UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME GOMES HAETINGER

Fast Iterative Inversion for Non-linear Vector Fields Applied to Images, Videos and Volumes

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Eduardo S. L. Gastal

Porto Alegre
July 2021

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitora de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Rodrigo Machado
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

CONTENTS

ACKNOWLEDGMENTS

"No man is an island entire of itself; every man is a piece of the continent, a part of the main;..."

– John Donne, *No Man Is an Island*

I would like to thank CNPq for financing the publication of the shortened version of this work. This work was sponsored by CNPq-Brazil (436932/2018-0), and financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

I have had a number of professors along the way, all of which I am happy to have met and worked with/learned from. A special thanks to my advisor and friend Prof. Eduardo Gastal for being an amazing teacher who is always prompt to help, advise and give feedback. All of this work, the previous publication and awarding was a team effort.

There are many outside of faculty involved with me graduating and being able to present such meaningful work as my thesis. Thus, I need to thank my friends from school and college that always pushed me to the next level. In particular, my appreciation goes to Andy, Arthur, Benjamin, Lucas and Vicente. Furthermore, I want to thank the unquestionable love and constant support from my family: Ricardo, Suzana, Henrique, Esther, Edite, Vera and Manuela. This wouldn't have been possible without you all.

ABSTRACT

The demand for image manipulation applications and software has become a great incentive to the research and implementation of new methods that generate amusing/useful visual results. Filters, face morphing and video special effects have gathered immense attention in present times and are good examples of multimedia manipulation applications. However, processing visual data in a fast manner is a challenging problem due to ever-increasing resolution. This is something that has been worked on for a long time and solutions in high performance hardware and efficient algorithms have been elaborated. Our proposal aims to use both.

We present a novel approach for image, video and volume deformation. Our technique involves the inversion of the nonlinear regularized Kelvinlet equations, leading to higher-quality results and time/space efficiency compared to naive solutions. Inversion for video and image transformations is performed by a per-pixel optimization process, being inherently parallel and achieving real-time performance in Full HD resolution (over 300 fps). For volumes, we analyze whether the best approach is to compute in a per-pixel manner (*i.e.* in *screen space*) or to process the data before it is rendered, using **CUDA** kernels, in a per-voxel manner (*i.e.* in *3D space*). We demonstrate our method on a variety of multimedia examples, in addition to discussing important technical and theoretical details along with practical usages.

Keywords: Numerical Calculus. Vector Fields. CUDA. Deformation. Images. Interpolation. OpenGL. Image Processing. Matricial Transformation. Video Retiming.

Inversão Rápida e Iterativa para a Aplicação de Campos Vetoriais Não-lineares em Imagens. Vídeos e Volumes

RESUMO

A demanda por aplicações e software de manipulação de imagens se tornou um grande incentivo à pesquisa e implementação de novos métodos que geram resultados visuais interessantes/úteis. Filtros, transofrmações faciais e efeitos especiais de vídeos atraíram uma atenção imensa nos tempos de hoje e são ótimos exemplos de aplicações de manipulação de multimídia. No entanto, processar dados visuais de maneira rápida é um grande desafio devido o constante aumento de resolução. Isso é algo no qual se trabalha há muito tempo e soluções em hardware de alto processamento e algoritmos eficientes vêm sido elaboradas. Nossa proposta tem como objetivo usar ambos.

Nós apresentamos uma nova abordagem para deformação de imagens, vídeos e volumes. Nossa técnica envolve a inversão das equações não-lineares *regularized Kelvinlet*, retornando resultados de maior qualidade e eficiência de tempo/memória que soluções simples e *forward mapped*. Inversão de transformações em vídeos e imagens é executado em um processo de otimização *per-pixel*, sendo inerentemente paralelo atingindo a marca de 300 fps em resolução Full HD. Para volumes, nós analizamos se a melhor abordagem seria computar em uma maneira *per-pixel* (*i.e. screen space*) ou processar os dados antes da sua renderização, usando ***CUDA*** *kernels* em uma maneira *per-voxel* (*i.e. 3D space*). Nós demonstramos nosso método em uma variedade de exemplos de multimídia, além de discutir detalhes técnicos e teóricos importantes junto a exemplos de uso.

Palavras-chave: Cálculo Numérico. Campos Vetoriais. CUDA. Deformação. Imagens. Interpolação. OpenGL. Processamento de Imagens. Processamento de Vídeos. Processamento de Volumes. Retemporização de Vídeos. Transormação Matricial. Visualização de Volumes. .

1 INTRODUCTION

The production of visually appealing images and videos is a critical factor for their consumption. In particular, the animation and movie industry have shown to be a much larger market than previously imagined, mostly because of amazing special effects that were not thought possible a few years ago. The development of new and improved image and video processing algorithms plays an important role in this context.

Our work targets the fields of image deformation, video time warping and volume deformation, with the goal of creating visually appealing artificial results. Image deformation techniques create new images by "deforming the pixels" of existing images (AVIDAN; SHAMIR, 2007; KARNI; FREEDMAN; GOTSMAN, 2009; KAUFMANN et al., 2013). We propose a new real-time algorithm that creates smooth pictures with sub-pixel accurate deformations. It is based on the elastic patterns designed into the Kelvinlet sculpting brush (GOES; JAMES, 2017). Our solution is trivially parallelizable using the graphics hardware, and can take advantage of the built-in mipmapping capabilities for antialiasing (required due to the resampling that occurs during deformation (WOLBERG, 1990)), generating smooth images and high-quality results like in Fig. 1.1. Video time warping is a less-explored area, which can be seen as an extension of image deformation, from 2D space to 3D space-time. We use the concept of the space-time video volume and demonstrate the use of our technique for smooth, real-time temporal deformations (time warping).

Volume visualization techniques have been extensively studied for a long time (COTIN; DELINGETTE; AYACHE, 1999; PFISTER et al., 1999). These in-

Figure 1.1 – Teaser deformation: Turning Big Buck Bunny's frown into a smile!

(a) Reference image (b) #1: Mouth (left) (c) #2: Eyebrows (d) #3: Mouth (right)

Video Source: Blender Foundation (https://www.blender.org/)

volve scientific fields such as medicine and archeology. *Volume deformation* has had less of an impact on the field of Computer Graphics, but still has meaningful research surrounding it (CHEN; HESSER; MäNNER, 2011; GASCON et al., 2013). We face the difficulties involving high-dimensional data by approaching this problem with efficient implementations in **CUDA** and **OpenGL**.

After citing related works and giving a summarized background on the Kelvinlets methd, we describe our work and implementation. The contributions of our work include:

- An efficient image and video warping algorithm that is inherently parallel and easy to implement. It is able to process Full HD videos at over 300 frames per second through our nonlinear per-pixel optimization process (Chapter 4);
- An inversion method for the nonlinear Kelvinlet equations, applied to image deformation and video time warping through backward mapping. This generates higher-quality results when compared to forward mapping (Section 4.2);
- A procedure for altering the Kelvinlet deformation field in order to preserve the image or video's rectangular boundary, while still generating smooth deformations (Section 4.3);
- A discussion of important technical details for generating high-quality antialiased images, as well as for detecting and handling non-invertible deformations. We also discuss the relationship between temporal and spatial aliasing (Chapter 5).
- The application of our inversion algorithm to Volumes, and implementations that aim for better and faster user interaction (Section 5.4).

2 RELATED WORKS

In this section, we review the most relevant related works on image deformation, time warping and volume deformation.

2.1 Image Deformation

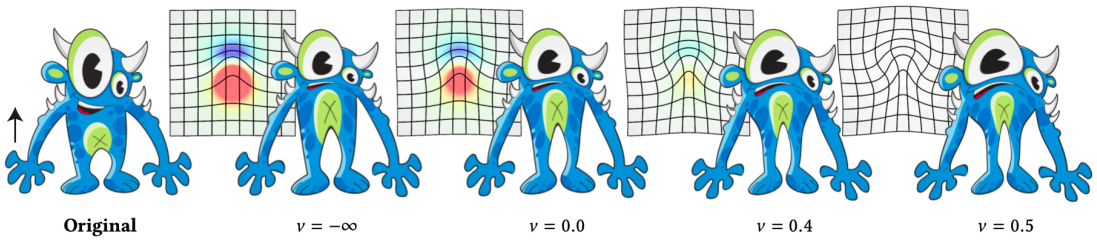
Image deformation has numerous applications, ranging from photo editing to content-aware image resizing (AVIDAN; SHAMIR, 2007). Kauffman *et al.* (KAUFMANN et al., 2013) consider a content-aware method for image warping. They use an image meshing algorithm that respects the geometry of its objects. The same method is also extendable to adaptive meshing for video sequences, enabling the deformation to persist throughout the frames. However, their algorithm depends on object detection in the images. The energy-based deformation proposed by Karni *et al.* (KARNI; FREEDMAN; GOTSMAN, 2009) uses energy minimization to avoid distorted results. This method uses multiple weight points to generate a structured deformation, much like the "warp tools" from modern photo editing software (FOUNDATION, 2020).

The regularized Kelvinlet brush (GOES; JAMES, 2017) describes a volume-preserving force field that may be used for deformations. In their paper, De Goes and James are mostly focused on "sculpting" 3D polygonal meshes, where the displacements can be directly applied to the vertices through a *forward-mapping* implementation. This approach, however, is not ideal for processing images and videos, for the reasons presented in Section 4.1. They still apply it to images, as shown in Fig. 2.1. Our work extends the Kelvinlet equations to a *backward-mapping* approach (WOLBERG, 1990). We discuss the Kelvinlet brush in more detail in Chapter 3.

2.2 Video Time Warping

The relationship between videos and 3D volumes is not new to the Computer Graphics community (FREY, 2018; KLEIN et al., 2002; RAV-ACHA et al., 2005; SOLTESZOVA et al., 2020). As pointed out by Rav-Acha *et al.* (RAV-ACHA

Figure 2.1 – Elastic deformation brush from (GOES; JAMES, 2017) applied with different poisson ratios.



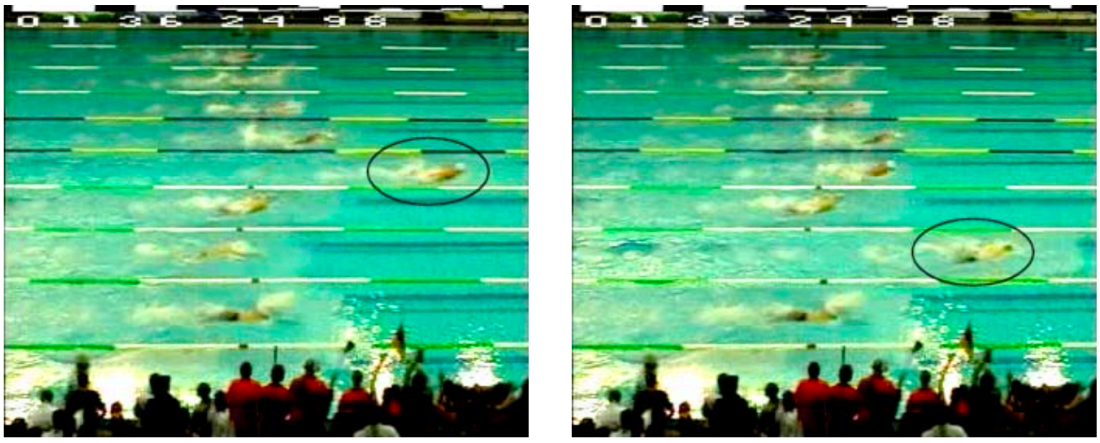
Source: Mirela Ben-Chen (GOES; JAMES, 2017)

et al., 2005), the space-time volume used by our algorithm (to depict and modify time) was introduced as the *epipolar volume* by Bolles in 1987 (BOLLES; BAKER; MARIMONT, 1987), and it has since been adopted by many researchers for video processing and analysis (FREY, 2018; BACH et al., 2016).

Many works make use of time warping for achieving specific artistic or technical results, without explicitly focusing on how the actual warping itself is performed. For example, Rav-Acha et al. (2005) elaborate a time warping framework based on *evolving time fronts*, which are smooth space-time surfaces that traverse the video volume, creating new frames. They achieve interesting results with this idea, such as altering the winner in a swimming competition video by delaying (or advancing) each competitor in time (Fig. 2.2). While they mostly focus on warping along the time dimension, they also mention the possibility of warping along time while also deforming in space (which our technique supports natively). Their work does not discuss the underlying algorithm used to warp each video frame.

Other works with similar ideas include that of Klein et al. (2002), which presents a non-photorealistic rendering tool for videos. It explores the notion of time and space questioned by paintings of the Cubist and Futurist art movements. More recently, Solteszova et al. (2020) target time warping for scientific visualizations. Operating on top of video streams instead of static files, *Memento* is their tool for real-time video interaction and exploration. It is applicable in diverse scenarios with a good user interface and useful results for monitoring and comparing video segments through time. Different from our approach, they only demonstrate warpings along time like in Fig. 2.3, and do not explore spatial deformations combined with time warping.

Figure 2.2 – Video deformation applied with the time warping framework of (RAV-ACHA et al., 2005). *Who is the winner of this swimming competition?*



Source: (*RAV-ACHA et al., 2005*)

Figure 2.3 – Videos with outlined time shifts during movement.

(a) Rewinding and slowing down.

(b) Delaying to synchronize a canon choreography in ballet.

Source: (*SOLTESZOVA et al., 2020*)

The aforementioned methods differ from ours in that they do not propose new algorithms for the actual warping procedure, and instead focus on applications of warping. We discuss this topic further in Section 4.1.

