vec3D( const Vec3D& V) : x( V.x ), y( V.y ), z( V.z )                      {}
    Vec3D( ) : x(0), y(0), z(0)                                                {}

    inline void set( const double InX, const double InY, const double InZ ) {
        x = InX; y = InY; z = InZ;
    }

    inline bool   operator== (const Vec3D& V2) const { return (x == V2.x && y == V2.y && z == V2.z); }
    inline Vec3D  operator+  (const Vec3D& V2) const { return Vec3D( x + V2.x,  y + V2.y,  z + V2.z);}
    inline Vec3D  operator-  (const Vec3D& V2) const { return Vec3D( x - V2.x,  y - V2.y,  z - V2.z);}
    inline Vec3D  operator-  (               ) const { return Vec3D(-x, -y, -z);                     }
    inline Vec3D  operator/  (const Vec3D& V2) const { return Vec3D (x / V2.x,  y / V2.y,  z / V2.z);}
    inline Vec3D  operator*  (const Vec3D& V2) const { return Vec3D (x * V2.x,  y * V2.y,  z * V2.z);}
    inline Vec3D  operator*  (double S       ) const { return Vec3D (x * S,  y * S,  z * S);         }
    inline Vec3D  operator/  (double S       ) const { double f=1.0/S; return Vec3D(x*f,y*f,z*f);    }
    inline double operator[] (int i          )       { return (i == 0 ? x : (i == 1 ? y : z));       }
    inline Vec3D& operator=  (const Vec3D& V2)       { x=V2.x; y=V2.y; z=V2.z; return *this;         }
    inline void   operator+= (const Vec3D& V2)       { x += V2.x; y += V2.y; z += V2.z;              }
    inline void   operator-= (const Vec3D& V2)       { x -= V2.x; y -= V2.y; z -= V2.z;              }

    inline double Dot( const Vec3D &V1 ) const {
        return V1.x*x + V1.y*y + V1.z*z;
    }

    inline Vec3D CrossProduct( const Vec3D &V2 ) const {
        return Vec3D( y * V2.z  -  z * V2.y,
                      z * V2.x  -  x * V2.z,
                      x * V2.y  -  y * V2.x );
    }

    Vec3D RotByMatrix( const double m[16] ) const {
        return Vec3D( x*m[0] + y*m[4] + z*m[8],
                      x*m[1] + y*m[5] + z*m[9],
                      x*m[2] + y*m[6] + z*m[10] );
    }

    // These require math.h for the sqrtf function
    inline double Magnitude( ) const {
        return sqrt( x*x + y*y + z*z );
    }

    inline double Distance( const Vec3D &V1 ) const {
        return ( *this - V1 ).Magnitude();
    }

    inline void Normalize() {
        double fMag = ( x*x + y*y + z*z );
        if (fMag == 0) return;

        double fMult = 1.0/sqrtf(fMag);
        x *= fMult;
        y *= fMult;
        z *= fMult;
        return;
    }
};

}

#endif
```

Appendix E

Perlin Noise

Listing E.1: perlin.h

```
/*
 * See Copyright Notice at the end of this file.
 */

class PerlinNoise {
    public:
    PerlinNoise( unsigned seed = 1 );

    double noise( double x           ) const { return noise(x,0.0,0.0); }
    double noise( double x, double y ) const { return noise(x,y,0.0);   }

    double noise( double x, double y, double z ) const;
    double octaveNoise( double x, int octaves ) const;
    double octaveNoise( double x, double y, int octaves ) const;
    double octaveNoise( double x, double y, double z, int octaves ) const;

    private:
    double fade( double t                    ) const { return t*t*t*(t*(t*6-15)+10); }
    double lerp( double t, double a, double b ) const { return a + t * (b - a);       }

    double grad( int hash, double x, double y, double z ) const;
    int p[512];
};

/*
The MIT License (MIT)

Copyright (c) 2013 Reputeless

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
*/
```

Listing E.2: perlin.cpp

```
/*
 * See Copyright Notice in perlin.h
 */
#include <cmath>
#include <array>
#include <numeric>
#include <random>
#include <algorithm>
#include "perlin.h"

