# Cage-based deformation with mean value coordinates

Benjamin Barral

With Nicolas Kiefer

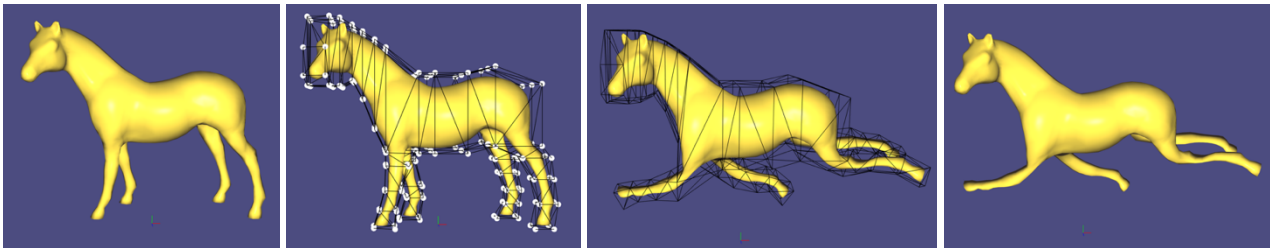


*Figure 1 : Left to right : Original mesh - Cage with moving handles - Cage after deformation - Mesh after deformation*

The purpose of this project is to simulate real-time deformations and animations of dense meshes by manipulating a coarser version of it (called a cage), using coordinate interpolation.
Our work was divided in three parts :
1) The creation of the cages :
- we hand-modelled cages using a modelling software
- we implemented an automatic cage generator algorithm, inspired by the work of Xian et al. [1]
2) The computation of the coordinate interpolation weight : we implemented the Mean Value Coordinate algorithm from Ju et al. [2].
3) The user-cage interaction (UI) based on joint-like handles : manipulating the cage in order to deform the dense mesh through the interpolation pipeline.
My efforts were focused on the implementation of the automatic cage generator (1), as well as the UI and cage manipulation (3). I will therefore emphasize specifically on these two aspects in this report. In section 2 I dive into the theory of the Mean Value coordinates, and provide some results.

We claim that the novelty/extension brought by this project come from the implementation of a method to generate coarse cages automatically, as well as the user interface and cage manipulation system we created. Additionally, we processed a rather large number of different meshes, and produced some deformations we believe to be original and visually interesting (see section 2).
On another note, we wish to stress that a particular effort was put on code design and structure : we created different classes (4 in total) for the different blocks of the project (see Appendix 2 for a detailed description of the classes).

Important : a video was uploaded on UCL Mediacentral as part of the project, which shows the whole program running as well as a demonstration of the cage-based deformation system we designed, and results. The link is : https://mediacentral.ucl.ac.uk/Play/18043

## 1) Automated bounding cage generator[1]

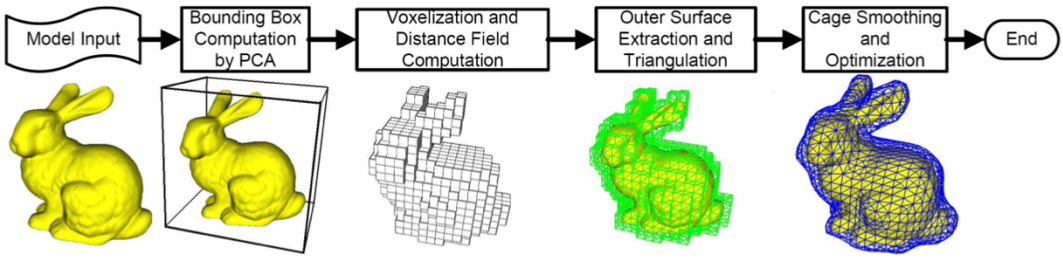



*Figure 2 : The algorithm workflow for automatic bounding cage generation, from [1]*

[1] The code for this part is in the class CageGenerator.

I implemented the method proposed by Xian et al. [1] to generate bounding cages from a dense mesh automatically.
The main steps in this method are :
a) Computing a model-aligned bounding box
b) Sampling the bounding box into voxels, and intersecting the model with the inner voxels
c) Extracting the outer surface of the mesh-intersecting voxels
d) Smoothing the so-generated voxelized cage

In the following subsections I will detail my implementation of each step, going through my different contributions, differentiations and specifications – namely the details missing from the paper that I had to figure out or adapt.

## a) Computing a model-aligned bounding box with PCA

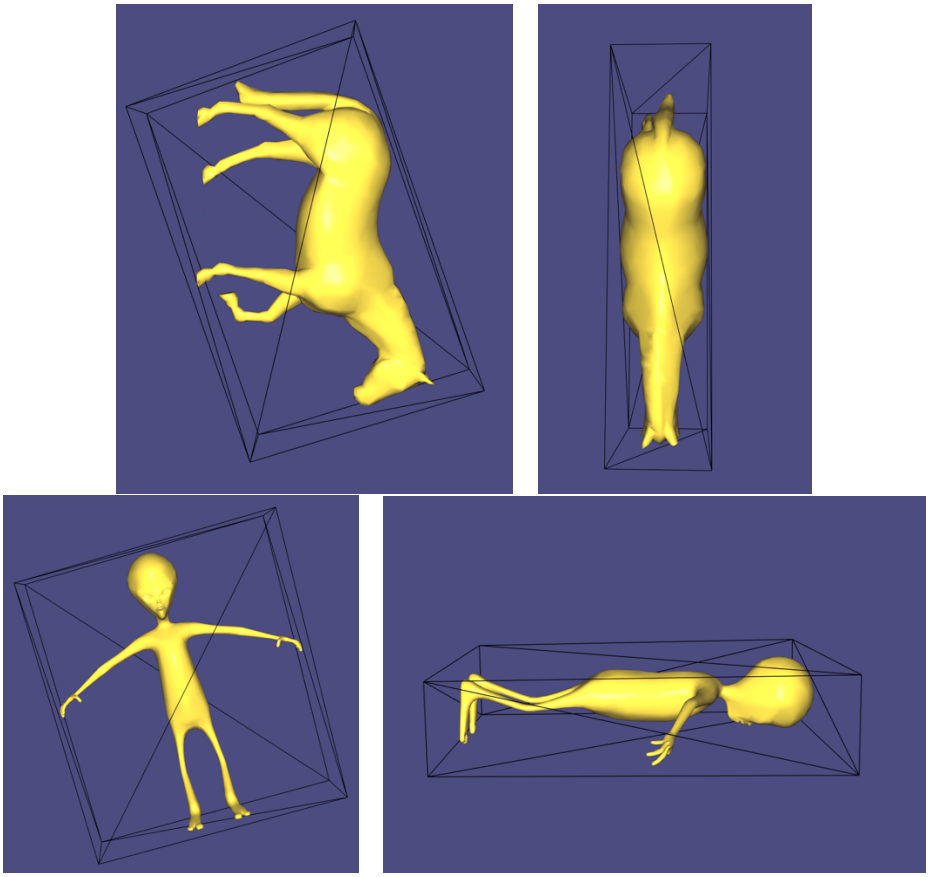


*Figure 3 : Examples of mesh-aligned bounding boxes*

In order to generate a bounding box which is aligned with the mesh, the first step of [1] is to compute the principal directions of the mesh using the PCA (Principal Component Analysis) algorithm :
- we compute the covariance matrix :