Relation to 3-D volume rendering: one of the difficulties involved in video time warping is the generation and "playback" of the resulting warped video frames. While seemingly related to Volume Rendering techniques (WYLIE et al., 2002; FREY, 2018; SILVA et al., 2005), it is important to note that these are not directly applicable to rendering time-warped videos. This occurs because volume rendering methods treat the volume as a semi-transparent material that can emit, transmit, and absorb light, *i.e.* is intended to be rendered all at once, whereas the space-time video volume is meant to be rendered "slice by slice", *i.e.* , frame by frame.

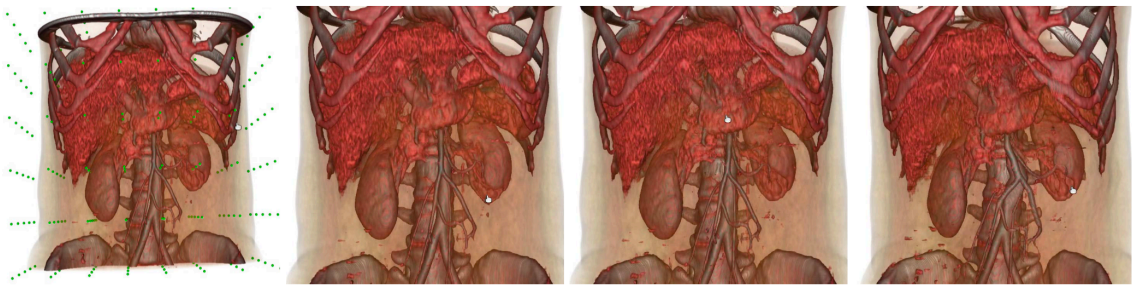
Other related topics include the generation of regular speed videos from high-frame-rate inputs. Zhou *et al.* (ZHOU; KANG; COHEN, 2014) process videos in order to detect movement. This way, they are able to lower the frame-rate by using temporally-variant filters to enhance the display of salient motions, which can be seen as a 1D global warping function applied uniformly to all pixels. Their work is orthogonal to ours and can be used together with our warping technique.

2.3 Volume Deformation

Volume deformation is used when trying to simulate physical events on dense three-dimensional bodies. However, this a difficult thing to do and even harder to make interactive. Volumes have three dimensions, so if a program has to deform every single voxel, it is of complexity $\mathcal{O}(n^3)$, where n is the number of steps along each dimension. Researches look for ways to mitigate this processing bottleneck by making volume deformation as fast as possible.

Cotin, Delingette e Ayache (1999) approach volume deformation in a medical perspective. Aiming to simulate tissue deformation during surgery, they use both a quasi non-linear and a linear physical model that, as our Kelvinlet model, is based on elasticity theory. They also implement a physical feedback device. However, instead of applying the deformation on the actual volume, they use preprocessed volume data to reconstruct a mesh in order to give faster feedback to the user.

Figure 2.4 – Volume deformation applied in a human body organ structure.



Source: (*GASCON et al., 2013*)

Chen, Hesser e Männer (2011) have a similar approach to ours. They use the inverse of a non-linear vector field to retrieve the ray marching points of the deformed volume. A big part of this work revolves around optimizing the number of casted rays into the volume to avoid unecessary computational cost. They use the *FFD* (Free-form-deform) model, which already has work around the field's consistent inversion (MODAT et al., 2012). Thus, the solution is specific to a given vector field and doesn't generalize the method for other deformation models. They also do not go into how they handled user interaction and how fast their program's runtime was in terms of framerate.

Gascon et al. (2013) works towards an interactive application to deform volumes in a fast manner. Their method, different from ours, takes in a tetrahedral mesh as a representation of the volume data. This makes it so that they take the advantage of being able to use forward mapping deformation and achieve high efficiency with a number of culling optimizations while still getting great results such as the one in Fig. 2.4. However, their algorithm is not very extensible, suffering from problems when rasterizing tetrahedrons, *e.g.* unaligned points due to volume rotation.

Figure 3.1 – Demonstration of the influence of the Poisson ratio (ν) and the radius (ε) on the deformation result. (a) Reference image and deformation parameters: the pivot p_0 is dragged by a user to a new position p_{end}, defining a deformation force $\vec{f}$. (b) Deformations with $\nu \to -\infty$ do not preserve the *local* volumes, as shown by the severe compression indicated by the green arrow. (c) Deformations with $\nu \approx 0.5$ preserve local volumes more consistently across the domain. For this to be possible, the image must "bulge outwards." (Section 4.3 discusses a method to preserve the rectangular image boundary, which was not applied here for illustrative purposes.) (d) Deformations with a larger influence radius ε generate less-concentrated results.

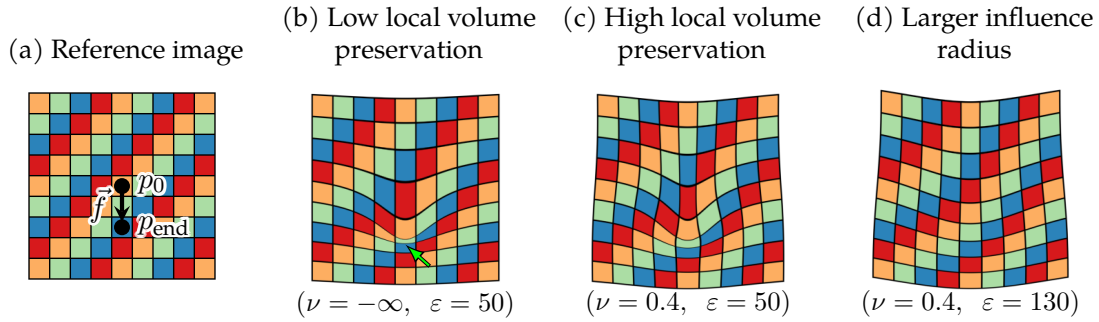


Image Source: The Author

3 BACKGROUND ON THE REGULARIZED KELVINLET

The Regularized Kelvinlets (GOES; JAMES, 2017) are elastostatic deformation methods based on the fundamental solution of linear elasticity, also known as Kelvin's state. In this work, we focus on the application of the grab brush solution (GOES; JAMES, 2017). This method defines a displacement vector $K(p)$ for each point p in the domain (in our case, each pixel in a 2D image or 3D video volume), such that the resulting vector field has a controllable degree of volume preservation.

The Kelvinlet equations can be better understood from a simple usage example. Imagine a user that clicks on a point at position p_0 in an image (called the deformation pivot), and drags it to a new position p_{end} (Fig. 3.1a). This defines a displacement force at p_0 to be the vector $\vec{f} := p_{\text{end}} - p_0$. The force vectors for all other points in the domain are then given by (GOES; JAMES, 2017):

$$K(p) = c\,\varepsilon\,U(p - p_0)\,\vec{f}, \tag{3.1}$$

where

$$U(\vec{r}) = \frac{(a-b)}{r_\varepsilon} I + \frac{b}{r_\varepsilon^3} \vec{r}\,\vec{r}^{\,\top} + \frac{a}{2}\frac{\varepsilon^2}{r_\varepsilon^3} I, \tag{3.2}$$
$$\text{with } \; r_\varepsilon = \sqrt{\|\vec{r}\|^2 + \varepsilon^2}.$$

The values $a = \frac{1}{4\pi}$, $b = \frac{a}{4(1-\nu)}$, and $c = \frac{2}{3a-2b}$ are constants that depend on the Poisson ratio ν, which controls the volume-preservation properties of the Kelvinlet field (GOES; JAMES, 2017). The parameter ε controls the "radius" of the deformation effect, *i.e.* , how much the displacement at p_0 affects its neighborhood. The effects of ν and ε are illustrated in Fig. 3.1. For all results shown in this work, we use $\nu = 0.4$, which generates good results in practice. The values for ε, the pivot p_0 and the force $\vec{f}$ are provided in each figure's caption. Finally, we note that in Eq. (3.2), I denotes the identity matrix, $\|\cdot\|$ denotes the Euclidean norm, and $\cdot^\top$ denotes a transpose. Also observe that this equation is valid for both 2D and 3D deformations.

4 OUR KELVINLET INVERSION METHOD FOR {2, 3}-DIMENSIONAL SPACES

As opposed to the 3D sculpting brush described by De Goes and James (GOES; JAMES, 2017), the deformation method proposed here uses an inverse transform. We now explain the inversion process, starting with a discussion on forward vs backward mapping (Section 4.1). All equations that follow are applicable to both image deformation (2D) and video time warping (3D domain).

4.1 Forward vs Backward-Mapping Implementations

The force field K from Eq. (3.1) translates each pixel p in the input image g to a new position $q = T(p) = p + K(p)$ in the output warped image g_w. A naive **forward-mapping** implementation for generating all output pixels works as follows:

Algorithm 1 Image deformation with forward mapping

1: **For each input pixel** p in the input image g:
2: **Compute the output** pixel location $q = T(p)$;
3: Set the output pixel color $g_w(q) = g(p)$.

This solution has important difficulties (SZELISKI, 2011). For instance, note that most input pixels p are mapped to non-integer output locations q, and also that neighboring input pixels may be mapped to *non*-neighboring output pixels. As such, a post-deformation interpolation is required to fill the gaps generated by the input pixel's displacements (WOLBERG, 1990). One way to do this in 2D is by associating each pixel with a vertex in a triangle mesh, and then performing interpolation between the displaced pixels through triangle rasterization (HECKBERT, 1989) (higher-order interpolations are also possible (WOLBERG, 1990)). This process becomes significantly more complicated in 3D, where each triangle is replaced by a 3D tetrahedron. In this scenario, rasterization becomes a bottleneck, achieving hardly interactive rates even for small volumes (GASCON et al., 2013). Another solution for 3D post-deformation interpolation is described in Appendix A.

A generally better solution is to use an inverse transformation, due to the fact that it does not require the post-deformation interpolation between pixels (WOLBERG, 1990; SZELISKI, 2011). Therefore, our method uses T^{-1} to map lo-

cations from the *output* image g_w to locations on the *input* image g (thus, in the reverse direction). This results in the following **backward-mapping** implementation:

Algorithm 2 Image deformation with backward mapping

1: **For each output pixel** location q in the output image g_w:
2: **Compute the input** pixel location $p = T^{-1}(q)$;
3: Set the output pixel color $g_w(q) = g(p)$.

This solution addresses all of the difficulties discussed above. It does not require any post-deformation interpolation since the for-loop traverses all *output* pixel locations q. Furthermore, the result for each q is independent of the other pixels and thus all output pixels can be computed in parallel. The only interpolation required is in the input domain, when sampling the input image g at location p, to obtain the color $g(p)$. But since the input domain defines a regular 2D or 3D pixel grid, this interpolation is a trivial operation, that can be performed by standard bilinear or higher-order kernels. In a GPU implementation, bilinear interpolation is directly supported by the hardware. One can also take advantage of mipmapping for antialiasing (Section 5.1).

The backward-mapping solution requires the computation of T^{-1}. As such, T must be invertible (a bijection), and one must be able to compute $T^{-1}(q)$ for each q. Unfortunately, in the case of the Kelvinlet field (Eq. (3.1)), there is no closed-form solution for its inverse. In the next section we show how a nonlinear optimization technique can be used to compute T^{-1} efficiently, and Section 5.2 discusses how to handle situations where T is not invertible. Fig. 4.1 illustrates the relationship between T and T^{-1}.

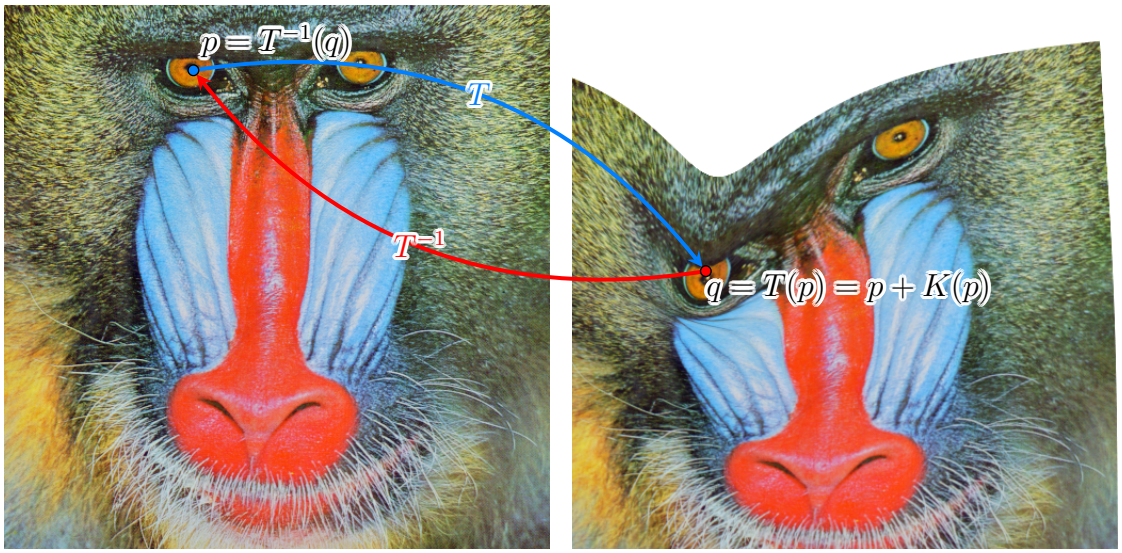
4.2 Inverting the Kelvinlet Deformation using Gauss-Newton

Given a particular pixel location q, we wish to compute $p = T^{-1}(q)$ (line 2 of Algorithm 2). Our insight is that this problem can be modeled as a nonlinear optimization:

$$T^{-1}(q) = \arg\min_{p'} \quad \|T(p') - q\|^2 . \tag{4.1}$$

In other words, the point p' which minimizes the functional on the right-hand-side of Eq. (4.1) is our solution $p = T^{-1}(q)$.

