Ecole Polytechnique - INF585 Computer Animation (2021-2022)
Final Project Report - Article implementation

Vector Field Based Shape Deformations [1]

Wolfram Von Funck, Holger Theisel, Hans-Peter Seidel

Antonin Wattel, Nathan Pollet

I. Introduction

In this project, we work on the implementation of the following research article: *Vector Field Based Shape Deformations* (Wolfram Von Funck, Wolfram and Theisel, Holger and Seidel) [1]. The topic of vector field-based shape deformation was discussed in INF585 in the context of shape deformation but was not implemented nor experimented with, which is a strong motivation factor. The approach presented in the paper, based on the creation of C^1 divergence-free vector fields is of particular interest due to its peculiar properties. Indeed, it is volume preserving, smoothness preserving as well as feature preserving, is free of self-intersections (global or local) and is local. Shape deformation is performed by applying a path line integration along it for all target mesh vertices. For instance, the new (deformed) position of a shape vertex $\mathbf{p}$ is calculated by performing the path line integration of $\mathbf{v}$ starting from $\mathbf{p}$. In order to obtain a local support of the deformation, the vector field $\mathbf{v}$ contains non-zero values only in a certain target area.

In this project, our focus is on implementing part of the method proposed in the academic paper of interest [1] in order to successfully deform a shape in the form of a triangular mesh using vector fields. In particular, we only focus on translation-based deformations. To achieve this goal, several lab classes are combined and enriched to create an interactive environment. We then build a grid, construct the velocity vector field and implement several interactive deformation methods, or metaphors. A user-friendly GUI enables users to quickly experiment and play with the shapes and their deformations in a creative fashion.

II. Approach

The essence of the vector field-based deformation approach resides in the computation of the deformation vector field $\mathbf{v}$. The field needs to be flexible enough to describe a large spectrum of deformations while staying simple enough that it can be computed and updated on-the-fly. $\mathbf{v}$ is constructed as a piecewise 3D vector field lwith the following objectives: 1. $\mathbf{v}$ is non-zero only inside of a target region where it should be simple and well-defined, i.e. constant (characterizing a translation) or linear (characterizing a rotation). This region is referred to in the paper [1] as the *inner region* of deformation, 2. $\mathbf{v}$ is equal to zero in the *outer region* of deformation, leading to a null deformation, 3. in the *intermediate region* which is located between the two regions $\mathbf{v}$ is blended in a globally divergence-free and C^1 continuous way.

The regions are defined implicitly through the definition of a scalar region field $r(\mathbf{x})$ with, in our case, $\mathbf{x} = (x, y, z)$. The inner and the outer regions are defined through spheres with given radius lengths r_i and r_o with $r_i < r_o$. With the following, a point $\mathbf{p}$ is 1. in the *inner region* if $r(\mathbf{p}) < r_i$, 2. in the intermediate region if $r_i \leq r(\mathbf{p}) < r_o$ or 3. in the outer region if $r_o \leq r(\mathbf{p})$. As explained later in the report $\mathbf{v}$ is computed using the cross product of two scalar fields and a blending function given in Bézier representation.

A. Vector field computation

In this subsection, details are given regarding the computation of the vector field in our case. We refer to sections ***3. Constructing the vector field v*** and ***8. Appendix*** of the studied paper [1]. More details, including the proof that $\mathbf{v}$ is C^1 and divergence free, can be found there.

We choose $r(\mathbf{x}) = ||\mathbf{x}, \mathbf{c}||$ the euclidean distance between $\mathbf{c}$ and $\mathbf{x}$. Inside the inner region, $\mathbf{v}$ is constant and describes a translation. Its direction can be determined by the user as we will see in section *III C*. Its length is a chosen constant. To get the constant direction of $\mathbf{v}$, we choose $\mathbf{u}$ and $\mathbf{w}$, two orthogonal unit vectors both orthogonal to the desired direction.

In 3D, a divergence-free vector field v can be constructed from the gradients of two scalar fields $p(x, y, z)$, $q(x, y, z)$ as

$$\mathbf{v}(x, y, z) = \nabla p(x, y, z) \times \nabla q(x, y, z)$$

Let $e(x)$, $f(x)$ be two C^2 continuous scalar fields which are supposed to define $\mathbf{v}$ in the inner region. We can define the piecewise scalar fields p, q as

$$p(\mathbf{x}) = \begin{cases} e(\mathbf{x}) & r(\mathbf{x}) \leq r_i \\ (1-b) \cdot e(\mathbf{x}) & r_i \leq r(\mathbf{x}) \leq r_0 \\ 0 & r_0 \leq r(\mathbf{x}) \end{cases}$$