PerlinNoise::PerlinNoise( unsigned seed ) {
    if(seed==0) seed = std::mt19937::default_seed;
    // p[0]..p[255] contains all numbers in [0..255] in random order
    std::iota(std::begin(p),std::begin(p)+256,0);
    std::shuffle(std::begin(p),std::begin(p)+256,std::mt19937(seed));
    for(int i=0; i<256; ++i) p[256+i] = p[i];
}

double PerlinNoise::noise( double x, double y, double z ) const {
    const int X = static_cast<int>(::floor(x)) & 255;
    const int Y = static_cast<int>(::floor(y)) & 255;
    const int Z = static_cast<int>(::floor(z)) & 255;

    x -= ::floor(x);
    y -= ::floor(y);
    z -= ::floor(z);

    const double u = fade(x);
    const double v = fade(y);
    const double w = fade(z);

    const int A = p[X ]+Y, AA = p[A]+Z, AB = p[A+1]+Z;
    const int B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;

    return lerp(w, lerp(v, lerp(u, grad(p[AA ], x , y , z ),
                   grad(p[BA ], x-1, y , z )),
                   lerp(u, grad(p[AB ], x , y-1, z ),
                   grad(p[BB ], x-1, y-1, z ))),
                   lerp(v, lerp(u, grad(p[AA+1], x , y , z-1 ),
                   grad(p[BA+1], x-1, y , z-1 )),
                   lerp(u, grad(p[AB+1], x , y-1, z-1 ),
                   grad(p[BB+1], x-1, y-1, z-1 ))));
}

double PerlinNoise::octaveNoise( double x, int octaves ) const {
    double result = 0.0;
    double amp = 1.0;

    for(int i=0; i<octaves; ++i) {
        result += noise(x) * amp;
        x *= 2.0;
        amp *= 0.5;
    }
    return result;
}

double PerlinNoise::octaveNoise( double x, double y, int octaves ) const {
    double result = 0.0;
    double amp = 1.0;

    for(int i=0; i<octaves; ++i) {
        result += noise(x,y) * amp;
        x *= 2.0;
        y *= 2.0;
        amp *= 0.5;
    }

    return result;
}

double PerlinNoise::octaveNoise( double x, double y, double z, int octaves ) const {
    double result = 0.0;
    double amp = 1.0;

    for(int i=0; i<octaves; ++i) {
        result += noise(x,y,z) * amp;
        x *= 2.0;
        y *= 2.0;
        z *= 2.0;
        amp *= 0.5;
    }
    return result;
}

double PerlinNoise::grad( int hash, double x, double y, double z ) const {
    const int h = hash & 15;
    const double u = h<8 ? x : y, v = h<4 ? y : h==12||h==14 ? x : z;
    return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
}
```

Appendix F

Experiments

In this part of the thesis, some of the more ponderous experiments the authors conducted are documented. For that, a personal computer with an AMD Phenom(tm) II X4 955 processor, 3.9 GiB memory and a 64-bit Linux Mint 14 for the operating system was used. All C++11[1] code was compiled with *GCC version 4.7.2*.

F.1 Optimal Distance Calculation

First, the authors wrote the distance function in four different ways (see listing F.1). Then, all compiler optimizations were disabled and code shown in listing F.2 was ran.

Listing F.1: Different Distance Function Implementations

```
inline uint32_t pixel_distance_1 (pixel p1, pixel p2 ) {
    return (int32_t(p1.x)-p2.x)*(int32_t(p1.x)-p2.x)+(int32_t(p1.y)-p2.y)*(int32_t(p1.y)-p2.y);
}

inline uint32_t pixel_distance_2 (pixel p1, pixel p2 ) {
    uint16_t x1,x2,y1,y2;
    x1 = p1.x; x2=p2.x;
    y1 = p1.y; y2=p2.y;

    return (int32_t(x1)-x2)*(int32_t(x1)-x2)+(int32_t(y1)-y2)*(int32_t(y1)-y2);
}

inline uint32_t pixel_distance_3 (pixel p1, pixel p2 ) {
    uint16_t x1,x2,y1,y2;
    int32_t xd,yd;
    x1 = p1.x; x2=p2.x; xd = x1-x2;
    y1 = p1.y; y2=p2.y; yd = y1-y2;
    return xd*xd+yd*yd;
}

inline uint32_t pixel_distance_4 (pixel p1, pixel p2 ) {
    int32_t xd = p1.x-p2.x;
    int32_t yd = p1.y-p2.y;

    return xd*xd+yd*yd;
}
```

[1] C++11 (formerly known as C++0x) is the latest ISO C++ standard, ratified in 2011 to replace C++03 [72].

Listing F.2: Speed Measurements