Figure 4.1 – Relationship between T and T^{-1}. The deformation $T(p) = p + K(p)$ maps pixels from the input image to new positions in the output deformed image, using the Kelvinlet field K. Our method computes the output image by "consulting" colors in the input domain using T^{-1}.



Reference image (input) Deformed image (output)

Image Source: Public Domain

We solve Eq. (4.1) using iterative Gauss-Newton (SOLOMON, 2015), which is applicable in this situation since K (Eq. (3.1)), and thus T, is continuously differentiable (the basic idea is to approximate the right-hand-side of Eq. (4.1) with its first-order Taylor expansion (SOLOMON, 2015)). The algorithm works as follows: starting from an initial guess p'_1, a series of improved solutions $p'_2, p'_3, \ldots$ are obtained from the update step $p'_{k+1} = p'_k + \delta_k$, where

$$\delta_k = \arg\min_{\delta} \quad \|p'_k + \delta + K(p'_k) + J(p'_k)\,\delta - q\|^2 . \tag{4.2}$$

In the above equation, $J(p'_k)$ is the Jacobian matrix of the operator K, computed at p'_k. In the case of image deformations, J is a 2×2 matrix, and in the case of video time warping and volume deformation, J is a 3×3 matrix. The closed-form expressions for the Jacobians are in Appendix B and Appendix C.

The right-hand-side of Eq. (4.2) is a quadratic functional on δ, whose minimum is obtained by differentiating and equating to zero. This results in a closed-form linear solution for δ_k:

$$\delta_k = A_k^{-1} \mu\, b_k, \tag{4.3}$$

Figure 4.2 – (Left) Histogram of the number of iterations required for convergence of the Gauss-Newton iterations. Most pixels require around 10 to 20 iterations for the residual norm to fall below the threshold of 10^{-1}. (Right) Per-pixel visualization of the number of iterations required for convergence. Note how pixels that are closer to the deformation pivot require more iterations. Results computed for the deformation in Fig. 4.1.

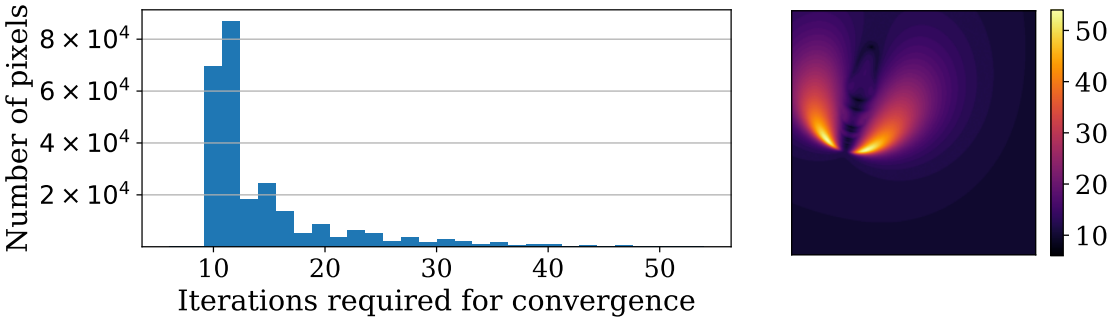


Image Source: The Author

where

$$A_k = J(p'_k)^\top J(p'_k) + 2J(p'_k)^\top + I, \tag{4.4}$$

$$b_k = -\left(J(p'_k)^\top + I\right)\, \xi_k, \tag{4.5}$$

$$\xi_k = p'_k + K(p'_k) - q. \tag{4.6}$$

We set the Gauss-Newton damping parameter $\mu = 0.5$ and stop the iterations when the residual norm falls below the threshold 10^{-1} (in pixel units): $\|\xi_k\| < 10^{-1}$. This permits a maximum error in $T^{-1}(q)$ of one-tenth the size of a pixel, which is visually imperceptible. For each pixel q we start with the initial guess $p'_1 = q$, which is a sensible choice since the majority of the pixels in an image are generally not affected by a particular deformation (*i.e.* , $K(p) \approx \vec{0}$ for most pixels, and thus $p = T^{-1}(q) \approx q$). Fig. 4.2 shows the typical number of iterations required for convergence.

4.3 Fixed Domain Boundary

Different from 3D sculpting of polygonal meshes (GOES; JAMES, 2017), when deforming an image or video one is generally interested in preserving its rectangular boundary positions. Thus, we propose an additional term that smoothly decreases the deformation field's strength as one gets close to the boundary (Fig. 4.3b).

This is done by using a control curve (Eq. (4.7)), which dictates how much of the original force will be used at a given pixel, depending on its distance from the boundary:

$$\beta_\circ(p) = \sin\left(\frac{\pi}{2}\frac{\min(D_\circ(p), \sigma)}{\sigma}\right). \tag{4.7}$$

In the equation above, σ represents the radius along the boundary where the force falloff starts to take place. Note that $\beta_\circ(p) \in [0, 1], \forall p$, smoothly varies from 0 at the boundary to 1 at σ-units from the boundary. Furthermore, $\circ \in \{x, y, t\}$ stands for a particular dimension, and $D_\circ(p)$ computes the distance of pixel p from the boundary along that particular dimension. For example, in an image of size $W \times H$, with one-based pixel indexing:

$$\begin{aligned} D_x(p) &= \min(p_x - 1, W - p_x), \quad \text{and} \\ D_y(p) &= \min(p_y - 1, H - p_y). \end{aligned} \tag{4.8}$$

We use $\sigma = 50$ pixels for the examples shown in this work where a fixed boundary was employed. The modified force field K_β is then given by:

$$K_\beta(p) = \begin{bmatrix} \beta_x(p) & \\ & \beta_y(p) \end{bmatrix} K(p). \tag{4.9}$$

When processing 3D video volumes (time warping), the entry β_t is added in a third row and column to the matrix above.

Since these operations modify the deformation field, the Jacobian matrix J in Eq. (4.2) must also be updated. This is done by applying the differentiation product rule to Eq. (4.9), resulting in a new Jacobian J_β:

$$J_\beta = \begin{bmatrix} \beta_x & \\ & \beta_y \end{bmatrix} J + \begin{bmatrix} \frac{\partial \beta_x}{\partial x} & \\ & \frac{\partial \beta_y}{\partial y} \end{bmatrix} \begin{bmatrix} K_x & \\ & K_y \end{bmatrix}. \tag{4.10}$$

The new matrix $J_\beta(p'_k)$, evaluated at p'_k, must be used in the Gauss-Newton update steps in Eqs. (4.3) to (4.6).

Figure 4.3 – (a) A reference image with fine details. (b) Our antialiased deformation with a fixed boundary results in a high-quality and smoothly-deformed image. (c) Without a fixed boundary, out-of-bounds pixels with undefined colors "leak" into the image (shown in yellow). (d) Without proper antialiasing, the fine lines from the reference grid result in "jagged" rendering artifacts (red arrow).

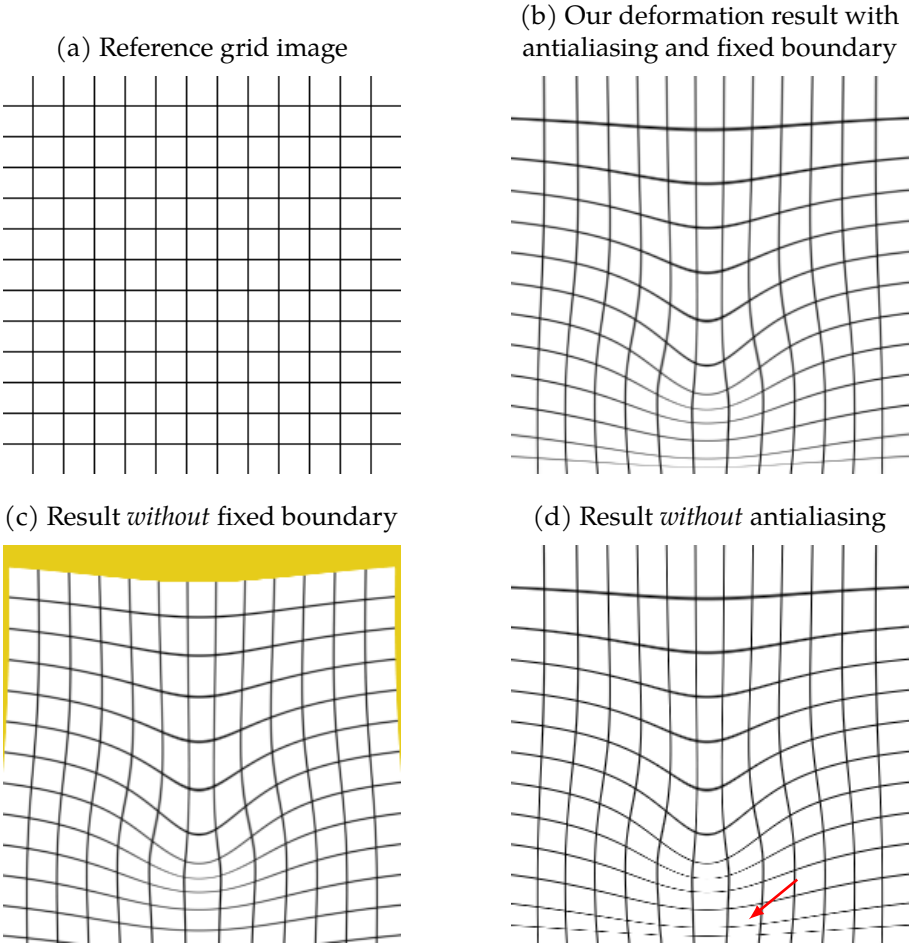


Image Source: The Author

5 IMPLEMENTATION DETAILS

We now describe some relevant implementation details of our image and video warping methods.

5.1 Antialiasing through Anisotropic Filtering

The sample-rate of the output warped image, relative to the input domain, varies locally as a funcion of the deformation. This means that prefiltering must be performed when generating the output pixels (Fig. 4.3b), to avoid aliasing artifacts (Fig. 4.3d).

The deformation's Jacobian matrix $J(p)$, computed at pixel p, encodes the local expansion and contraction of the space around p (SZELISKI, 2011). It can thus be used to identify the proper antialiasing kernels. More precisely, the quadratic form

$$\varphi_{\vec{v}}(p) = 1 + \frac{\vec{v}^{\top} J(p)\, \vec{v}}{\|\vec{v}\|^2} \tag{5.1}$$

describes the local contraction (if $\varphi_{\vec{v}}(p) \in [0, 1)$) or expansion (if $\varphi_{\vec{v}}(p) \in [1, \infty)$) that occurs at pixel p in the direction $\vec{v}$. This concept is illustrated in Fig. 5.1(a–b) for the vertical ($\vec{v} = \vec{y}$) and horizontal ($\vec{v} = \vec{x}$) directions. The direction $\vec{v}_1$ that minimizes $\varphi_{\vec{v}}(p)$ represente the major axis of spatial contraction at p, and can thus be used to define oriented anisotropic antialiasing kernels (Fig. 5.1c). $\vec{v}_1$ is the eigenvector associated with the smallest eigenvalue of $J + J^{\top}$.

In our implementation, we employ a mipmap pyramid for fast antialiasing. We perform anisotropic filtering using multiple samples along the major axis $\vec{v}_1$, while selecting the mipmap level based on the contraction along the minor axis $\vec{v}_2$ (SZELISKI, 2011), which is associated with the largest eigenvalue. This procedure is supported in hardware through the OpenGL's `GL_ARB_texture_filter_anisotropic`.

5.2 Detecting Non-Invertible Deformations

If $\varphi_{\vec{v}_1}(p) < 0$ then the image is being *locally folded-over itself*, meaning that T is not invertible and the warped image is not properly defined. This often occurs if the deformation is too strong. Our implementation emits a warning to the user in

Figure 5.1 – (a) Visualization of the *vertical* contractions (pixels with $\varphi_{\vec{y}}(p) < 1$, in shades of blue) and expansions (pixels with $\varphi_{\vec{x}}(p) > 1$, in shades of red), for the deformation in Fig. 4.3b. (b) A similar visualization for the *horizontal* contractions and expansions. (c) Illustration showing the major axes of spatial contraction $\vec{v}_1$ (black lines), and the corresponding mipmap pyramid levels computed from $\varphi_{\vec{v}_1}$ (colormap).