$$q(\mathbf{x}) = \begin{cases} f(\mathbf{x}) & r(\mathbf{x}) \leq r_i \\ (1-b)\cdot f(\mathbf{x}) & r_i \leq r(\mathbf{x}) \leq r_0 \\ 0 & r_0 \leq r(\mathbf{x}) \end{cases}$$

where $b = b(r(\mathbf{x}))$ is a blending function given in Bezier representation as

$$b(r) = \sum_{i=0}^{4} w_i B_i^4 \left(\frac{r - r_i}{r_o - r_i} \right)$$

B_i^4 being the Bernstein polynomials, $w_0 = w_1 = w_2 = 0$ and $w_3 = w_4 = 1$.

In this project, we focus on a constant vector field $\mathbf{v}$ in the inner region, describing a translation. In this case, the linear scalar fields e, f are defined as

$$e(\mathbf{x}) = \mathbf{u}(\mathbf{x} - c)^T,\ f(\mathbf{x}) = \mathbf{w}(\mathbf{x} - c)^T$$

so

$$\nabla e \equiv \mathbf{u}, \nabla f \equiv \mathbf{w}$$

We have

$$\nabla((1-b)e) = (1-b)\cdot\nabla e - e\nabla b$$

$$\nabla((1-b)f) = (1-b)\cdot\nabla f - f\nabla b$$

Using the chain rule,

$$\nabla b = \nabla b(r(\mathbf{x})) = \frac{db}{dr}\cdot\nabla r$$

and

$$\frac{db}{dr} = \frac{2r}{r_0 - r_i},$$

$$\frac{\partial r}{\partial x_i} = \frac{x_i - c_i}{r}$$

Putting these together yields a closed-form expression for the vector field.

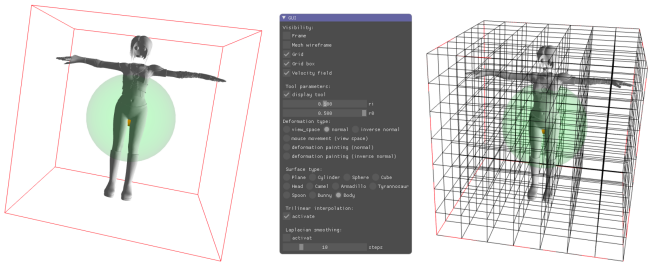
III. Implementation details

In this section, we describe the different features of our implementation. The source code structure is originally based on the lab 3 of the INF585 course [4], providing us with a basic GUI, and an environment to pick points on 3D meshes in order to deform them. It makes use of CGP (Computer Graphics Programming library) [5], a lightweight and minimalist C++ library using OpenGL to represent, animate, and interact in real time with 3D scenes. The implementation features a GUI allowing the user to display different elements, and deform meshes in several ways, as seen in Fig.1.

A link to the source code and compilation instructions is provided.

A. Environment

Meshes are loaded from *.obj* files. On the GUI, buttons are provided to select several of the shapes used during the INF 585 lab sessions as well as meshes from the Stanford scanning repository converted to **.obj** format [6]. All meshes are scaled and rotated appropriately before being presented to the user. With the desired shape loaded, a unit size 3D grid is created. This structure is an essential part of the deformation process, as it will be explained later in the report. The number of grid cells is controlled through a variable which can easily be changed. Setting the number of cells leads to a tradeoff between precision and computation speed. In most of the results presented in this report, we choose 20^3 cells, which allows for precision and real-time interaction for meshes of a reasonable size. The user cannot deform a shape beyond the grid and the grid is, for convenience and visual purposes, contained in a (red) bounding box (Fig. 1).



(a) bounding box (b) grid and GUI

Fig. 1: Grid, bounding box and GUI representations

B. Tool

We first define a spherical tool to apply translation deformations. Its visual representation is similar to the one seen in the paper [1].

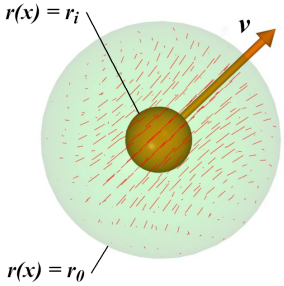


Fig. 2: Deformation tool

Parameters r_i and r_0 can be interactively modified using the GUI. The arrow represents the direction of constant vector field in the inner sphere. When pressing the *shift* button on the keyboard, the user can move the tool on the shape surface. A deformation is applied with *shift* + *click* or *shift* + *click* + *mouse movement* depending on the choice of the transformation method (see *D. Deformations*).