$$C = \tilde{V}^T * \tilde{V} \quad \text{where } \tilde{V}_i = V_i - \bar{V} \quad \textbf{(1)}$$

($\bar{V}$ being the mesh barycenter, $V$ of size Nx3 with N vertices, C of size 3x3).
- we compute the eigen value decomposition of C : $C = U\Delta U^T$

Therefore the eigenvectors (columns of U) are the principal directions, and the scale of the eigenvalues represents how spread the data is in that direction.
We then convert the data in the new C.S., in which the whole cage computation will be done :

$$v' = U^T v \quad \textbf{(2)}$$ ($v$ is 3x1, the mesh vertex world position).[2]

Then we calculate the bounding box in this C.S. (c.f. figure 3).

[2] When later transforming the voxels into the original C.S., the inverse transformation is applied : $v = Uv'$.

Note that it is helpful to make the bounding box slightly larger for the cage to wrap the mesh correctly.

### b) Voxelization and feature voxels

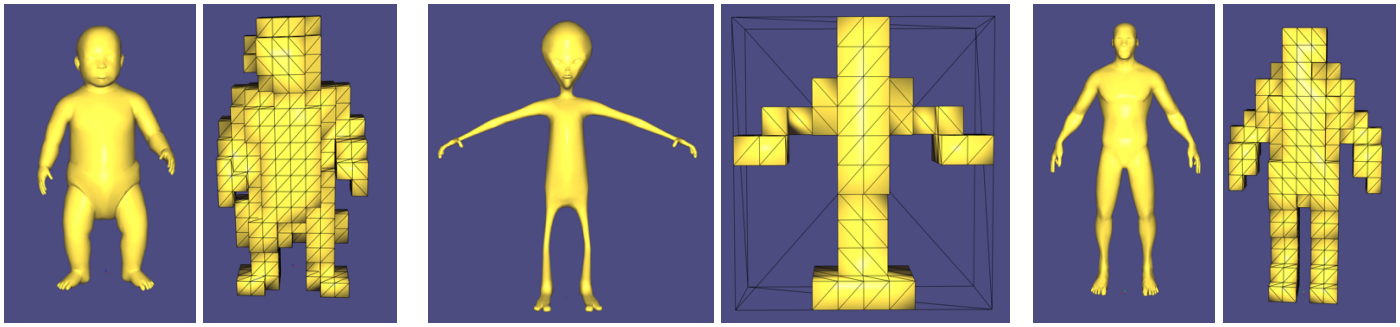


*Figure 4 : Meshes with their computed feature voxels*
*Left : $s_F$ = 0.2 – Middle : $s_F$ = 0.1 – Right : : $s_F$ = 0.2*

After generating the mesh-aligned bounding box, we sample it into voxels.
Importantly, at this point the voxel resolution will determine the ultimate coarseness of the cage.
We use Xian et al.'s definition [1] using the sparse factor $s_F$ (i.e. how sparse the cage will be relative to the dense mesh) :

$$n = \sqrt{\frac{N_D\, s_F}{6}} \quad \textbf{(3)}$$

where $n$ is the voxel resolution on the largest axis, $N_D$ is the number of vertices of the dense mesh.
We then set the resolution on the other dimensions so that we get square voxels.
We set the sparse factor as a modifiable parameter in our implementation.
Note : a large sparse factor means a rather dense cage, and therefore a large computation time for Mean value coordinates, but a too small factor $s_F$ leads to limited cages that do not represent the mesh correctly.

The next step of the algorithm consists in computing the "feature voxels" [1], i.e. the voxels which intersect with the mesh.
For this we use the "Filling algorithm" [1] which distinguishes the voxels into 3 types : inner, outer and feature voxels. Starting from strategic seeds (the 8 corners of the bounding box) which are necessarily outer voxels, we expand by recursively visiting the neighboring voxels (depth first search) until we get out of the bounding box, or the current voxel intersects the mesh. We set the value of the explored voxels to 1 (outer voxels), that of the feature voxels to 0, and that of the unvisited (inner voxels) to -1.
To determine whether a mesh triangle intersects with a voxel, my method consists in :
- clipping the triangle edges against the 6 voxel faces
- at each clipping step, if the new vertex (introduced by the clipping) is on the voxel surface, there is intersection
- if the three vertices of the triangle (updated with the clipping) are on the positive plane of a face, there is no intersection

The pseudo code is the following :

```
bool VoxelTriangleIntersection(voxel, triangle) :
        clippedTriangle = triangle
        For each face in voxel :
                If AllPositivePlane(face,clippedTriangle) : return false
                For each edge in triangle :
                        (newPoint, clippedTriangle) = ClipEdge(edge,face, clippedTriangle)
                        if IsOnVoxelSurface(newPoint,voxel) return true
```

Figure 4 shows the output of this step.