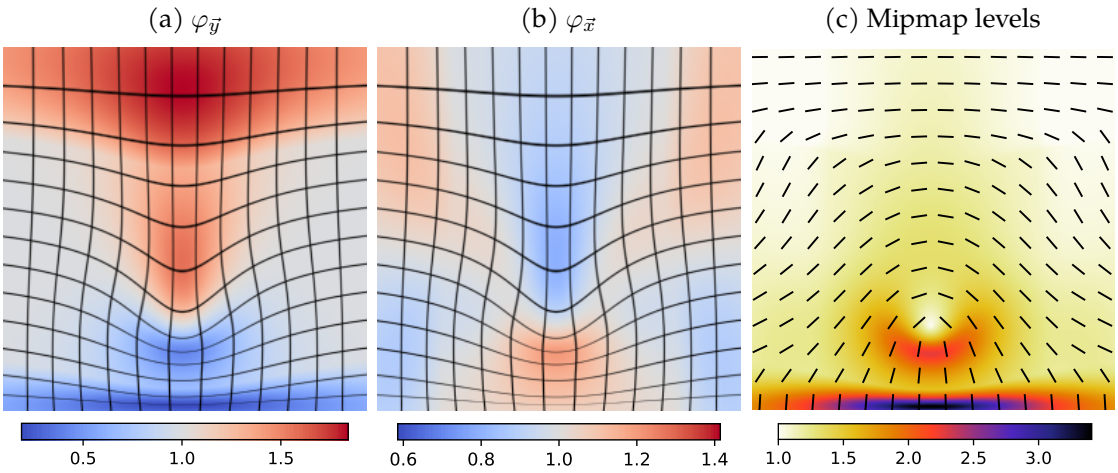


Image Source: The Author

such situations, and automatically dampens the deformation force by the smallest amount required to guarantee invertibility. This is done by replacing $T(p)$ with $T_\alpha(p) = p + \alpha K(p)$ where the scaling $\alpha \in (0, 1)$ is defined as

$$\alpha = \min_p -\frac{1-\epsilon}{\varphi_{\vec{v}_1}(p) - 1}. \tag{5.2}$$

Here, alpha is the highest contraction detected in the vector field. Thus, by damping the deformation such that this α is an invertible contraction, we can avoid non-invertibility. $\epsilon = 10^{-2}$ is a small constant used to avoid numerical issues.

5.3 Video Deformation UI

Video deformation is an unusual application to develop a seamless user interface for. Usual video editing programs have a slider to move around the media with frame-precise interaction. In our implementation, however, we must make sure it is easy to bring a pixel from one frame to another in a fast way. Thus, we propose an interface using the mouse left click to select a pixel, the scroll to move around the frames, deforming the selected pixel or just altering the time value, and the space bar to play the video, which can also be deformed in the XY axis. Since

there is not any major overheads in finding the coordinate of the selected pixel and selecting a frame to visualize, the framerate of our program is still the same. The UI can be visualized at https://inf.ufrgs.br/~eslgastal/gghaetinger/kelvinlets.html.

5.4 Volume Deformation

Volumes are three dimensional data that hold density values in each coordinate, possibly representing a real-world object. Volume visualization is used for many reasons. This is the case for medical diagnosis, to which volume rendering techniques are applied to understand the structure of bodies, *e.g.* broken bones after an accident and malformed organs, detect anomalies such as tumors and sketch surgery steps and approaches. Another use case for volume rendering is to visualize geological structures, *e.g.* oil extraction plans and archeological simulations and discoveries.

Extending this concept, we propose our novel volume deformation approach. Our implementation takes on the challenge of rendering data while applying on-screen "click-and-drag" warping as our algorithm aims for real-time efficiency. We came to the conclusion that this application would be useful for numerous simulation scenarios. An interesting application is for *Plagiocephaly* and *Brachycephaly* (SCHREEN; MATARAZZO, 2013), *i.e.* an assymetrically shaped cranium. Many infants suffer from this illness and there are many solutions to this problem, including surgery and a small head cast. Any of these lead to drastic long or short term changes that need to be evaluated by the children's parents. Our approach could provide a way to foresee the result and ease decision by enabling doctors to deform the patient's virtual cranium to preview what is the expected result of the medical procedure.

Fig. 5.2 shows an example of how the Image warping tool was able to simulate 3 months of using orthesis. We can see how this would be fitting for a 3D volume in Fig. 6.9.

Figure 5.2 – Treatment simulation with the Image Warping tool.

(a) Before treatment | (b) Kelvinlet inverse deform simulation | (c) Real outcome after 3 months of orthesis

Image Source: (*a*, *c*) (*SCHREEN; MATARAZZO, 2013*), (*b*) *The Author*

5.4.1 Volume Rendering

As previously mentioned, our Video visualization implementation uses three-dimensional textures to render the frames along the time axis. This is but a simple variation of regular Volume visualization technique in OpenGL. Thus, we decided to implement Volume deformation by altering the shaders and the interactive process of our Video deformation application.

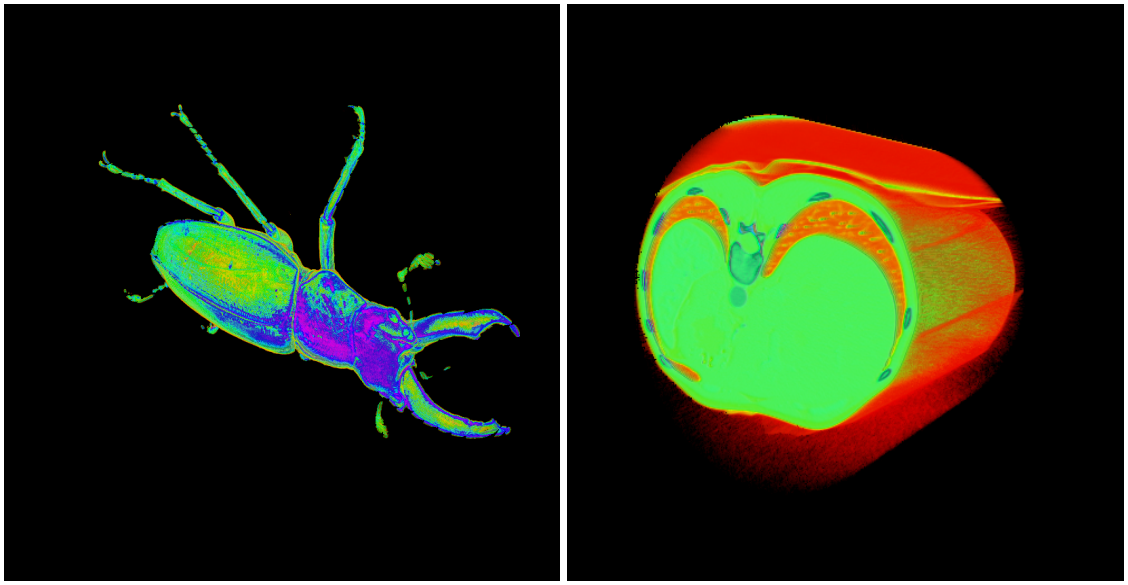
Volume visualization needs much more processing to retrieve correct and insightful visualizations. In spite of its similar domain to Videos, the plain rasterization of the latter does not require knowledge of data except the frame being shown. The former, however, needs us to process the volume by ray marching while capturing the density/value of each point crossed. Nonetheless, as volumes are three-dimensional representation of real world objects, organisms or even a set of particles, we usually want them to see them in perspective projection. To do this, we also designed an OpenGL camera with interactive movement. The camera also gives us a starting point and angle for the casted rays. Our algorithm is described in Algorithm 3 and generates images such as the ones in Fig. 5.3.

It is worth saying that the visualization accuracy and speed is directly related to the length of the interval between points fetched along the ray. A high accuracy might give you well defined edges, but might also come with low framerate.

We have also implemented a density filter to magnify higher density voxels, *i.e.* hardened structures that are, sometimes, in the center of the object. This, of

Figure 5.3 – Simple volumes rendered by Algorithm 3

(a) Stagbeetle (b) Bone



Volume Sources: TU Wien - https://www.cg.tuwien.ac.at/ (Stagbeetle), Dicom Library - https://www.dicomlibrary.com/ (Bone CT Scan)

Algorithm 3 Ray Marching algorithm applied in a fragment

For each fragment point p_f in the normalized device coordinates (NDC):
    Calculate the world fragment position $p_{wf} = \text{VP}^{-1} * p_f$ {VP is the View-Projection Matrix};
    Calculate the volume entry point p_e;
    Trace a ray r from p_{wf} to p_e;
    Find first point along r that has **value** higher than a threshold;
    Perform binary search for optimal border **first point**;
    Capture **values** along ray from optimal point until out of volume or accumulated value over threshold;
    For each captured value v update color value $c \leftarrow (0, 0, 0, 0)$:
        Apply a transfer function to the captured value $c_v \leftarrow f(v)$;
        $c \leftarrow c_v.alpha * c_v.rgb + (1 - c_v.alpha) * c.rgb$
    Return color output $fragment_{color} = c$;

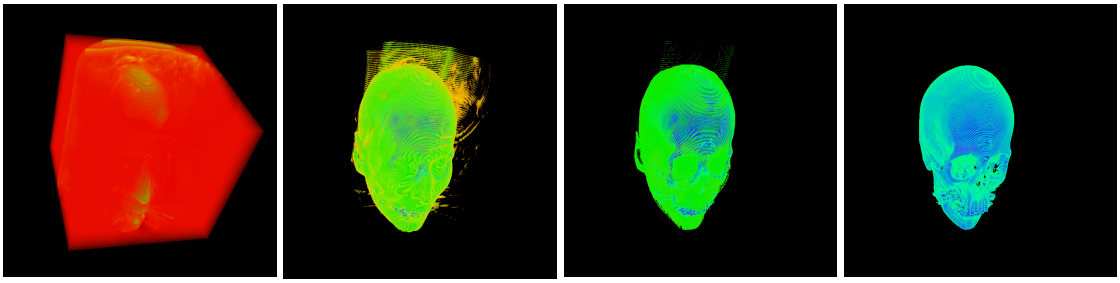
course, slows down our runtime since it needs to go deeper into the volume to find suitable voxels to render. We can see a density progression in Fig. 5.4.

5.4.2 Volume Deformation during Rendering (GLSL)

Although the visualization alone can render volumes quite fast, we now have to account for the volume deformation process. For image or video deforma-

Figure 5.4 – Density visualization applied to the volume representation of a head.

(a) Density = 0.0 (b) Density = 0.1 (c) Density = 0.3 (d) Density = 0.4



Volume Source: Stanford Volume Data Archive
(https://graphics.stanford.edu/data/voldata/)

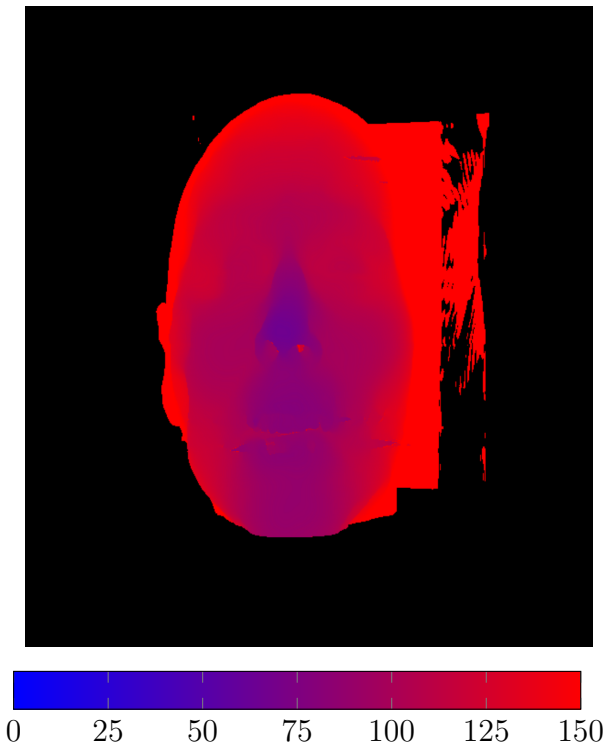
tion, one needs to perform only a single inversion for each rendered fragment. For volume deformation, however, several coordinates from the three-dimensional volume affect the color of each rendered fragment, meaning that one must perform several inversions for each pixel. In other words, we need to apply the inverse deformation for every point we go through in our ray marching algorithm, *i.e.* replace the **red** highlights in Algorithm 3 by the Kelvinlet iterative inversion process.

Thus, comparing to our video deformation process, we have a setback on performance when there are a lot of invertible points being visualized in each traced ray, which is shown in Fig. 5.5. This is not desirable since the inversion process occurs for all frames frames of the visualization and not only during the "click-and-drag" movement. The efficiency decrease is explained when looking at Table 5.1, in which we compare all three applications in terms of operations needed to render a frame.

The interactive deformation process (*"click-and-drag"*) occurs in three steps:

1. We receive the click coordinate in the viewport and trace it to the center of the volume. We set x_0 as the final position of the cast ray with the accumulated voxel value over a given threshold $(0.9$ in our case$)$.
2. The user drags the clicked position and the force is calculated as they move the mouse along the camera uv coordinates, which can be parametrized as world xy coordinates.
3. All the parameters are sent as uniforms to the fragment shader.

Figure 5.5 – A colormap showing how many inversions happen for the volume being deformed in a higher precision and reasonable density filtering. Obs.: The error voxels (To the right side of the head) can have numbers over 150, but we chose to keep this colormap so that the distribution is more visible.