C. Vector field

Our implementation provides different ways to deform a given shape. All are based on a constant vector field in the inner sphere, corresponding to a translation deformation. This vector field can be chosen arbitrarily. During the vector field computation, **u** and **w** are defined as orthogonal vectors to the constant velocity inside the inner sphere.

- *view space* : the constant velocity direction is defined as the 3D translation implied from the vector $\begin{pmatrix}0\\1\end{pmatrix}$ in the camera plane.
- *normal (resp. inverse normal) / deformation painting* : **v** is the normal (resp inverse normal) of the shape at the picked position.
- *mouse movement* : the direction of **v** is defined as the 3D translation implied from the mouse displacement in the camera plane.

The velocity field is computed for each cell of the grid inside the tool, using formulas described in the previous section. On Fig. 2, we can see a visual representation of this field, shown by red segments. The vector field is updated every time it is needed. After moving the tool position or changing its parameters r_i or r_0, After we move the camera when the field depends on the camera plane, in the *view space* or *mouse movement* cases.

D. Deformations

When the user induces a deformation, one or several steps of numerical integration of the velocity are carried out. We use a simple 1st order Explicit Euler scheme. the position **p** of each point is updated as follows (h being the step size). This differs from the paper, which uses a 4-th order Runge-Kutta path-line integration.

$$\mathbf{p}^{k+1} = \mathbf{p}^k + h\mathbf{v}^k$$

For the user, there are two main ways of deforming the shape. first, when choosing *view space*, *normal* or *inverse normal*, deformations are applied at each *shift + click* from the user. When choosing *mouse movement*, or *deformation painting*, deformations additionally depend on a movement of the mouse. These methods provide a more intuitive and interactive way to deform the object.

Mouse movement: For this method, the idea is that the deformation follows the movement of the mouse of the user, projected in the screen space. We fix $\Delta r < ri$. When the tool moves by Δr, the integration inside the inner region moves the points by Δr as well. While the mouse is moving, every time the tool moves by Δr, we carry integration steps, and use the trick of adjusting the velocity magnitude. the speed at which the mouse is moved does not matter. Fig. 3 shows an example of use of this method.

This method, however, gives bad results for too strong deformations, specifically when they are curved, as can be seen on

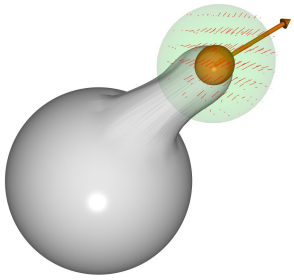
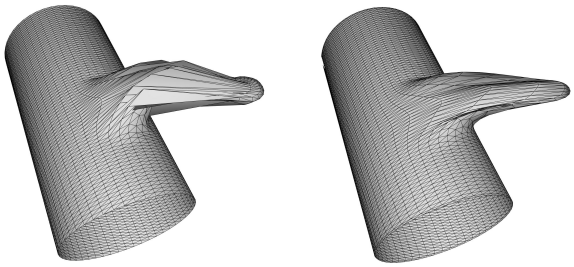


Fig. 3: An example of deformation using the *mouse movement method*

Fig. 4. Some artifacts are caused by an undersampling of the surface. A smoother shape can be acheived by using laplacian smoothing (see *F.*) after applying the deformation, but can lead to a loss of the desired properties. For this reason, an important improvement would be to use remeshing. In our case, it is better to work with only small deformations, and for more precision to use the other tools that perform 'one click at a time'. Moreover, the inner region follows the mouse movement, or more precisely the center of the tool **c**, but does not change its orientation, which can be unintuitive and lead to to unpleasing results. To correct this, one could define the inner deformation as a rotation around a rotational axis perpendicular to the osculating plane of c and passing through the curvature center of **c**, as seen in [2].

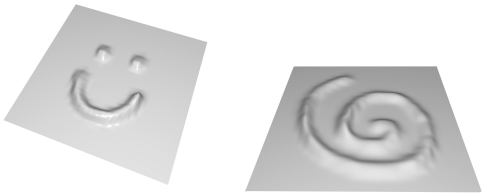


(a) w/o smoothing (b) with smoothing

Fig. 4: An example of large deformation using *mouse movement*

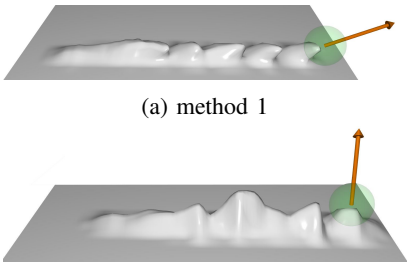
Deformation painting: This method is based on the idea of section ***4.2 Deformation Painting*** of the paper. The idea is to move the tool along a path on the surface of the shape, and deform it on this path. We use a constant **v** in the direction of the surface normal (or opposite to it), and set the inner radius $r_0 = 0$. As for the *Mouse movement* method, we carry an integration step every time the tool moves by a chosen Δr, but here, the tool moves along the surface normal. We provide two different implementations of deformation painting. The first one recomputes the normals of the deformed shape after each deformation. It can lead to the formation of 'spikes' if the user does not have a good control of the tool, due to the fact that the tool picking points