### c) Outer surface extraction from the feature voxels

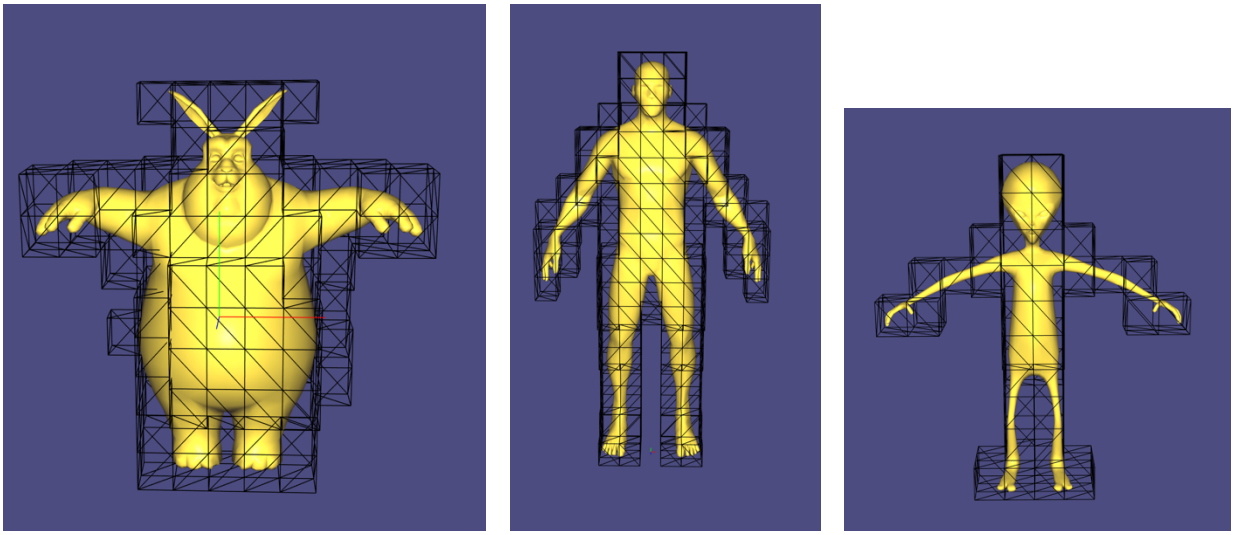


*Figure 5 : Meshes with their extracted outer surface*
*The surface is « voxelized ». Not that faces are triangular at this point*

After finding the feature voxels, the next step of the algorithm consists in computing the bounding cage surface, using the voxel type information from the Filling algorithm.
The surface faces are the faces that separate feature voxels (value 0) from outer voxels (value 1). Therefore, this step consists in looking at the values of the feature voxels' neighbors.
It is in this step that we triangulate the faces from the voxels, by simply creating a diagonal.
It is crucial to make sure that these generated faces are oriented in the same way (e.g. counter-clockwise) for the implementation of the Mean Value Coordinates to work correctly.
In my implementation, I therefore distinguished between the 6 cases corresponding to each neighboring direction (i.e. +X / -X / +Y…) in order to define the faces in the proper ordering.

Note: when feature voxels lie on the bounding box surface, they have no adjacent "outer voxel" within the bounding box. It was a special case I had to account for in my implementation.

Therefore, this point of the algorithm, the output is a cage that wraps the dense mesh in a scraggly and "step-like" way (c.f. figure 5).

### d) Cage smoothing

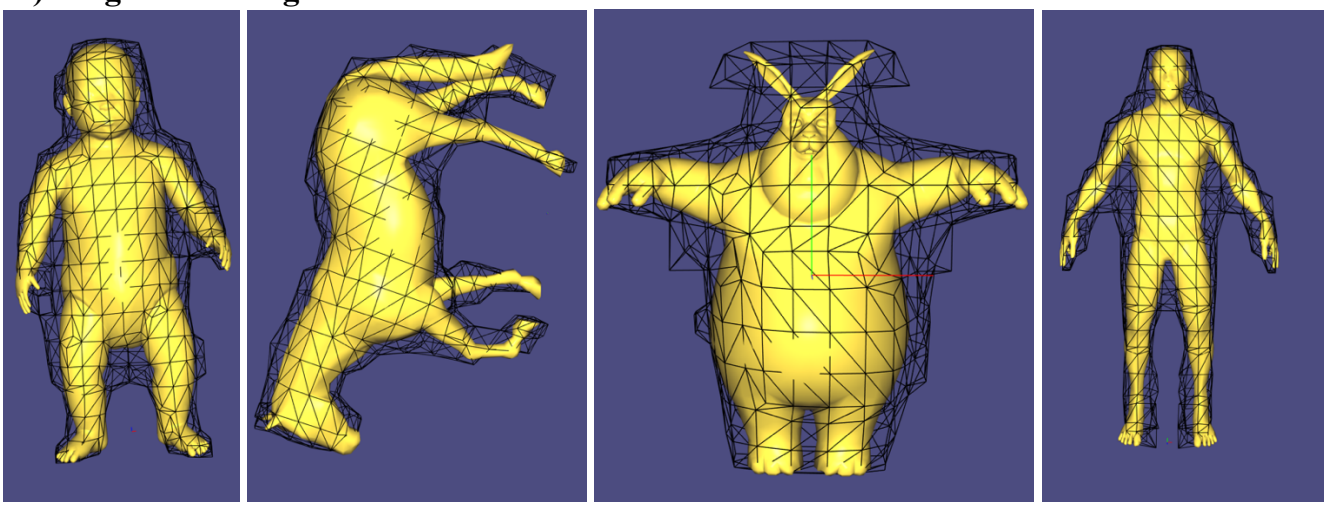


*Figure 6 : Meshes I obtained after the smoothing step*

**Mesh Laplacian smoothing**
The last step of the algorithm consists in smoothing the so generated cage.
I use the Laplacian matrix (cotan form) and the implicit smoothing algorithm, with 2 iterations, and a lambda value equal to 0.5 (c.f. previous coursework) :

$$(I - \lambda\Delta)V_{i+1} = V_i \quad \textbf{(4)}$$

**Cage-mesh penetrations**

To avoid cage-mesh penetrations as much as possible, for each vertex $v_i$, I test whether the segment $[v_i, v_{i+1}]$ intersects the mesh : if it does, I only move $v_i$ halfway towards the intersection point.
The pseudo code for that step becomes :

Smoothing step i :

$(I - \lambda\Delta)V_{i+1} = V_i$

For j in (0,N)

(intersect,t) = Intersect($v_i^j, v_{i+1}^j$, mesh)