```
am::pixel p1 = am::create_pixel(0,0,0,0,0,0);
am::pixel p2 = am::create_pixel(UINT16_MAX,UINT16_MAX,0,0,0,0);
std::chrono::steady_clock::time_point measure_start, measure_end;
size_t dur;

measure_start = std::chrono::steady_clock::now();
for (size_t i=0; i<1000000000; ++i) am::pixel_distance_1(p1, p2);
measure_end   = std::chrono::steady_clock::now();
dur = std::chrono::duration_cast<std::chrono::nanoseconds>(measure_end - measure_start).count();
printf("am::pixel_distance_1 took %lu ns.\n", dur);

measure_start = std::chrono::steady_clock::now();
for (size_t i=0; i<1000000000; ++i) am::pixel_distance_2(p1, p2);
measure_end   = std::chrono::steady_clock::now();
dur = std::chrono::duration_cast<std::chrono::nanoseconds>(measure_end - measure_start).count();
printf("am::pixel_distance_2 took %lu ns.\n", dur);

measure_start = std::chrono::steady_clock::now();
for (size_t i=0; i<1000000000; ++i) am::pixel_distance_3(p1, p2);
measure_end   = std::chrono::steady_clock::now();
dur = std::chrono::duration_cast<std::chrono::nanoseconds>(measure_end - measure_start).count();
printf("am::pixel_distance_3 took %lu ns.\n", dur);

measure_start = std::chrono::steady_clock::now();
for (size_t i=0; i<1000000000; ++i) am::pixel_distance_4(p1, p2);
measure_end   = std::chrono::steady_clock::now();
dur = std::chrono::duration_cast<std::chrono::nanoseconds>(measure_end - measure_start).count();
printf("am::pixel_distance_4 took %lu ns.\n", dur);
```

Running the above code wrote text shown in listing F.3 into the terminal window. The authors repeated the experiment several times and found out that the fourth implementation of the distance formula remained the fastest. Hence, it was decided that the AtoMorph library should use the fourth variant of the proposed set of possible implementations.

Listing F.3: Speed Measurement Results

```
am::pixel_distance_1 took 10904739000 ns.
am::pixel_distance_2 took 12457372000 ns.
am::pixel_distance_3 took 15360695000 ns.
am::pixel_distance_4 took 10845143000 ns.
```

However, it remains questionable whether the authors' choice of disabling the compiler's optimizations was a good idea. When optimizations are enabled, all dead code is usually left out. Thus, it would be impossible to measure the time cost of the given functions with just a couple of lines of code.

Theoretically, a more correct experiment would keep the compiler optimizations enabled and measure the distances of randomly generated points, summing these and printing the outcome to the standard output. That way, the compiler could not optimize out the dead code, because there would not be any. Yet, the tests would be the closest to a real life scenario, where optimizations are enabled.

F.2 Bug in std::modf

During the development of the AtoMorph library, the authors stumbled into a strange bug in the `std::modf` function. It took several good hours to track down the bug to the named function that occasionally returns an invalid value. The function is ought to return the fractional part of a given number but fails to do so in some specific cases. To isolate the bug and verify its repetitiveness, the authors implemented a program that has its source code given in listing F.4. The program was compiled using the GNU toolchain, having `-std=c++11 -Wall -O2` for the compiler flags.

Listing F.4: Proof of Bug

```
double d_integ;
float  f_integ;
double d_val = 0.8;
float  f_val = 0.8;

printf("std::modf(%f/0.2, &d_integ) = %f\n", d_val, std::modf(double(d_val/0.2), &d_integ));
printf("std::modf(%f/0.2, &f_integ) = %f\n", f_val, std::modf(float(f_val/0.2), &f_integ));

d_val = 0.6;
f_val = 0.6;

printf("std::modf(%f/0.2, &d_integ) = %f\n", d_val, std::modf(double(d_val/0.2), &d_integ));
printf("std::modf(%f/0.2, &f_integ) = %f\n", f_val, std::modf(float(f_val/0.2), &f_integ));

d_val = 3.0;
f_val = 3.0;

printf("std::modf(%f, &d_integ) = %f\n", d_val, std::modf(double(d_val), &d_integ));
printf("std::modf(%f, &f_integ) = %f\n", f_val, std::modf(float(f_val), &f_integ));
```

When ran, the above program prints text shown in listing F.5 to the terminal window. The reader can see that the `std::modf` function sometimes invalidly returns *1.0* when it actually should have returned *0.0*. A brief browsing in the Internet using the *Google Search Engine* gave no significant results for the named bug, leaving a small chance for it to still remain unknown for the wider public.

Listing F.5: Program Output