on the surface of the shape, if the tool is in a position where the normal is directed too upwards. The second one uses the normals before the deformation of the shape, without recomputing them at each deformation. this leads to a more regular and controllable tool, but can feel less natural, and can aldo lead to unpleasing artifacts.



(a) (b)

Fig. 5: Examples of deformations using the *deformation painting* method (in the normal direction)



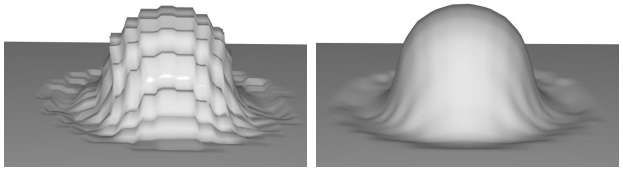
(a) method 1

(b) method 2

Fig. 6: Undesired effects of our deformation painting implementation

E. Trilinear interpolation

Trilinear interpolation was implemented in order to have more accurate velocity values for any point inside the grid. Figures 7 shows results of deformations with and without trilinear interpolation. We can visually observe the significant improvements brought by the addition of the interpolation method.



(a) no interpolation (b) trilinear interpolation

Fig. 7: Trilinear interpolation

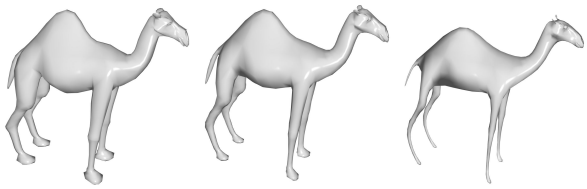
F. Laplacian smoothing

In the paper, the problem of surface undersampling due to large deformations is dealt with by remeshing. This implementation was out of the scope of our project. In order to smooth out discrepancies (e.g. sharp angles) and thereby improve the visual outcome, *Laplacian smoothing* is implemented. This algorithm is aimed at smoothing out a polygonal mesh by calculating for every vertex a new position based on local information (i.e. the position of neighbors). *Laplacian smoothing* can be activated or deactivated by the user and its number of smoothing steps can be controlled in the GUI.

Although its application results in much nicer visual outputs (see Fig. 4), the smoothing method leads to the loss of important information such as surface details (e.g. skin details). This is particularly visible when observing the deformation of the Tyrannosaur mesh with Laplacian smoothing (see Fig 8). For meshes with low poly count, and thin features, too many steps of laplacian smoothing can also lead to undesired effects (see Fig. 9).

(a) undeformed (b) w/o smoothing (c) with smoothing

Fig. 8: Example of loss of local details using laplacian smoothing



(a) no smoothing (b) 30 steps (c) 120 steps

Fig. 9: Example of undesired effects of laplacian smoothing

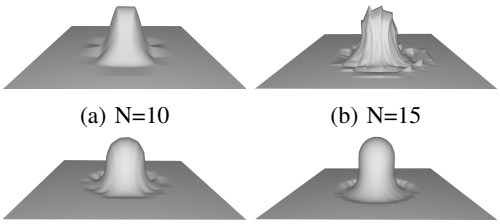
IV. Evaluation and Assessment

A. visual results

Overall, as seen in the different figures, the visual quality of the deformation is satisfying, as long as the deformations are not too large. Some Improvements and tuning could be added to certain of our methods, in particular for *deformation painting*. Thanks to the use of trilinear interpolation of the velocity field values, as well as laplacian smoothing, choosing a grid of size 20^3 cells can already give very satisfying results (Fig. 10), and allows for a smooth real-time user experience for meshes with a low number of points.

B. time performance

As the meshes get more complex, the grid size increases, and the number of integration steps increases, a real-time interaction becomes harder. Moreover, using laplacian smoothing at each deformation step for interactive methods such as *deformation painting* is not suited for real time interaction.