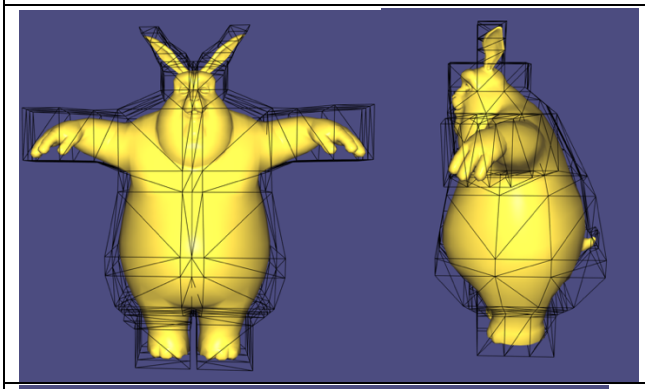
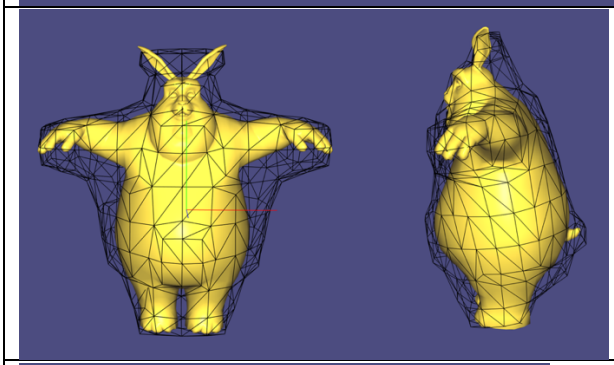
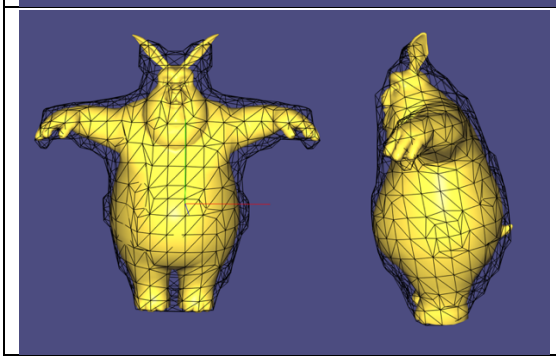
If (intersect & t < $\|v_{i+1}^j - v_i^j\|$)

$$v_{i+1}^j = v_i^j + 0.5 * t * \frac{v_{i+1}^j - v_i^j}{\|v_{i+1}^j - v_i^j\|}$$

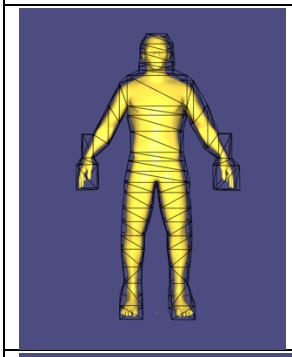
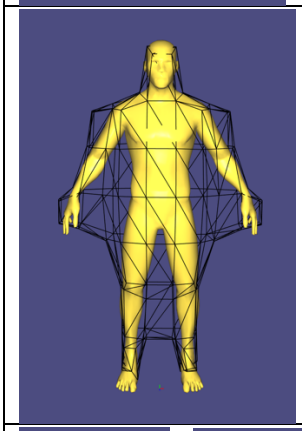
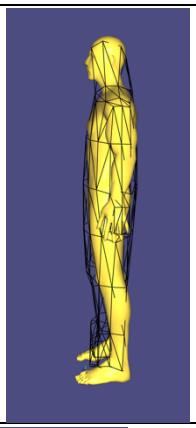
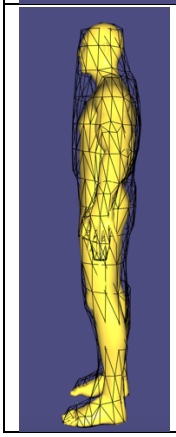
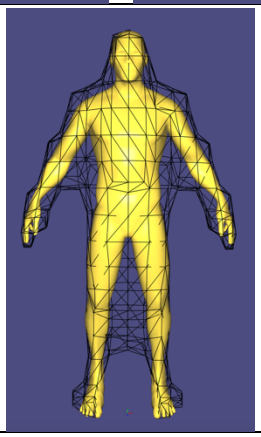
## e) Results and evaluation

**Comparing with hand made cages**

In this section I show examples of the meshes I generated with this algorithm, for different sparseness values $s_F$, and compare them with the cages modelled by hand by Nicolas. I then discuss the limitations of the method and my implementation.

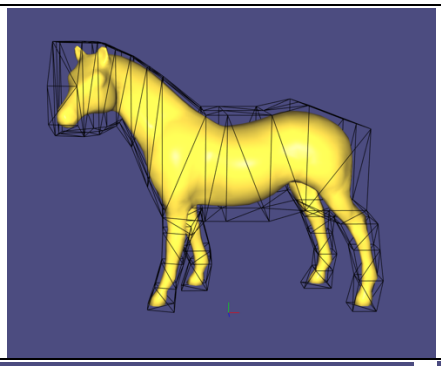
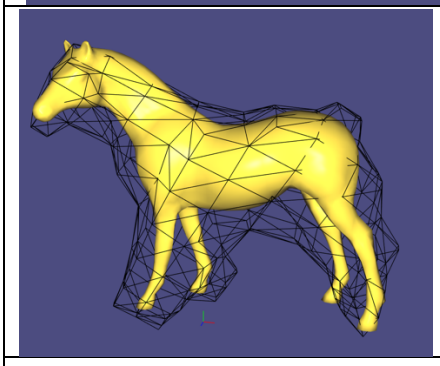
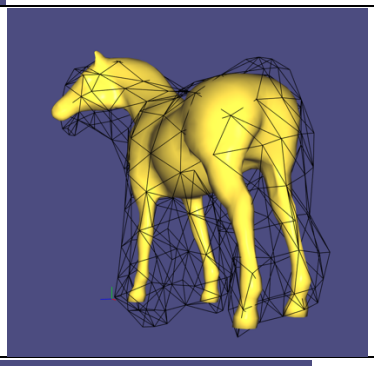
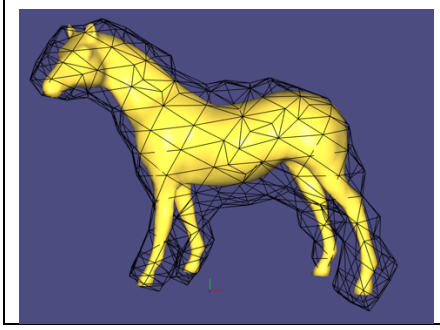
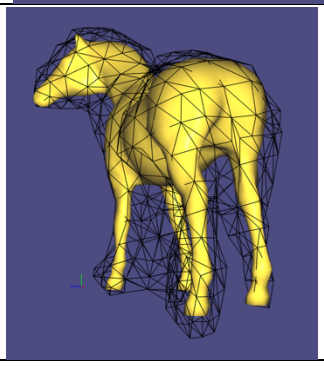
| BUNNY | Modelled by hand VS Algorithm | Number of cage triangles |
|---|---|---|
|  | Modelled | 582 |
|  | Algorithm ($s_F = 0.2$) | 590 |
|  | Algorithm ($s_F = 0.5$) | 1176 |