```
std::modf(0.800000/0.2, &d_integ) = 0.000000
std::modf(0.800000/0.2, &f_integ) = 0.000000
std::modf(0.600000/0.2, &d_integ) = 1.000000
std::modf(0.600000/0.2, &f_integ) = 0.000000
std::modf(3.000000, &d_integ) = 0.000000
std::modf(3.000000, &f_integ) = 0.000000
```

Sometimes such absurd bugs are caused by malfunctioning hardware. To determine whether this is the case or not, the authors conducted the same experiment on their *Dell Vostro 1710* laptop running a 64-bit Linux Mint 15 for the operating system. Turns out that even on different hardware the bug is persistent.

References

[1] Starizona: Adventures In Astronomy & Nature. http://starizona.com/acb/ccd/procadvcomb.aspx. Accessed: 2014-03-11. (p. 18)

[2] Pinaki Pratim Acharjya, Ritaban Das, and Dibyendu Ghoshal. A study on image edge detection using the gradients. Available as: http://www.ijsrp.org/research-paper-1212/ijsrp-p1243.pdf. (p. 2)

[3] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. IEEE Trans. Pattern Anal. Mach. Intell., 33(5):898–916, May 2011. (p. 29)

[4] James Arvo. Graphics gems II, volume 2. Morgan Kaufmann, 1991. (p. 16, 17)

[5] SG Bardenhagen and EM Kober. The generalized interpolation material point method. Computer Modeling in Engineering and Sciences, 5(6):477–496, 2004. (p. 19)

[6] William Baxter, Pascal Barla, and Ken Anjyo. N-way morphing for 2d animation. Computer Animation and Virtual Worlds, 20(2-3):79–87, 2009. Available as: http://hal.archives-ouvertes.fr/docs/00/40/08/30/PDF/main.pdf. (p. 1, 13, 14, 68)

[7] Isameddine Boukhriss, Serge Miguet, and Laure Tougne. Discrete average of two-dimensional shapes. In Andr Gagalowicz and Wilfried Philips, editors, Computer Analysis of Images and Patterns, volume 3691 of Lecture Notes in Computer Science, pages 145–152. Springer Berlin Heidelberg, 2005. (p. 1, 12)

[8] Isameddine Boukhriss, Serge Miguet, and Laure Tougne. Two-dimensional discrete morphing. In Reinhard Klette and Jovia uni, editors, Combinatorial Image Analysis, volume 3322 of Lecture Notes in Computer Science, pages 409–420. Springer Berlin Heidelberg, 2005. Also available as: http://dx.doi.org/10.1007/978-3-540-30503-3_29. (p. 12, 66, 68)

[9] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 23(11):1222–1239, 2001. (p. 4, 9)

[10] Rodney A Brooks. Intelligence without representation. Artificial intelligence, 47(1):139–159, 1991. (p. 15)

[11] Wilhelm Burger and Mark J. Burge. Principles of Digital Image Processing: Fundamental Techniques. Springer Publishing Company, Incorporated, 1 edition, 2009. (p. 18)

[12] Robert P Burns and Richard Burns. Business research methods and statistics using SPSS. Sage, 2008. (p. 42)

[13] Sema Candemir and Yusuf Sinan Akgül. Adaptive regularization parameter for graph cut segmentation. In Image Analysis and Recognition, pages 117–126. Springer, 2010. (p. 6)

[14] Edwin Catmull. The problems of computer-assisted animation. ACM SIGGRAPH Computer Graphics, 12(3):348–353, 1978. (p. 2, 10)

[15] Edwin Catmull and Raphael Rom. A class of local interpolating splines. Computer aided geometric design, 74:317–326, 1974. (p. 16)

[16] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. In Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, pages 219–228. ACM, 2005. (p. 19)

[17] Jose Maria de la Portilla Fernández et al. 3d modelling and animation study of the industrial heritage wonders. In History of Machines for Heritage and Engineering Development, pages 139–159. Springer, 2011. (p. 70)

[18] Erich Erstu. Random World Generator. Bachelor's thesis, University of Tartu: Faculty of Mathematics and Computer Science, 2012.
Available as: http://www.hyena.net.ee/rwg/rwg.pdf. (p. 17)