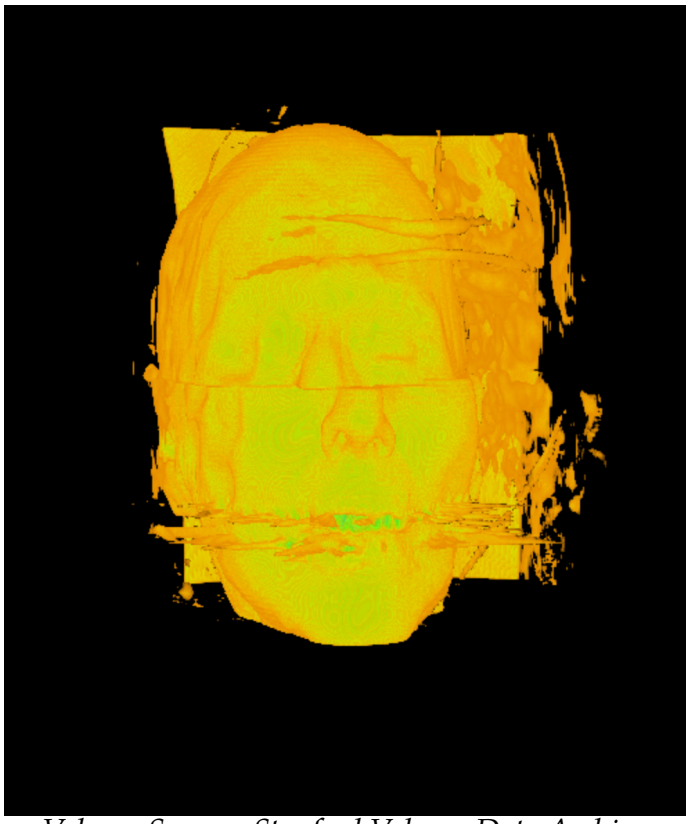


Volume Source: Stanford Volume Data Archive (https://graphics.stanford.edu/data/voldata/)

Table 5.1 – This table shows how Volumes are perceived in terms of needed operations per frame. To grasp the magnitude of it, we exemplify the number of operations for the Mandrill image, the Full HD water serving video and whichever Volume we used (all used the same viewport size). We do so using an average number of iterations 15 for the inversion process and by approximating the number of inversions per fragment in a volume (maximum value at Fig. 5.5).

	Viewport Size	Num. Operations
Image	512×512	$512 * 512 * 15 = 3.932.160$
Video	1920×1080	$1920 * 1080 * 15 = 31.104.000$
Volume	1000×1000	$1000 * 1000 * 150 * 15 = 2.250.000.000$

Figure 5.6 – Image picturing **CUDA**'s interoperability with **OpenGL** generating a scanline through the volume being deformed.



Volume Source: Stanford Volume Data Archive
(https://graphics.stanford.edu/data/voldata/)

5.4.3 Buffered Volume Deformation (CUDA)

To solve the previously described problem, we propose a **CUDA** implementation. In this implementation, we deform the whole volume before rendering it in order to use the simple and fast volume visualization technique Algorithm 3.

To do so, we define two initially equal **CUDA** surfaces, which are readable and writable data structures allocated to hold the Volume data. These artifacts will be sent into a kernel, *i.e.* a function that runs on the GPU, which will read from one and write to the other. We constantly switch which one will be read from and written to so we can keep track of a single volume to be rendered.

We achieved this using blocking and non-blocking **CUDA** streams. The first allows for faster deformation but makes the program stutter until the whole volume has been processed. The latter, shown in Fig. 5.6, allows us to move around while still watching the volume be deformed, but slows down the deformation process by exaustively switching context with **OpenGL**.

5.4.4 Hybrid Deformation (CUDA + GLSL)

Having the two implementations explained in the previous sections, we se that:

- The GLSL solution provides instant feedback, but maintains a lower framerate throughout every interaction. The framerate also decreases immensely with higher accuracy;
- The CUDA solution computes the complete deformed volume in a slow rate, making the interaction feedback infrequent and stuttery. However, once it has completed the deformation, it makes it so that the volume rendering runs smoothly and fast;

Taking a more careful look at the pros and cons of each approach, we see that they are close to complimentary. While one provides interaction during deformation, the other provides fast movement after it. So, it begs the question: Why not join them in a single implementation?

We can reach it by keeping the GLSL implementation during the "click-and-drag" action and triggering the non-blocking **CUDA** computation when the user lets go of the deformation button. In addition to reading from and writing to the texture surfaces, we create a new surface that keeps track of whether the coordinate has been deformed. All points mapping to a coordinate that has been modified will not be calculating the inversion in the fragment shader, but reading from the output surface. The rest will still look for the inverted points along the cast ray. This will make it so we can have progressively faster computations in the fragment shader, by avoiding unnecessary computation. This workflow is better described by the flowchart in Fig. 5.7.

Figure 5.7 – Our Hybrid volume deformation flow. This highlights the essential logic behind it.

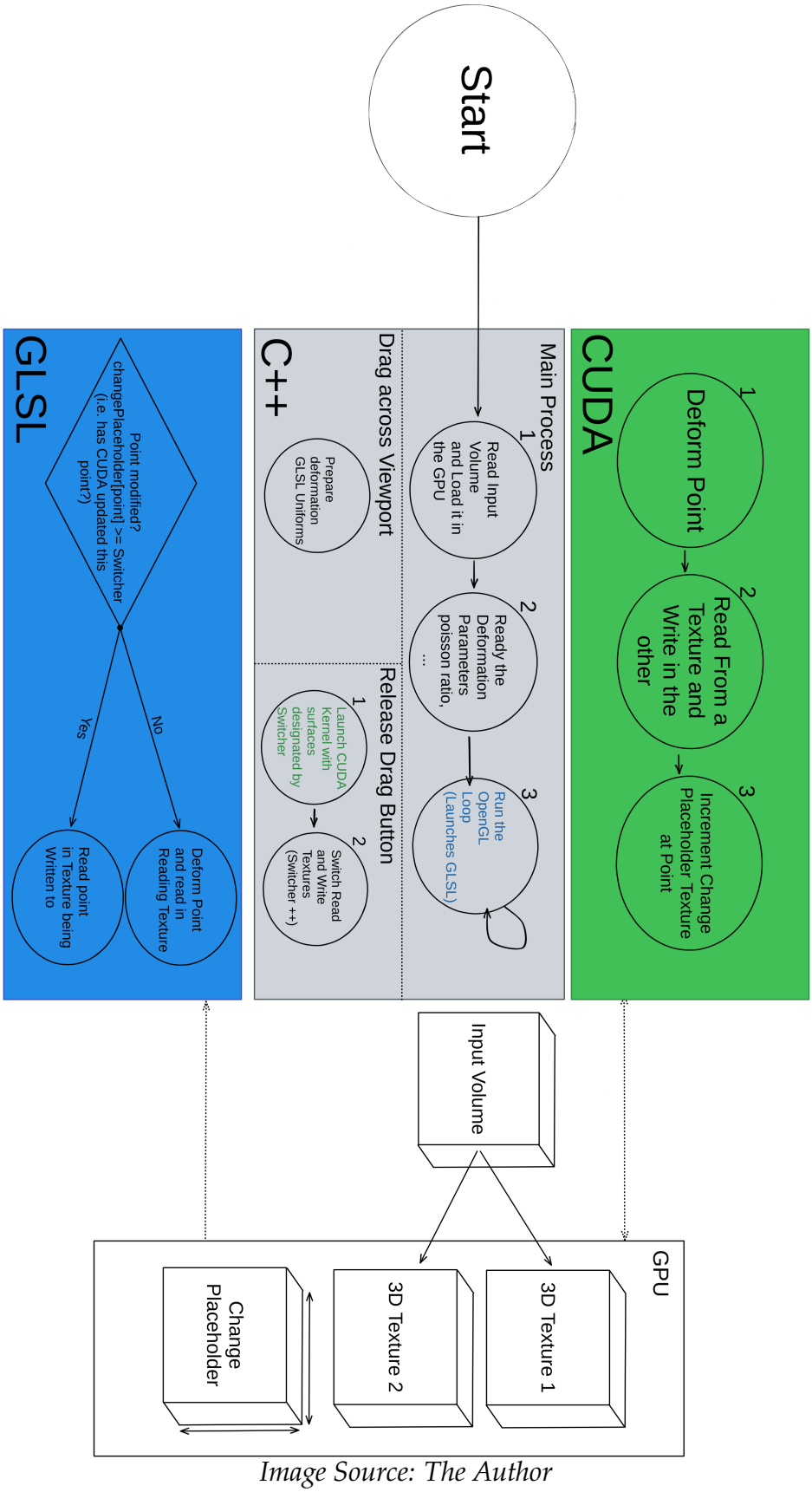


Image Source: The Author

6 RESULTS AND DISCUSSION

We implemented our image deformation method in *Julia*, a just-in-time compiled programming language that focuses on efficiency for scientific computing. Our implementation is multithreaded and generates all output pixels in parallel, achieving high-performance deformations. For example, the execution time for the deformation in Fig. 6.1b (with bilinear prefiltering) is 0.042 seconds on a 6-core 3.2 GHz CPU.

For video time warping, we implemented our method in OpenGL to take advantage of the GPU's high processing power. This allows us to execute the deformation processing and video display in real time (over 300 fps for Full HD 1080p video on an RTX 2070S). We load the video into a 3D texture in GPU memory, and for each (x, y) fragment rendered in the current frame at time t, we find the corresponding point $T^{-1}(x, y, t)$ in the volume to sample its color. The Gauss-Newton iterations are computed in parallel for all pixels in the current frame, during video playback, using a GLSL shader. We solve the small 3×3 linear systems from Eq. (4.3) using GLSL's `inverse()` instruction, which is provided in the hardware.

All image deformation results shown in this work were generated with our Julia implementation, while the video time warping results were generated with our OpenGL implementation. Now, we present results achieved with both methods.

6.1 Image Deformation Results

Fig. 6.1 compares the result of our deformation method to the available alternatives from Krita, an open source image editing software (FOUNDATION, 2020). Our method achieves better results when compared to Krita's warp and liquify tools (see discussion in the figure's caption). Fig. 6.2 shows a similar comparison on a synthetic image, showing that our approach preserves local volume, even while using our fixed boundary constraint. Krita's tool results in a strong shape deformation in the thick black line visible in Fig. 6.2a, and our result shows the line with minimal volume size change and with smooth, antialiased displacement (notice the aliasing artifacts in Krita's result). Our fixed border constraint

Figure 6.1 – Comparison between our inverse Kelvinlet image deformation method and the Krita tools. (b) Our method produces a high-quality deformation result that is not possible to reproduce with Krita. It is able to smoothly translate the eyes and nose of the mandrill downward, while preserving the image's rectangular boundary. (c) Krita's warp tool requires two user interactions and introduces boundary artifacts (green pixels). (d) Krita's liquify tool produces a slighty better result for boundaries, but compresses the nose too much and does not move the eyes. (e) Krita's warp tool generates artifacts if the deformation is too strong (red arrow), since it uses a forward-mapping polygonal mesh for warping. Kelvinlet params.: $p_0 = (256, 256), \vec{f} = (0, -90), \varepsilon = 100$.

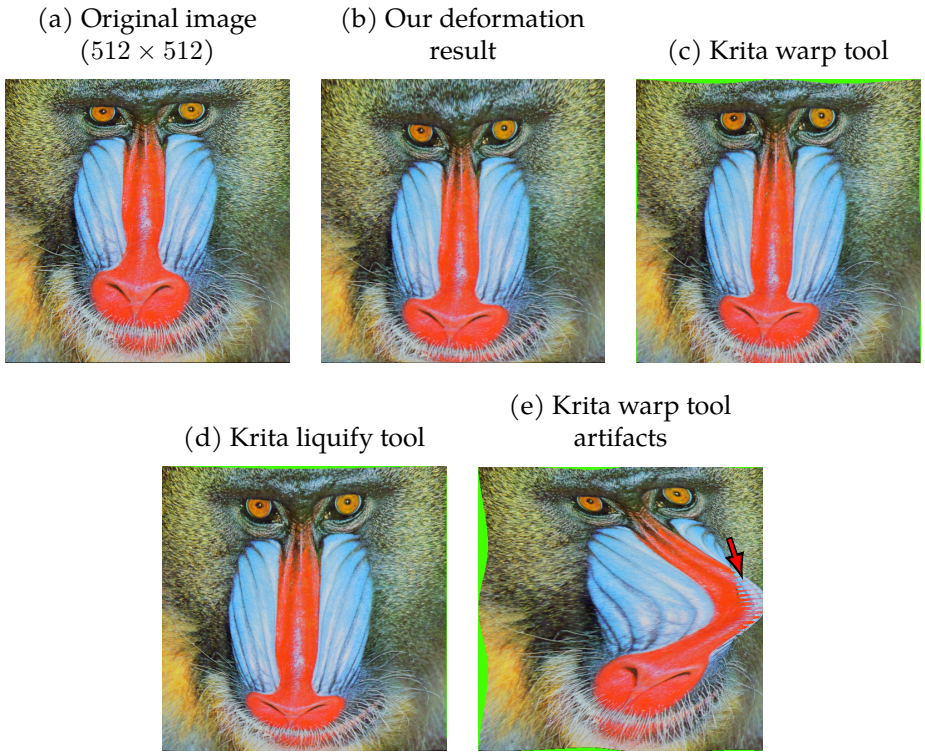


Image Source: Public Domain

also shows to be more rigid (Fig. 6.2b), as we see small portions of background in Fig. 6.2c. As such, our technique is able to generate novel and high-quality deformation results, allowing for a wider range of artistic possibilities.

Fig. 6.3 shows that our *backward-mapping* Kelvinlet warping method works great with fractal images. The fact that it does not need post-deformation interpolation allows us to deform these images with continuous precision, meaning that it maps the output pixel q to the perfect color value of the displaced point, by evaluating the fractal at $T^{-1}(q)$. In contrast, a *forward-mapping* Kelvinlet warping (implemented using the commonly-used triangle mesh technique) results in unwanted blurriness in highly displaced areas. Since the fractal defines a continuous image (broadband signal), all the results shown in Fig. 6.3 were antialiased

Figure 6.2 – Distortion
comparison between our inverse Kelvinlet warping method and the Krita deformation tool. Kelvinlet deformation parameters: $p_0 = (128, 128), \vec{f} = (0, -70), \varepsilon = 30$.