These examples show how manual cage creation provides with better results for the same amount of triangles.
One of the main limitations with the automatic cage generator algorithm is that it is difficult to wrap body parts that are tight or close to one another, typically for legs (c.f. the bunny mesh).

| HUMAN | Modelled by hand VS Algorithm | Number of cage triangles |
|---|---|---|
|  | Modelled | 334 |
|   | Algorithm ($s_F = 0.15$) | 272 |
|   | Algorithm ($s_F = 0.65$) | 852 |

The horse case shows that $s_F = 0.5$ starts to separate the two back legs, but not the front legs.
The same problem is found with the Human mesh : the space between the legs as well as between the arms and the body is hard to grasp for low values of $s_F$.

Also, even though I added a test to make sure the cage points do not penetrate the mesh, nothing in the implementation prevents the cage triangles to intersects those of the mesh (c.f. bunny ears on the 3rd row or the Human feet in the 2nd row).
It is indeed delicate to smooth the mesh correctly without shrinking it too much : i.e. the “peaky” parts moving towards the inside too much, and the cage sticking to the mesh.

| HORSE | Modelled by hand VS Algorithm | Number of cage triangles |
|---|---|---|
|  | Modelled | 284 |
|   | Algorithm ($s_F = 0.25$) | 396 |
|   | Algorithm ($s_F = 0.5$) | 672 |

## Computation time[3]

Here are some computation times for the cage automatic generator, for different meshes and different sparseness factors $s_F$.

| Number of vertices Mesh | Sparseness factor [1] | TIME TO COMPUTE CAGE (s) |
|---|---|---|
| 2119 | 0.5 | 33 |
| 3670 | 0.3 | 36 |
| 3670 | 0.6 | 93 |
| 1525 | 0.2 | 4 |
| 1525 | 0.35 | 10 |
| 1525 | 0.55 | 14 |
| 1525 | 0.75 | 24 |
| 1525 | 0.85 | 25 |
| 1525 | 0.95 | 28 |
| 3670 | 0.15 | 19 |
| 3670 | 0.65 | 113 |
| 4796 | 0.25 | 26 |
| 4796 | 0.5 | 64 |
| 4796 | 0.7 | 94 |

*Table 1 : Computation times of the automatic cage generator algorithm*

[3] The tests were run on a 2 GHz Intel Core i5

All in all, my experimentations lead to determine $s_F = 0.5$ to be a good sparseness value.
The code takes around 20 to 40 seconds to run depending on the size of the mesh.
I did not find any direct relationship between the computation time of the bounding cage and the value of $s_F * N$.

# 2) Mean value coordinates

## a) Overview of the method

We implemented Ju et al.'s Mean Value Coordinates [2][4]. The main achievement of this method is a geometric formula to interpolate a set of values defined at vertices of a mesh. The designed weights satisfy the properties of interpolation, smoothness and linear precision.[5]

Their work starts by defining the formula of the interpolation function for the general case of continuous surfaces, which mathematical details we will not explore in this work.
Here we will focus on the specific algorithm for the case of closed triangular meshes which results from it.
The geometric configuration for this is represented in figure 8. Considering a vertex $v$ (whose function value needs to be computed) and a triangle $\{p_1, p_2, p_3\}$ (from the surface the function is defined on), projecting the triangle onto a unit sphere centered on $v$ allows to define the following values :

- $\theta_i$ : length of the circular arc corresponding to the $i^{th}$ edge
- $n_i$ : inward unit normal to the $i^{th}$ edge
- $m = \sum_{i=1}^{3} \frac{1}{2}\theta_i n_i$ ***(5)***, the so-called "mean-value", which is proved to be the integral of the outward unit normals over the spherical triangle T.

Therefore the weight $w_i$ for the $i^{th}$ vertex computes as :

$$w_i = \frac{m.n_i}{n_i.(p_i - v)} \qquad \textbf{(6)}$$

Note that $w_i$ is the contribution of vertex $p_i$ within the considered triangle, and that the actual weight from $v$ to $p_i$ is the sum of the individual weights defined in the triangles where $p_i$ appears.
Additionally, this weight is ultimately normalized by dividing by the sum of $w_i$s over all the vertices of all the triangles.

We used mathematic simplifications of this formula (provided by the paper's pseudocode) in order to implement the algorithm.
We also took care of the co-planar cases mentioned in the paper (i.e. cases where the point $v$ lies within a certain triangle or on the same plane but outside of it).

---

[4] The code for this part, mostly written by Nicolas, is in the class MeanValueCoordinateController

[5] Interpolation : If a point v converges to a vertex of the mesh $p$, $\hat{f}[v]$ converges to $f_p$, i.e. the value of the auxiliary function on $p$.
Smooth : the interpolation function $\hat{f}$ is $C_1$.
Linear precision : If the auxiliary function $f_p$ is the position of the vertex itself, then the interpolated value of a vertex $v$ is $v$ itself.