[19] Erich Erstu. Sohni — Second Visit to the Underworld.
http://www.sohni.net.ee, 2012. Accessed: 2014-03-11. (p. 1)

[20] Erich Erstu. Seminar on Enterprise Software:
Fluid Morphing for 2D Animations. 2013. Available as:
http://hyena.net.ee/ut/pdf/FluidMorphingSeminar.pdf. (p. 1, 58)

[21] Erich Erstu and Siim Kallas. Computer Graphics Project: Fluid Rendering and Image Morphing. 2013. Available as: https://courses.cs.ut.ee/2013/cg/fall/Main/ProjectFluid. (p. 47, 62)

[22] Erich Erstu, Janar Sell, and Suido Valli. Cloud Computing Project: Perlin Noise Generator. http://www.hyena.net.ee/rwg/Perlin%20Noise%20Generator.pdf. Accessed: 2014-05-12. (p. 17)

[23] Pedro F. Felzenszwalb. A stochastic grammar for natural shapes. 2013. Also available as http://arxiv.org/abs/1303.2844. (p. 2)

[24] Adrian Ford and Alan Roberts. Colour space conversions. 1998. (p. 23)

[25] Thomas-Peter Fries and Hermann G Matthies. Classification and overview of meshfree methods. Scientific Computing, Informatikbericht, 3, 2003. (p. 18)

[26] Emerson D. Fuentes, Mark T. Atienza, Mary Grace L. Tuazon, and Ronel V. Fernandez C. Pineda Agao. A hero game: Role playing game based on life and works of rizal of sti college - caloocan, 2013. (p. 70)

[27] Giovanni Paolo Galdi. An Introduction to the Mathematical Theory of the Navier-Stokes Equations: Steady-State Problems, volume 501. Springer, 2011. (p. 19)

[28] Fred Glover. Tabu search-part i. ORSA Journal on computing, 1(3):190–206, 1989. (p. 15)

[29] Stephen Gould. Multiclass pixel labeling with non-local matching constraints. In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, pages 2783–2790. IEEE, 2012. (p. 7, 8)

[30] Michael J. Gourlay. xmorph: library that implements digital image warping, known as morphing. http://xmorph.sourceforge.net/, 1994–2000. (p. 3)

[31] D Gruen, S Seitz, and GM Bernstein. Implementation of robust image artifact removal in swarp through clipped mean stacking. arXiv preprint arXiv:1401.4169, 2014. (p. 18)

[32] Stefan Hinz. Fast and subpixel precise blob detection and attribution. In Image Processing, 2005. ICIP 2005. IEEE International Conference on, volume 3, pages III–457. IEEE, 2005. Also available as http://pdf.aminer.org/000/323/513/fast_and_subpixel_precise_blob_detection_and_attribution.pdf. (p. 2, 4, 5)

[33] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. ACM Transactions on Computer Graphics, 24(3):1134–1141, 2005. http://www-ui.is.s.u-tokyo.ac.jp/~takeo/research/rigid/. (p. 13)

[34] Suraj Jain and Venu Madhav Govindu. Efficient Higher-Order Clustering on the Grassmann Manifold. ICCV, 2013. (p. 8)

[35] Kenneth I Joy. Catmull-rom splines. http://graphics.cs.ucdavis.edu/education/CAGDNotes/Catmull-Rom-Spline/Catmull-Rom-Spline.html. Accessed: 2014-05-26. (p. 16)

[36] Doyub Kim, Oh-young Song, and Hyeong-Seok Ko. A semi-lagrangian cip fluid solver without dimensional splitting. In Computer Graphics Forum, volume 27, pages 467–475. Wiley Online Library, 2008. (p. 19)

[37] Tae Hoon Kim, Kyoung Mu Lee, and Sang Uk Lee. Nonparametric higher-order learning for interactive segmentation. In Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on, pages 3201–3208. IEEE, 2010. (p. 9)

[38] Pushmeet Kohli, M Pawan Kumar, and Philip HS Torr. P^3 & beyond: Move making algorithms for solving higher order functions. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 31(9):1645–1656, 2009. (p. 7)

[39] Pushmeet Kohli, Philip HS Torr, et al. Robust higher order potentials for enforcing label consistency. International Journal of Computer Vision, 82(3):302–324, 2009. (p. 9)