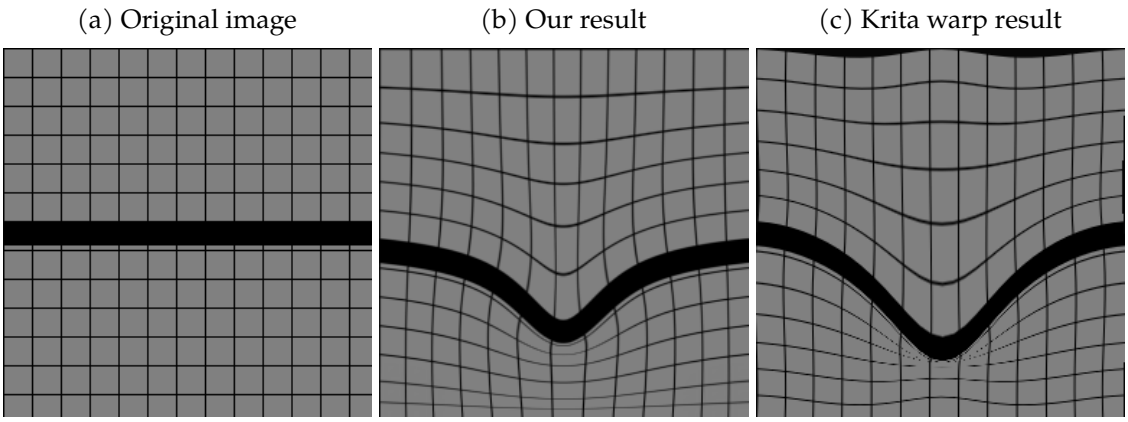


Image Source: The Author

through supersampling.

6.2 Video Time Warping Results

Fig. 6.4 illustrates the idea of time warping in a simple synthetic video sequence, and Fig. 6.5 shows how time warping can be applied to a steady zoom-out video sequence for a real use case: to increase the shot's emphasis on a given object (in this case, the flower).

Fig. 6.6 shows the results of time warping on a Full HD 1920×1080 video of someone pouring water into a cup, used to delay the cup's filling. As we apply a deformation that pushes back the water from filling the middle of the cup, it is visible, by looking at the top of the water level, that the cup is being filled above the deformation point. This is the result we want, since the transition from the top water level to the unfilled gap generated by the displacement is exactly as smooth as the transition between water and air on the original video Therefore, we are able to achieve smooth blending between objects in a scene through time. This result was generated at ~325 frames per second on an RTX 2070 Super GPU.

Figure 6.3 – Deformation of continuous fractal images. (a,d) Original supersampled rendering. (b,e) Our *backward-mapping* method results in a high-quality deformed fractal. (c,f) A *forward-mapping* implementation results in a blurred image. Deformation parameters: $p_0 = (200, 400), \vec{f} = (200, -100), \varepsilon = 100$.

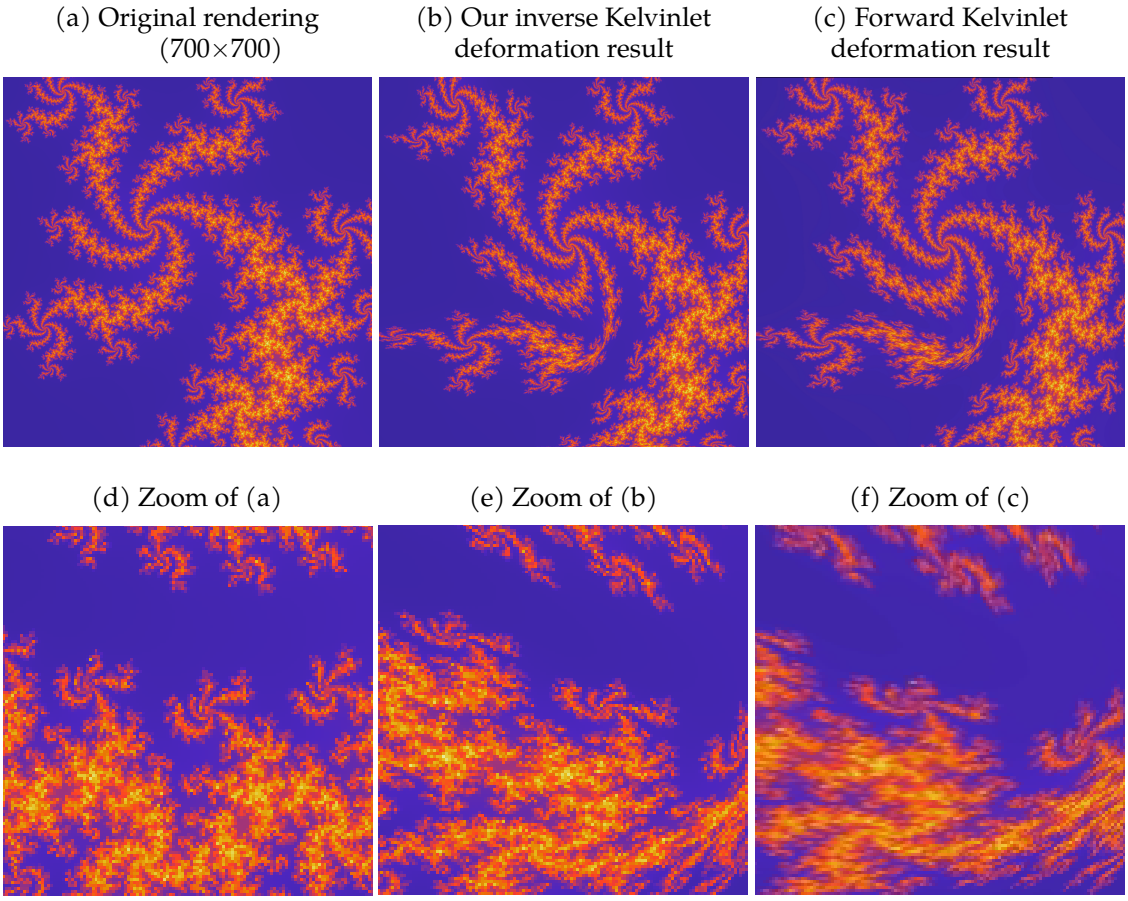
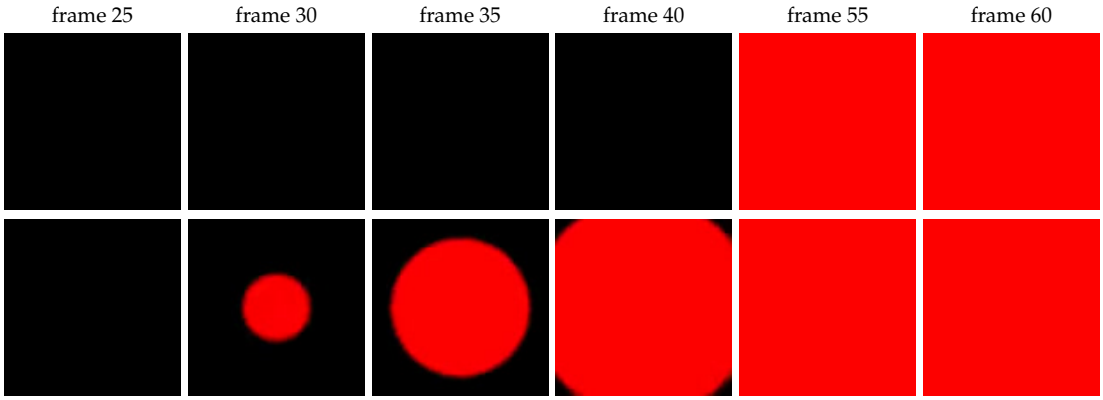


Image Source: The Author

Figure 6.4 – (Top row) Input video consisting of an instantaneous transition from black to red at frame 52. (Bottom row) Smooth transition obtained using our time warping method to bring the red values forward in time. Warping parameters: $p_0 = (50, 50, 50), \vec{f} = (0, 0, -30), \varepsilon = 50$. Video size $50 \times 50 \times 100$.



Video Source: The Author

Figure 6.5 – (Top row) Input video of a flower with a zooming-out camera ($352 \times 288 \times 260$). (Bottom row) Time warped video, maintaining a zoomed-in flower size throughout most of the sequence, and concentrating the zoom-out at the last moments. Warping parameters: $p_0 = (170, 130, 20), \vec{f} = (0, 0, 100), \varepsilon = 80$.

Video Source: Xiph.org

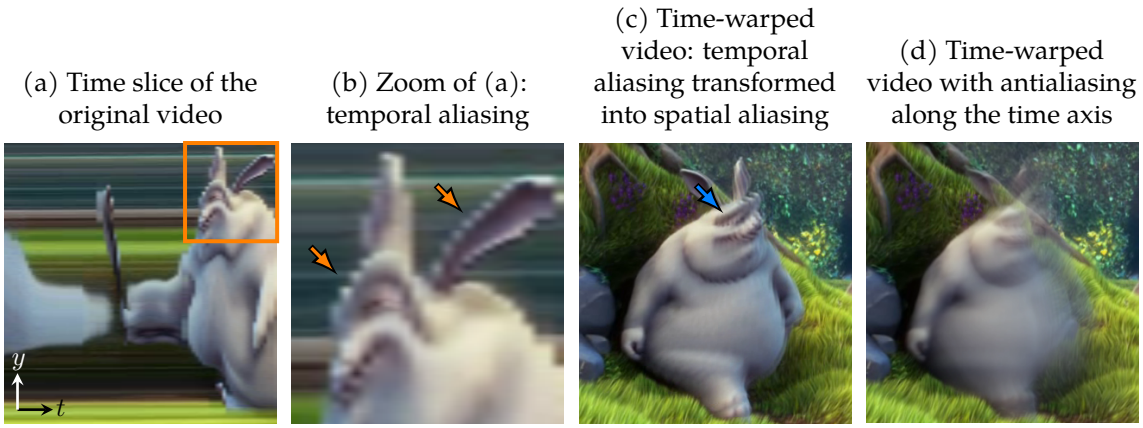
Figure 6.6 – (Top row) Unmodified video of someone pouring water into a cup ($1920 \times 1080 \times 118$). (Bottom row) Same video warped to delay the cup's filling. (See our supplementary materials for the full video sequences). Deformation parameters: $p_0 = (1050, 700, 10), \vec{f} = (0, 0, 60), \varepsilon = 60$.

Video Source: Polina Tankilevitch on Pexels.com

6.2.1 Temporal aliasing

Time sampling is inherently different from spatial sampling (FINLAY; DODWELL, 1987). While our visual system is severely attuned to spatial discontinuities, object motion can be spaced throughout larger pixel intervals (temporal discontinuities) and still be perceived as continuous movement. That is why, depending on the frame-rate and shot steadiness, one can find jagged edges (temporal aliasing) when slicing a video through the time axis (Fig. 6.7(a-b)). Hence, when we combine different frames into a new frame through a warping of the time axis, temporal aliasing is transformed into spatial aliasing, as seen in Fig. 6.7c. This is a

Figure 6.7 – (a) Time slice along the (y, t) plane of a video sequence containing object motion. (b) Zoom of (a), showing the presence of jagged edges, *i.e.* , temporal aliasing (orange arrows). (c) Time-warped video sequence, showing how temporal aliasing has been transformed into spatial aliasing (blue arrow). (d) Applying an antialiasing filter along the time axis removes aliasing artifacts but introduces motion blur. Deformation parameters: $p_0 = (100, 100, 20)$, $\vec{f} = (0, 0, 100)$, $\varepsilon = 60$.



Video Source: Blender Foundation (https://www.blender.org/)

problem that occurs with any time-warping technique, and it can be fixed by capturing the video with a higher frame-rate or by applying a temporal antialiasing filter (Fig. 6.7d). This transforms aliasing into motion blur (KOREIN; BADLER, 1983).

6.3 Volume Deformation Results

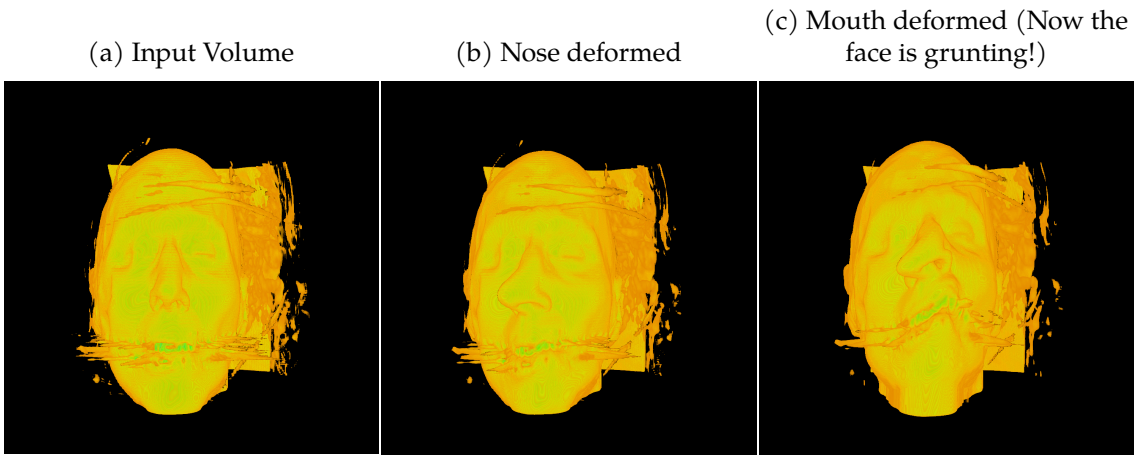
Our volume deformation user interaction has its framerate decrease with the amount of rays marched into the volume, *i.e.* fragments processed. Thus, the efficiency measures aren't really accounted to the volume size, but the window size. In the following paragraphs, we describe a few deformation outcomes.