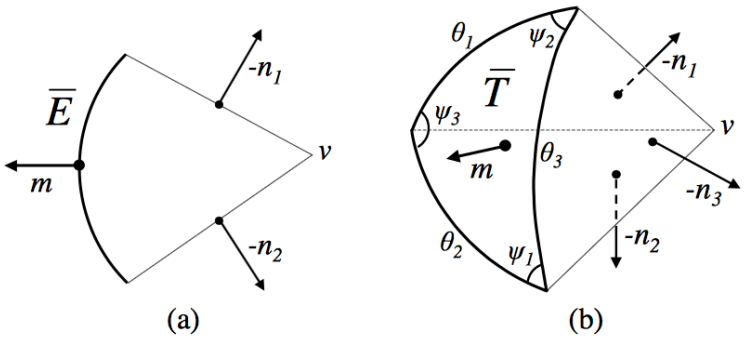


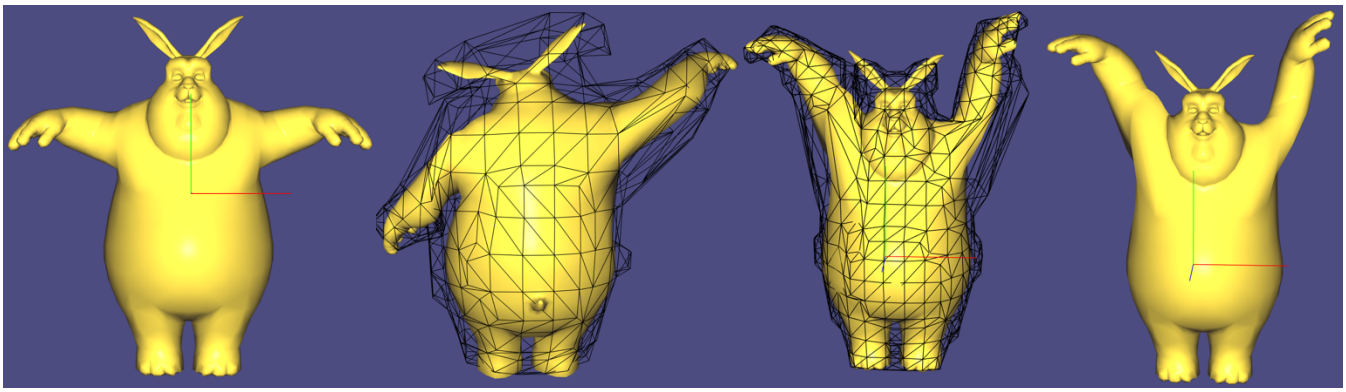

*Figure 7 : Geometric construction for the Mean value coordinates [2] : 2D (a) and 3D (b) diagrams of the projected spherical triangle*

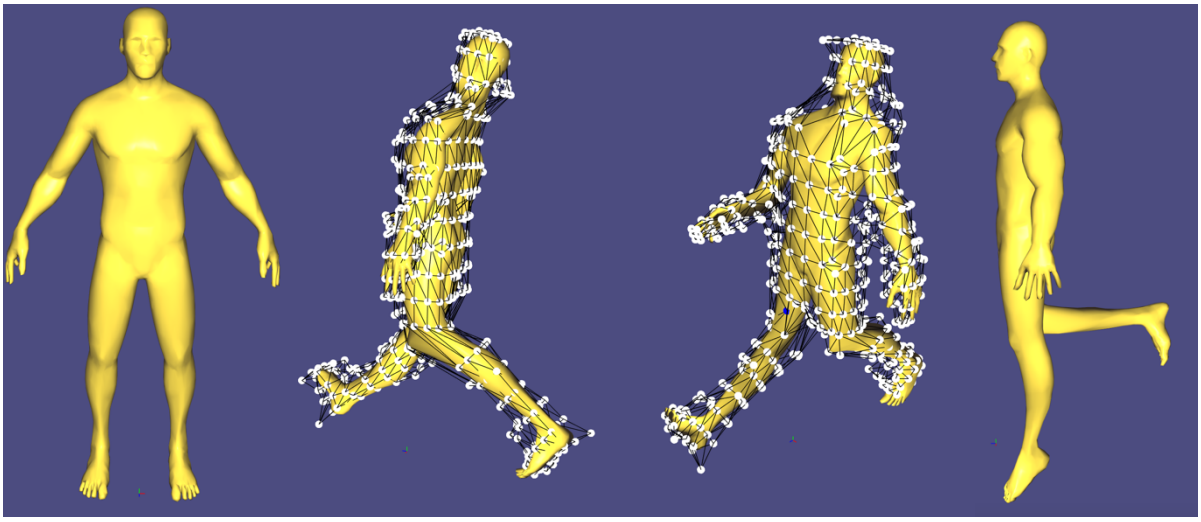
Therefore, for the specific case of surface deformation, the auxiliary function $f$ is the position of the cage vertex itself, and the overall algorithm consists :
1- pre-computing the mean value coordinates (i.e. interpolation weights) between the initial dense mesh and the initial (i.e. undeformed) cage.
2- deforming the cage by moving its points
3- computing the deformed mesh's coordinates by interpolating the deformed cage using the initial weights.

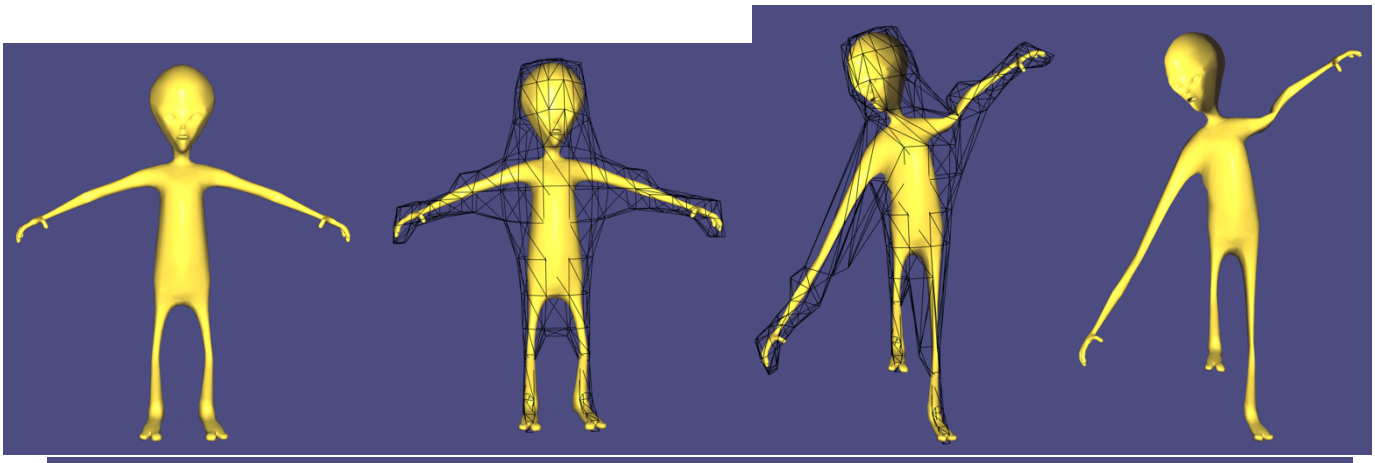
The main claimed advantages of the Mean Value Coordinates [2] are :
1) it produces smooth and realistic deformations
2) it is fast : computing the weights takes the most time, but it is only done once. On the opposite, interpolating is very straightforward, which makes the method real-time friendly (see table 2 for time performance values).
3) it is performant even with a very small number of control points