[40] Vladimir Kolmogorov. Convergent tree-reweighted message passing for energy minimization. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 28(10):1568–1583, 2006. (p. 4)

[41] Vladimir Kolmogorov and Ramin Zabin. What energy functions can be minimized via graph cuts? Pattern Analysis and Machine Intelligence, IEEE Transactions on, 26(2):147–159, 2004. (p. 4)

[42] Alexander Kort. Computer aided inbetweening. In Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering, pages 125–132. ACM, 2002. (p. 2)

[43] Grant Kot. Fluid Simulator. http://grantkot.com/MPM/Liquid.html. Accessed: 2014-03-12. (p. 19)

[44] Yuriy Kotsarenko and Fernando Ramos. An alternative color space for color selection and image manipulation with improved brightness component. (p. 23)

[45] Friedrich Reinhold Kreutzwald, Jüri Kurman, and Ann Liivak Ojamaa. Kalevipoeg: An ancient estonian tale. Symposia Press, 1982. (p. 1)

[46] Jenny Lam. Implementing regenerative morphing. 2010. Also available as: http://www.ics.uci.edu/~jlam2/regmorph.pdf. (p. 14)

[47] Jens Christian Morten Laursen. Improving the realism of real-time simulation of fluids in computer games. 2013. (p. 19)

[48] Seung-Yong Lee, Kyung-Yong Chwa, James K. Hahn, and Sung Yong Shin. Image morphing using deformation techniques. Journal of Visualization and Computer Animation, 7(1):3–23, 1996.
Available as: http://www.icg.gwu.edu/Publications/Lee96.pdf. (p. 1, 10, 11)

[49] Jing Li, Gang Zeng, Rui Gan, Hongbin Zha, and Long Wang. Higher-order clique based image segmentation using evolutionary game theory. Artificial Intelligence Research, 3(2):p1, 2014. (p. 9)

[50] Nadia Magnenat Thalmann and Daniel Thalmann. Computer animation in future technologies. In Interactive computer animation, pages 1–9. Prentice-Hall, Inc., 1996. (p. 11)

[51] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In Proc. 8th Int'l Conf. Computer Vision, volume 2, pages 416–423, July 2001. (p. 9)

[52] Talya Meltzer, Chen Yanover, and Yair Weiss. Globally optimal solutions for energy minimization in stereo vision using reweighted belief propagation. In Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on, volume 1, pages 428–435. IEEE, 2005. (p. 4)

[53] Marvin Minsky. Steps toward artificial intelligence. Proceedings of the IRE, 49(1):8–30, 1961. (p. 15)

[54] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In Proceedings of the 2003 ACM

SIGGRAPH/Eurographics symposium on Computer animation, pages 154–159. Eurographics Association, 2003. (p. 19)

[55] Heesoo Myeong and Kyoung Mu Lee. Tensor-based high-order semantic relation transfer for semantic scene segmentation. In Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on, pages 3073–3080. IEEE, 2013. (p. 8)

[56] Michael Newberry. Combining Images Using Integer and Real Type Pixels. http://www.mirametrics.com/tech_note_intreal_combining.htm. Accessed: 2014-03-11. (p. 18)

[57] Nikhil R Pal and Sankar K Pal. A review on image segmentation techniques. Pattern recognition, 26(9):1277–1294, 1993. (p. 4)

[58] Ken Perlin. An image synthesizer. ACM Siggraph Computer Graphics, 19(3):287–296, 1985. (p. 17)

[59] Dilip Kumar Prasad. Object Detection in Real Images. Phd confirmation report, Nanyang Technological University, School of Computer Engineering, August 2010. Also available as http://arxiv.org/abs/1302.5189. (p. 2)

[60] C.T. Reviews. e-Study Guide for: Digital Information Management by Stephen J. Ethier, ISBN 9780131997738. Cram101, 2012. (p. 70)

[61] C.T. Reviews. e-Study Guide for: Film Production Technique : Creating the Accomplished Image by Bruce Mamer, ISBN 9780534629168. Cram101, 2012. (p. 2)