Obs.: The head volumes used for Fig. 6.8 and Fig. 6.9 have a few capture errors that weren't filtered out. They occur mostly on the back of the head and around the mouth area.

Fig. 6.8 shows how our deformation works with a high precision, displaying very well the face silhouettes. It is worth noting that we are allowed to apply multiple sequential deformations since we store the deformed volume back into the texture/surface.

Fig. 6.9 simulates the inverse process of Fig. 5.2. In it, we use a higher den-

Figure 6.8 – Deformation and high precision applied to a human head volume.



(a) Input Volume (b) Nose deformed (c) Mouth deformed (Now the face is grunting!)

Volume Source: Stanford Volume Data Archive
(https://graphics.stanford.edu/data/voldata/)

sity layer to show how the internal organization of the cranium would compare to the outside, of lower density. It shows how the cranium Fig. 6.9b follows along with the pattern left by Fig. 6.9d.

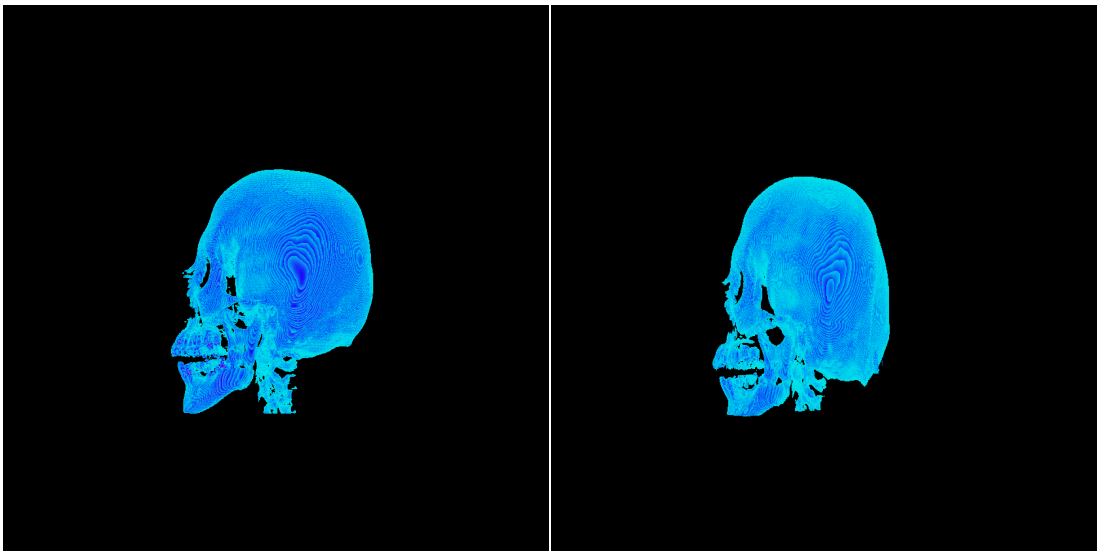
Fig. 6.10 shows the deformation on other volumes. Both other volumes are of larger size.

Table 6.1 lists the metrics for speed in all deformed volumes from this section. All of the rows represent executions in a **GTX 750Ti** GPU with a window size of 1000×1000. By taking a look into the table, we see the problem in each of the approaches. The second column represents the framerate for the **GLSL** based deformation and it is explicit why we needed to find a way to precompute the result: it is very slow. By precomputing the values, we have much faster camera movement, getting up to **60 fps** in a **GTX 750Ti**. Thus, the third column shows why using **CUDA** can make the interactivity a bit better: after a short period of time, we get much better framerate in all cases. The last column presents our hybrid implementation. In it, it is visible there is a decrease in framerate when both **CUDA** and **OpenGL** are running. However, there is an interesting point when looking at the second row: the framerate is higher and the **CUDA** computation time is shorter. That is because the camera, during the deformation, was directed to the batch of voxels that would be precomputed early in the process, which meant the fragment shader wouldn't need to compute that many inversions. Hence, the shader's speed increases and there is more resources to be allocated by **CUDA**.

Figure 6.9 – Deformation applied to volume with density layers representing a human head.

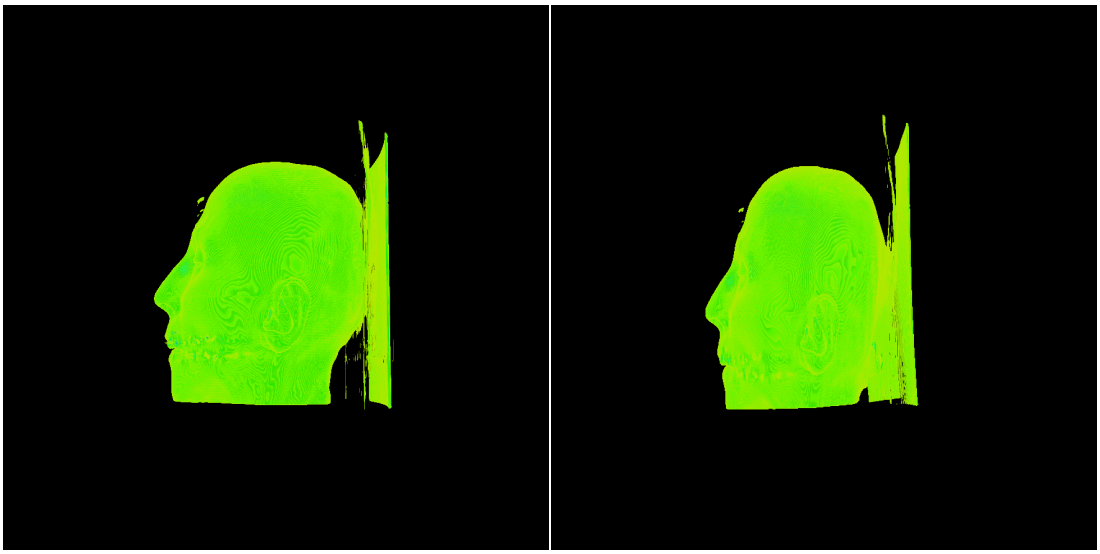
(a) Regular unmodifyied high density portion of the volume.

(b) Deformed high density portion of the volume.



(c) Regular unmodifyied low density portion of the volume.

(d) Deformed low density portion of the volume.



Volume Source: Stanford Volume Data Archive (https://graphics.stanford.edu/data/voldata/)

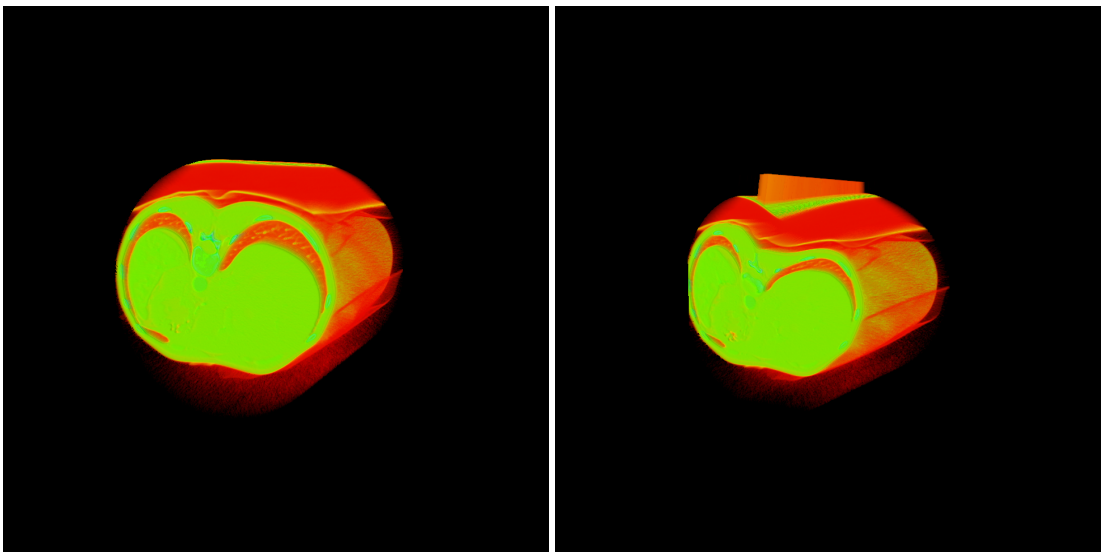
Table 6.1 – Efficiency metrics for the previously displayed images.
*The CUDA implementation allocates 3x more resources than the the GLSL approach. The **GTX 750Ti**'s memory doesn't handle allocating the stagbeetle volume three times.

Figure	GLSL	CUDA	CUDA + GLSL		
	Visualization	Deformation	Deformation	Visualization while deforming	Visualization after deforming
Fig 6.10b	5-20fps	2.18s	35s	10-12 fps	40-60 fps
Fig 6.9	10 fps	0.43s	3s	40 fps	40-60 fps
Fig 6.8	3 fps	0.46s	11s	3-5 fps	40-60 fps
Fig 6.10d	5-10	*	*	*	*

Figure 6.10 – Extra volume deformations examples.

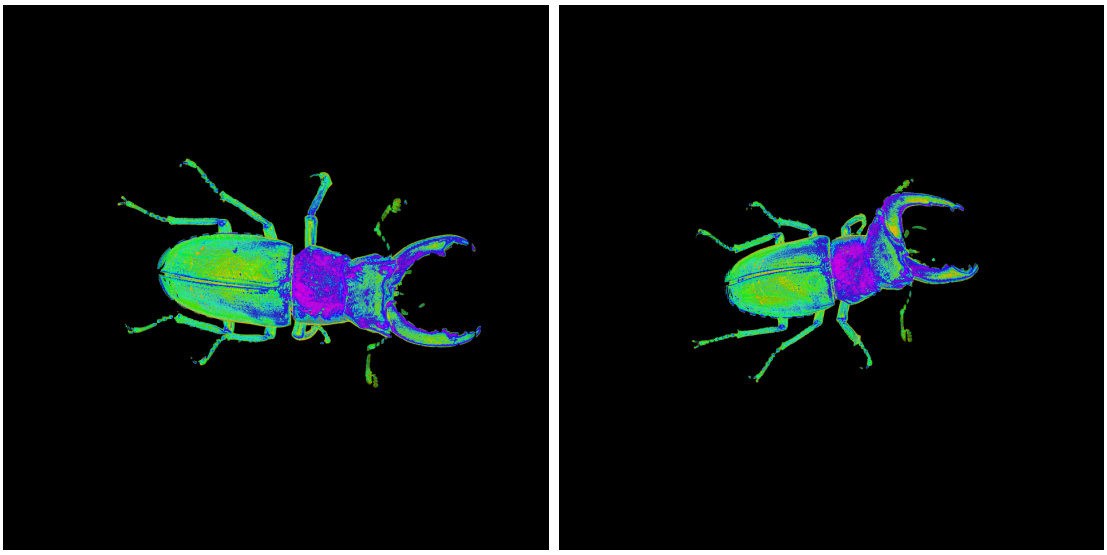
(a) Normal bone volume

(b) Deformed bone volume



(c) Normal stagbeetle volume

(d) Deformed stagbeetle volume



Volume Sources: TU Wien - https://www.cg.tuwien.ac.at/ (Stagbeetle), Dicom Library - https://www.dicomlibrary.com/ (Bone CT Scan)

7 CONCLUSION AND FUTURE WORK

We have presented an algorithm for the inversion of the Regularized Kelvinlet equations, enabling its use for image deformation, video time warping and volume deformation through backward mapping. Inversion is performed by a nonlinear per-pixel optimization process (Gauss-Newton), allowing for fast execution time. Our solution is inherently parallel and achieves real-time performance in Full HD videos, computing deformations at over 300 fps on modern hardware.

We showed how to preserve the image or videos rectangular boundary through a modification of the deformation field, and also how to detect and handle non-invertibility. Furthermore, we discussed the procedures for proper antialiasing of the resulting images, based on local measures of spatial contraction. We discussed the relationship between temporal and spatial aliasing, providing some insights into the limits of time warping. We have also gotten into how to extend our inversion concept to high-dimensional volumes, explaining the throttles and difficulties along with the implementation of possible solutions.

Our technique has a number of applications. Image deformation can be used for interactive photo editing, creating smooth curves and providing a realistic feeling of control over a picture, i.e., warping as if we were touching it. The 3D time warping brush enables the creation of movie reels that change the previously established sequence of frames. Volume deformation has innumerous employments in medical, geological and archeological simulation. Possibilities of future work include extending our solution to other types of deformation fields and the optimization of the volume deformation hybrid schema, using partial low resolution chunks to return faster visual feedback.

8 LESSONS LEARNED

It has been a long time since we started doing this work. In this time, we were able to learn many things both technical and professional. We got to publish our paper, which includes a great part of this work in an international symposium (HAETINGER; GASTAL, 2020). There, we had our work evaluated and awarded for its merits.

This incredible experience was all due to taking the non-naive path for the deformation implementation. We started the research taking the forward deformation as our ground truth and, when the possible flaws and bottlenecks of the post-deformation interpolation were raised to our attention, we decided to explore the inversion method. This was hard and required hard work, but we got amazing results and it was worth it.

Writing a paper that describes a highly graphical algorithm can be hard. Generating a larger number of results helped us with our method explanation.

Writing code that uses OpenGL and CUDA interoperability is hard. It is important to read the documentation and follow any sort of example there is.

REFERENCES

AVIDAN, S.; SHAMIR, A. Seam carving for content-aware image resizing. In: **ACM SIGGRAPH 2007 papers**. [S.l.: s.n.], 2007. p. 10–es.