(a) N=10 (b) N=15

(c) N=20 (d) N=30

Fig. 10: Visual comparison or results depending on the grid resolution (N^3 cells)

C. Volume preservation

The absence of remeshing and the possible use of Laplacian smoothing as a (limited) substitute can corrupt the natural volume preservation property of vector field based deformation. In order to assess to what extent mesh volume is being preserved, volume computations are performed. Intuitively, to calculate the volume of a 3D mesh, one can transform the structure into its volumetric representation and then estimate the volume in the voxel space. However, transforming a 3D mesh model into its volumetric representation is time-consuming and can lead to large storage requirements. It is therefore easier to estimate the volume directly from the mesh representation, using elementary shapes such as tetrahedrons and triangles [3]. This way, the computational complexity is proportional to the number of elementary shapes, which is typically much smaller than the number of voxels in the equivalent volumetric representation. A straightforward implementation of the approach in [3] is used to obtain the volume of a mesh from the sum of the (signed) volumes of elementary tetrahedrons formed by connecting each mesh triangle to the origin, as seen in Fig. 11. Using this method, the mesh volume is obtained directly from the mesh *position* and *connectivity*.

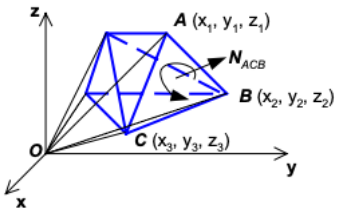
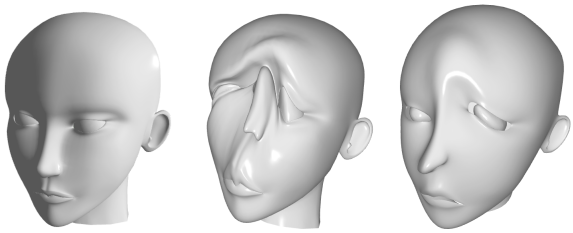


Fig. 11: Elementary tetrahedrons for the computation of the mesh volume

Example deformation results are visible Table 1 along with their corresponding mesh deformation images (Fig. 8 and Fig. 12). For both model shapes (here the Tyrannosaur and the Head meshes), the *Average Volume Error* is estimated with and without Laplacian smoothing. In the case of the Tyrannosaur, the error is computed for a 16-step normal deformation applied at a single point. For the Head, it is computed for a vertical and centered deformation painting (normal) motion.

model shape	deformed shape	avg volume error	smoothing
Tyrannosaur	Figure 8b.	-0.067	None
Tyrannosaur	Figure 8c.	-0.029	Laplacian
Head	Figure 9b.	-0.0062	None
Head	Figure 9c.	-0.058	Laplacian

TABLE I: Summary results for volume preservation experiments



(a) undeformed (b) w/o smoothing (c) with smoothing

Fig. 12: Volume preservation deformation results for Head model shape with tool parameters $r_o = 0.258$

For both model shapes we can observe reasonably low average volume error values. Nevertheless, these results are higher than the original paper's [1]. Again, this is largely due to the undersampling of the shapes due to large deformations.

V. Conclusion

During this project, we implemented some aspects of the paper *Vector Field Based Shape Deformations* (Wolfram Von Funck, Wolfram and Theisel, Holger and Seidel) [1]. We focused on the implementation of a C^1 divergence free vector field, and implemented several interactive ways to use this vector field for mesh deformations. The remarkable properties of this approach make it a useful in many applications. Further improvements to this work could be the implementation of remeshing, the use of acceleration structures (e.g. moving away from the of a linear constant and symmetric grid), GPU support, the implementation of better performing integration methods such as Runge–Kutta methods, as well as the implementation of other types of vectors fields to perform twisting and bending transformations.

References

[1] Wolfram Von Funck, Wolfram and Theisel, Holger and Seidel, Hans-Peter, Vector Field Based Shape Deformations, 2006, Association for Computing Machinery, https://doi.org/10.1145/1141911.1142002

[2] Wolfram von Funck, Holger Theisel, and Hans-Peter Seidel. Explicit control of vector field based shape deformations. In Proceed- ings of Pacific Graphics 2007, 2007. https://doi.org/10.1109/PG.2007.26

[3] Cha Zhang and Tsuhan Chen, Efficient feature extraction for 2D/3D objects in mesh representation, 2001, Carnegie Mellon University, https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.9775

[4] Damien Rohmer, INF585 - Computer Animation, lab 3 - Manual deformers https://imagecomputing.net/damien.rohmer/teaching/inf585/practice/content/004/index.html

[5] Damien Rohmer, CGP - Computer Graphics Programming library https://github.com/drohmer/CGP

[6] .obj models from the stanford scanning repository https://www.prinmath.com/csci5229/OBJ/index.html