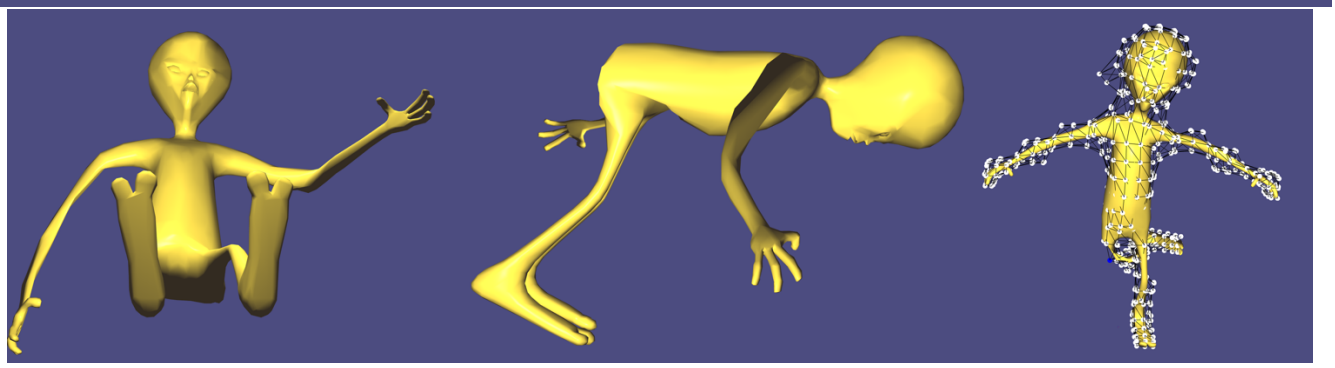
**b) Results and evaluation of the method**

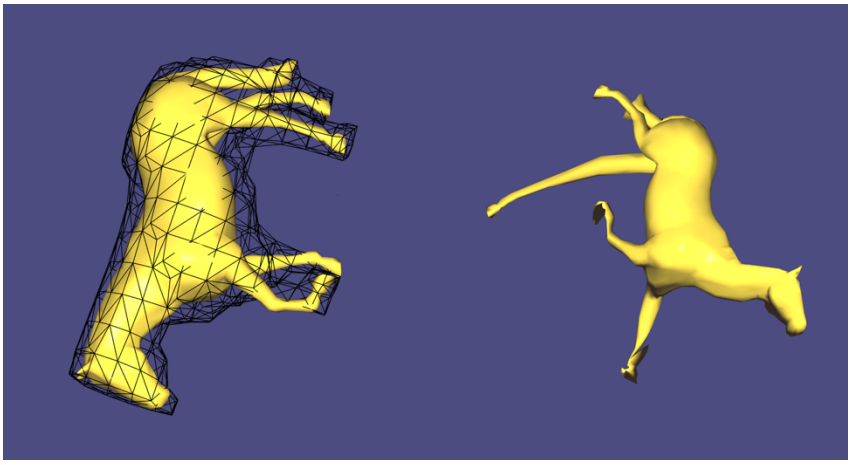
Here I present the results of the deformations achieved with the Mean value coordinates (using both the hand-modelled cages and the automatically generated ones, and using the UI described in section 3.). The first image of each row shows the undeformed mesh.











We found that the deformations created with this method were very smooth and stable (no degenerate cases).
It is also very fast to run : for typical cages of up to 600 triangles with meshes of up to 4000 vertices and meshes we could run the deformations in real time.
Note : look at the video (link in the intro) for a demonstration of the real-time deformations.

It seems important to stress that one particularity of the Mean Value Coordinate algorithm is that it computes weights that are relative to the entire cage : i.e. each mesh vertex has as many weights as the number of vertices in the cage.
The advantage of that is that the method doesn't require volumetric cells or extra computation to recover the cage points to interpolate on. Additionally, this participates in making the deformations smoother.
The drawback is that with some cages, it becomes delicate to isolate a specific part of the cage to generate local deformations.

### Computation time

Here are a few figures on computation time (in seconds) to run the mean value coordinate calculation algorithm :

| Number of mesh vertices | Number of cage triangles | Time to compute Mean Value Coordinates |
|---|---|---|
| 2119 | 480 | 32 |
| 3670 | 552 | 74 |
| 3670 | 852 | 112 |
| 1525 | 270 | 15 |
| 1525 | 336 | 18 |
| 1525 | 452 | 24 |
| 1525 | 568 | 28 |
| 1525 | 632 | 32 |
| 1525 | 692 | 36 |
| 1525 | 944 | 50 |
| 4080 | 588 | 105 |
| 4796 | 2400 | 30 |
| 8150 | 1200 | 213 |
| 7336 | 244 | 56 |

*Table 2 : Computation times of mean value coordinate weight calculation*

Note that the interpolation times vary from 0.02 to 0.1s depending on the number of vertices, which makes it real-time.

# 3) User interface and cage manipulation

In this part, I describe the different ways the user can interact with the mesh in the UI, and how I implemented them. [6]

### Cage stretching

The user has the possibility to stretch (and compress) the cage in the three dimensions by moving a slider, and the mesh will deform accordingly by interpolating.
The cage deformation for that matter is set by the following equation :
For stretching in dimension j (x,y,z) :

$$\tilde{V}_j^{stretched} = v_j^{barycenter} + s^j \, (V_j^0 - v_j^{barycenter}) \quad (7)$$

where $s^j$ is the stretching coefficient, and $v_j^{barycenter}$ is the cage barycenter's jth coefficient.

[6] The code for this part is in the class DeformCageViewerPlugin (header file only)

This very simple deformation type is visually not very exciting, but served as an efficient way to check the validity of the code and the cage models.

**Joint-based grouped point handling and dragging**

In order to design fancier and more realistic ways to deform the cage, we drew inspiration from the "joint" concept of skeleton-based deformations.
Indeed, we thought of moving the cage points by rotating them around "joint" points.
In order to avoid moving each single point at once, I designed a way to gather all the points of interest (typically an entire limb) through a single click.
Here is the system I designed :

1- I select the joint point of choice by roughly clicking on it while **pressing key 'J'**. The corresponding point on the mesh turns blue (c.f. figure 8 top). We will name this first click position $C_J$[7].
Note that the « click points » $C_J$ (and $C_S$ in the next paragraph) are the 3D world coordinates of the intersection point between the ray casted by the mouse click and the mesh. I therefore implemented a mouse-to-world ray function, as well as a ray-mesh intersection test.
The same procedure is used to unselect a joint point (turns white again).