BACH, B. et al. Time Curves: Folding Time to Visualize Patterns of Temporal Evolution in Data. **IEEE TVCG**, v. 22, n. 1, 2016. ISSN 1077-2626.

BOLLES, R. C.; BAKER, H. H.; MARIMONT, D. H. Epipolar-plane image analysis: An approach to determining structure from motion. **International Journal of Computer Vision**, v. 1, n. 1, p. 7–55, 1987. ISSN 0920-5691, 1573-1405.

CHEN, H.; HESSER, J.; MäNNER, R. Fast volume deformation using inverse-ray-deformation and FFD. p. 8, 2011.

COTIN, S.; DELINGETTE, H.; AYACHE, N. Real-time elastic deformations of soft tissues for surgery simulation. v. 5, n. 1, p. 62–73, 1999. ISSN 10772626. Disponível em: <http://ieeexplore.ieee.org/document/764872/>.

FINLAY, D. J.; DODWELL, P. C. Speed of apparent motion and the wagon-wheel effect. **Perception & Psychophysics**, v. 41, n. 1, 1987. ISSN 0031-5117, 1532-5962.

FOUNDATION, K. **Krita**. 2020. Disponível em: <https://krita.org/en/>.

FREY, S. Spatio-Temporal Contours from Deep Volume Raycasting. **Computer Graphics Forum**, v. 37, n. 3, p. 513–524, 2018. ISSN 1467-8659.

GASCON, J. et al. Fast deformation of volume data using tetrahedral mesh rasterization. In: **SIGGRAPH/EG Symp. on Comput. Animation**. [S.l.: s.n.], 2013. p. 181–185.

GOES, F. D.; JAMES, D. L. Regularized kelvinlets: sculpting brushes based on fundamental solutions of elasticity. **ACM Transactions on Graphics**, v. 36, n. 4, p. 1–11, jul. 2017. ISSN 07300301.

HAETINGER, G. G.; GASTAL, E. S. L. Regularized kelvinlet inversion for real-time image deformation and video time warping. In: **2020 33rd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)**. [S.l.: s.n.], 2020. p. 124–131.

HECKBERT, P. S. Fund. of texture mapping and image warping. Citeseer, 1989.

KARNI, Z.; FREEDMAN, D.; GOTSMAN, C. Energy-Based Image Deformation. **Computer Graphics Forum**, v. 28, n. 5, 2009. ISSN 1467-8659.

KAUFMANN, P. et al. Finite Element Image Warping. **Computer Graphics Forum**, v. 32, n. 2pt1, p. 31–39, 2013. ISSN 1467-8659.

KLEIN, A. W. et al. Stylized video cubes. In: **SIGGRAPH/EG Symp. on Comput. Animation**. [S.l.: s.n.], 2002.

KOREIN, J.; BADLER, N. Temporal anti-aliasing in computer generated animation. In: **SIGGRAPH**. [S.l.: s.n.], 1983. p. 377–388. ISBN 978-0-89791-109-2.

MODAT, M. et al. Inverse-consistent symmetric free form deformation. In: DAWANT, B. M. et al. (Ed.). **Biomedical Image Registration**. Springer Berlin Heidelberg, 2012. v. 7359, p. 79–88. ISBN 978-3-642-31339-4 978-3-642-31340-0. Series Title: Lecture Notes in Computer Science. Disponível em: <http://link.springer.com/10.1007/978-3-642-31340-0_9>.

PFISTER, H. et al. The VolumePro real-time ray-casting system. In: **Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99**. ACM Press, 1999. p. 251–260. ISBN 978-0-201-48560-8. Disponível em: <http://portal.acm.org/citation.cfm?doid=311535.311563>.

RAV-ACHA, A. et al. Evolving Time Fronts: Spatio-Temporal Video Warping. **Proc. 32nd Int. Conf. Comput. Graph. Interactive Tech**, p. 8, 2005.

SCHREEN, G.; MATARAZZO, C. G. Tratamento de plagiocefalia e braquicefalia posicionais com órtese craniana: estudo de caso. **Einstein (São Paulo)**, v. 11, n. 1, p. 114–118, mar. 2013. ISSN 1679-4508. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1679-45082013000100021&lng=pt&tlng=pt>.

SILVA, C. T. et al. A Survey of GPU-Based Volume Rendering of Unstructured Grids. **Revista de informática teórica e aplicada**, n. 2, p. 9–29, 2005.

SOLOMON, J. **Numerical algorithms: methods for computer vision, machine learning, and graphics**. [S.l.]: CRC Press, Taylor & Francis Group, 2015. ISBN 978-1-4822-5188-3.

SOLTESZOVA, V. et al. Memento: Localized Time-Warping for Spatio-Temporal Selection. **Computer Graphics Forum**, v. 39, n. 1, p. 231–243, 2020. ISSN 1467-8659.

SZELISKI, R. **Computer Vision**. [S.l.]: Springer London, 2011. ISBN 978-1-84882-934-3 978-1-84882-935-0.

WOLBERG, G. **Digital Image Warping**. [S.l.]: IEEE computer society press Los Alamitos, CA, 1990. v. 10662.

WYLIE, B. et al. Tetrahedral projection using vertex shaders. In: **Symp. on Vol. Vis. and Graph.** [S.l.: s.n.], 2002. p. 7–12. ISBN 978-0-7803-7641-0.

ZHOU, F.; KANG, S. B.; COHEN, M. F. Time-Mapping Using Space-Time Saliency. In: **CVPR**. [S.l.: s.n.], 2014. p. 3358–3365.

APPENDIX A — NAIVE SOLUTION FOR VIDEO WARPING AND VISUALIZATION

The naive solution for Video Warping and visualization is defined by processing each pixel as a vertex and deforming each of those by using forward warping. This method, however simple it must sound, requires a dense setup phase, in which we structure the vertex arrays to be the size of a frame. This step also takes time, making the own setup for the deformation, a slow operation. In our current solution, by inverse warping and receiving all the data from a texture, we avoid this needless use of memory and time.

Once the arrays are all structured, we deform them individually. What makes this solution hard to use is not the pixel displacement but how to render the *space-time volume* once it is deformed, *i.e.* how to slice down the volume into frames. As cited in the Related Works section in our article, the use of volume rendering is not directly applicable. We implement what we call a *two-step interpolation*, in which we find that the value of a frame's vertex is equal to the interpolation of the volume's vertices that surround it time-wise, *i.e.* the vertices that had the same x, y values before the displacement and that are now placed in front and behind the frame. Once these values are interpolated, we move on to gather all the vertices of each frame, trace a triangle strip that forms an image and interpolate between those points. This algorithm was proved to be extremely inefficient (*45s* to create a 640x368 video) and result unwanted artifacts of discontinuity as shown by.

Fig. A.1 compares our Video time warping solution to a naive forward-warping visualization technique, where we can see that our implementation achieves better results visual quality when the Kelvinlet force field includes xy as well as time translations.

(a) Naively deformed video frame. (b) Inverse deformed video frame.

Figure A.1 – Comparison between the inverse and naive methods. Deformation parameters: $p_0 = \{200, 200, 75\}$, $f = \{100, 100, 60\}, \varepsilon = 70$. Video proportions: 640x368. *Video Source: Blender Foundation* (*https://www.blender.org/*)

APPENDIX B — 2D JACOBIAN MATRIX EQUATIONS

These are the equations for assembling the Jacobian Matrix used for our Image Deformation method. These equations represent the derivatives of the Kelvinlet displacement field for the x and y axes.

$$J(p) = \begin{bmatrix} \frac{\partial K_x}{\partial x}(p) & \frac{\partial K_x}{\partial y}(p) \\ \\ \frac{\partial K_y}{\partial x}(p) & \frac{\partial K_y}{\partial y}(p) \end{bmatrix} \tag{B.1}$$

where

$$f = (f_x, f_y), \tag{B.2}$$

$$p = (x, y), \tag{B.3}$$

$$p_0 = (x_0, y_0), \tag{B.4}$$

and

$$\begin{aligned} \frac{\partial K_x}{\partial x}(p) = c\varepsilon(f_x(&\frac{(a-b)(x_0-x)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(x_0-x)}{2((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} \\ &+\frac{3b(x_0-x)(x-x_0)^2}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{2b(x-x_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}}) \\ &+\frac{bf_y(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_y(x_0-x)(x-x_0)(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) \end{aligned}$$

$$\begin{aligned} \frac{\partial K_y}{\partial x}(p) = c\varepsilon(f_y(&\frac{(a-b)(x_0-x)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(x_0-x)}{2((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} \\ &+\frac{3b(x_0-x)(y-y_0)^2}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{bf_x(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} \\ &+\frac{3bf_x(x_0-x)(x-x_0)(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) \end{aligned}$$

$$\frac{\partial K_x}{\partial y}(p) = c\varepsilon(f_x(\frac{(a-b)(y_0-y)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(y_0-y)}{2((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{3b(x-x_0)^2(y_0-y)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{bf_y(x-x_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_y(x-x_0)(y_0-y)(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_x}{\partial y}(p) = c\varepsilon(f_y(\frac{(a-b)(y_0-y)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(y_0-y)}{2((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{3b(y_0-y)(y-y_0)^2}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{2b(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}}) + \frac{bf_x(x-x_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_x(x-x_0)(y_0-y)(y-y_0)}{((x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

APPENDIX C — 3D JACOBIAN MATRIX EQUATIONS

These are the equations for assembling the Jacobian Matrix used for our Video Time Warping method. These equations represent the derivatives of the Kelvinlet displacement field for the x, y and t (time) axes.

$$J(p) = \begin{bmatrix} \frac{\partial K_x}{\partial x}(p) & \frac{\partial K_x}{\partial y}(p) & \frac{\partial K_x}{\partial t}(p) \\ \frac{\partial K_y}{\partial x}(p) & \frac{\partial K_y}{\partial y}(p) & \frac{\partial K_y}{\partial t}(p) \\ \frac{\partial K_t}{\partial x}(p) & \frac{\partial K_t}{\partial y}(p) & \frac{\partial K_t}{\partial t}(p) \end{bmatrix} \tag{C.1}$$

where

$$f = (f_x, f_y, f_t), \tag{C.2}$$

$$p = (x, y, t), \tag{C.3}$$

$$p_0 = (x_0, y_0, t_0), \tag{C.4}$$

and

$$i = \frac{(a-b)(x_0-x)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(x_0-x)}{2((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}$$

$$j = \frac{(a-b)(y_0-y)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(y_0-y)}{2((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}$$

$$k = \frac{(a-b)(t_0-t)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3a\varepsilon^2(t_0-t)}{2((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}$$

$$\frac{\partial K_x}{\partial x}(p) = c\varepsilon(f_x(i+ \frac{3b(x_0-x)(x-x_0)^2}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{2b(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}}) + \frac{bf_t(t-t_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_t(t-t_0)(x_0-x)(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{bf_y(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_y(x_0-x)(x-x_0)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_y}{\partial x}(p) = c\varepsilon(f_y(i+$$
$$\frac{3b(x_0-x)(y-y_0)^2}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{3bf_t(t-t_0)(x_0-x)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}+$$
$$\frac{bf_x(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_x(x_0-x)(x-x_0)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_t}{\partial x}(p) = c\varepsilon(f_t(i+$$
$$\frac{3b(t-t_0)^2(x_0-x)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{bf_x(t-t_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}}+$$
$$\frac{3bf_x(t-t_0)(x_0-x)(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{3bf_y(t-t_0)(x_0-x)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_x}{\partial y}(p) = c\varepsilon(f_x(j+$$
$$\frac{3b(x-x_0)^2(y_0-y)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{3bf_t(t-t_0)(x-x_0)(y_0-y)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}+$$
$$\frac{bf_y(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_y(x-x_0)(y_0-y)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_y}{\partial y}(p) = c\varepsilon(f_y(j+$$
$$\frac{3b(y_0-y)(y-y_0)^2}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{2b(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}})+$$
$$\frac{bf_t(t-t_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_t(t-t_0)(y_0-y)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}+$$
$$\frac{bf_x(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_x(x-x_0)(y_0-y)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_t}{\partial y}(p) = c\varepsilon(f_t(j+$$
$$\frac{3b(t-t_0)^2(y_0-y)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{3bf_x(t-t_0)(x-x_0)(y_0-y)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}+$$
$$\frac{bf_y(t-t_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_y(t-t_0)(y_0-y)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_x}{\partial t}(p) = c\varepsilon(f_x(k+$$
$$\frac{3b(t_0-t)(x-x_0)^2}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{bf_t(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}}+$$
$$\frac{3bf_t(t_0-t)(t-t_0)(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{3bf_y(t_0-t)(x-x_0)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_y}{\partial t}(p) = c\varepsilon(f_y(k+$$
$$\frac{3b(t_0-t)(y-y_0)^2}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}) + \frac{bf_t(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}}+$$
$$\frac{3bf_t(t_0-t)(t-t_0)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{3bf_x(t_0-t)(x-x_0)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$

$$\frac{\partial K_t}{\partial t}(p) = c\varepsilon(f_t(k+$$
$$\frac{3b(t_0-t)(t-t_0)^2}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}} + \frac{2b(t-t_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}})+$$
$$\frac{bf_x(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_x(t_0-t)(t-t_0)(x-x_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}}+$$
$$\frac{bf_y(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{3/2}} + \frac{3bf_y(t_0-t)(t-t_0)(y-y_0)}{((t_0-t)^2+(x_0-x)^2+(y_0-y)^2+\varepsilon^2)^{5/2}})$$