[62] Azriel Rosenfeld and John L Pfaltz. Distance functions on digital pictures. Pattern recognition, 1(1):33–61, 1968. (p. 12)

[63] Joel Le Roux, Philippe Chaurand, and Mickael Urrutia. Matching edges in images; application to face recognition. 2006.
Available as http://arxiv.org/abs/cs/0603086. (p. 2)

[64] J. Russell and R. Cohn. Artificial Imagination. Book on Demand, 2012. (p. 69)

[65] Bart Selman and Carla P Gomes. Hill-climbing search. Encyclopedia of Cognitive Science. (p. 15)

[66] Eli Shechtman, Alex Rav-Acha, Michal Irani, and Steve Seitz. Regenerative morphing. In IEEE Conference on Computer VIsion and Pattern Recognition (CVPR), San-Francisco, CA, June 2010. Also available as: http://grail.cs.washington.edu/projects/regenmorph/. (p. 1, 14, 64, 66, 70)

[67] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 22(8):888–905, 2000. (p. 4)

[68] Denis Simakov, Yaron Caspi, Eli Shechtman, and Michal Irani. Summarizing visual data using bidirectional similarity. In Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on, pages 1–8. IEEE, 2008. (p. 14)

[69] Jos Stam. Stable fluids. In Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999. (p. 19)

[70] Michael Steffen, Robert M Kirby, and Martin Berzins. Analysis and reduction of quadrature errors in the material point method (mpm). International journal for numerical methods in engineering, 76(6):922–948, 2008. (p. 19)

[71] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. ACM Transactions on Graphics (TOG), 32(4):102, 2013. (p. 19, 58)

[72] Bjarne Stroustrup. C++11 - the new ISO C++ standard. http://www.stroustrup.com/C++11FAQ.html. Accessed: 2014-05-12. (p. 78)

[73] Roger Temam. Navier–Stokes Equations. American Mathematical Soc., 1984. (p. 19)

[74] Christopher Twigg. Catmull-rom splines. Computer, 41(6):4–6, 2003. (p. 16)

[75] Sara Vicente, Vladimir Kolmogorov, and Carsten Rother. Graph cut based image segmentation with connectivity priors. In Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on, pages 1–8. IEEE, 2008. (p. 6)

[76] Martin J Wainwright, Tommi Jaakkola, and Alan S Willsky. Tree-based reparameterization for approximate inference on loopy graphs. In NIPS, pages 1001–1008, 2001. (p. 4)

[77] Alyce Watson. Sprite Byte: All About Masks. http://babek.info/libertybasicfiles/lbnews/nl143/sprite.htm. Accessed: 2014-03-11. (p. 2)

[78] Max Wertheimer. Laws of organization in perceptual forms. (p. 4)

[79] Richard Williams. The Animator's Survival Kit. Faber & Faber, 2001. (p. 2, 10)

[80] George Wolberg. Image morphing: a survey. The visual computer, 14(8):360–372, 1998. Also available as http://css1a0.engr.ccny.cuny.edu/~wolberg/pub/vc98.pdf. (p. 1)

[81] Enhua Wu and Feitong Liu. Robust image metamorphosis immune from ghost and blur. The Visual Computer, 29(4):311–321, 2013. (p. 11)

[82] Xueqiao Xu. Fluid Simulation. http://code.google.com/p/mycodeplayground/. Accessed: 2014-03-12. (p. 19)

[83] Jonathan S Yedidia, William T Freeman, Yair Weiss, et al. Generalized belief propagation. In NIPS, volume 13, pages 689–695, 2000. (p. 4)

[84] Cem Yuksel, Scott Schaefer, and John Keyser. Parameterization and applications of catmull–rom curves. Computer-Aided Design, 43(7):747–755, 2011. (p. 16)

[85] Song Chun Zhu and David Mumford. A Stochastic Grammar of Images. Now Publishers Inc, 2007. Also available as: http://books.google.ee/books?id=2QULLmiErNgC&source=gbs_navlinks_s. (p. 2)

Non-exclusive licence to reproduce thesis and make thesis public

I, **Erich Erstu** (date of birth: **15. 02. 1989**),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Fluid Morphing for 2D Animations,

supervised by **Benson Muite**,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **26. 05. 2014**