2- **Pressing 'S' and clicking** at a different position on the mesh (named $C_S$) will select the group of points which will be moved by the user (they turn green, c.f. figure 8 bottom).
My method to determine this set of points out of a user click is the following :
I recursively explore the vertex neighbors, starting from the seed (i.e. the closest vertex to $C_S$) and add them until the distance (from the point to add) to $C_S$ becomes greater than $d_{max} = \|C_S - C_J\|$.
This means that after selecting a joint (e.g. on the armpit of a body), one has to click approximately halfway between the joint and the end of the limb in order to select the whole limb (c.f.).
Attention has to be paid to not 'overshoot' through a too distant click (or a wrong joint selection) that will add more points than necessary and trigger a deformation that is different than wanted.

3- Once the group of points has been selected, and as long as 'S' is pressed with the mouse down, by **dragging** the mouse, each selected point (green point) is rotated around the joint point $C_J$ in the plane defined by the point and the camera normal : i.e. in the plane parallel to the view plane and which goes through the point itself.
Indeed, when the mouse moves, the vectors $(C_J C'_s)$ and $(C_J C_s)$ (i.e. joint→ new mouse and joint→initial mouse) allow to define the angle of rotation, as well as a factor of scaling.
The exact calculation of the updated cage positions is in the callback function `bool mouse_move(int button, int modifier)` in the class `DeformCageViewerPlugin : public ViewerPlugin.`

Note : The group selection can be tricky on some parts of the cage. It is necessary to understand how it works in order to deform the cage according to one's will.
Also, look at the videos submitted, with a demo of how to handle the cage.

[7] The « click points » $C_J$ and $C_S$ are 3D world coordinates of the intersection point between the ray casted by the mouse click and the mesh. I therefore implemented a mouse-to-world ray function, as well as a ray-mesh intersection test.

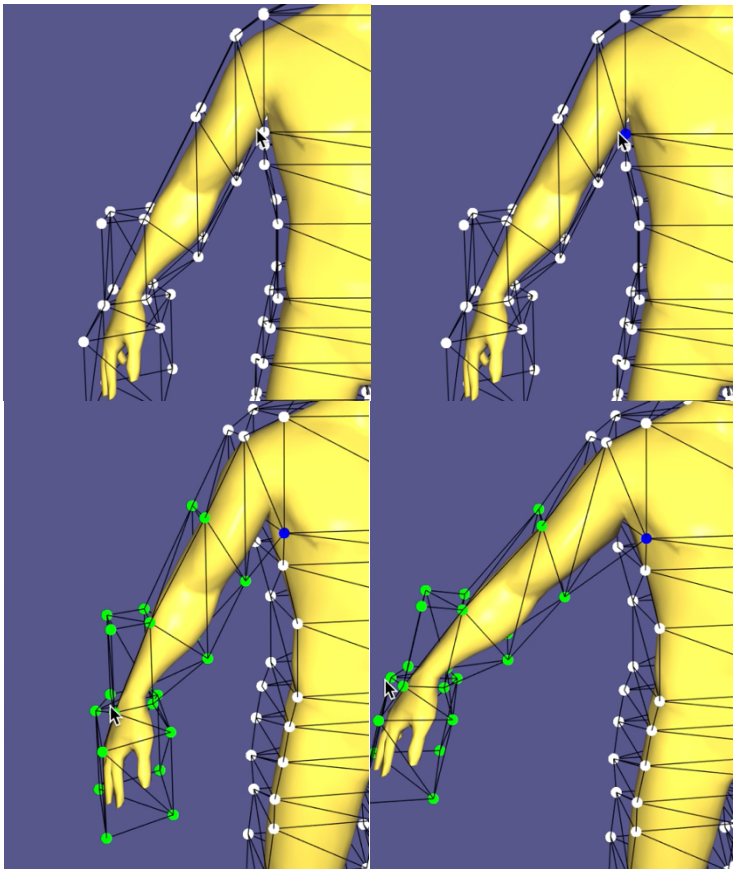


*Figure 8 : Top : Select the joint point by pressing "J" and clicking on the mesh : the joint point (blue point) is determined as the closest vertex to the click position*
*Bottom left : Once the joint point has been selected, select the moving cage points by pressing "S" and clicking again on the mesh : the set of moving points is determined by expanding from the nearest vertex to the second click, adding points until the distance to the seed is greater than the distance between the two clicked points.*
*Bottom right : move the mouse to rotate the set of selected points around the joint point, in the plane perpendicular to the view.*

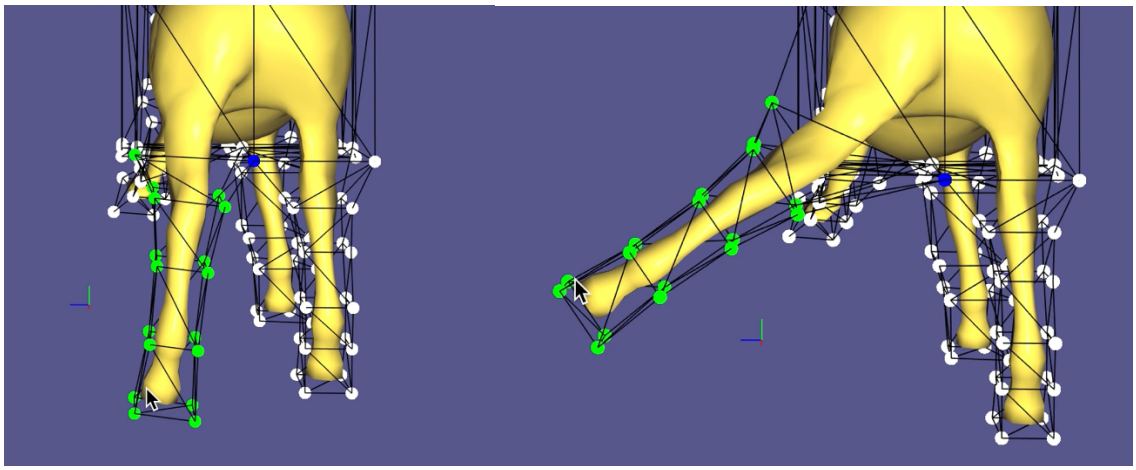


*Figure 9 : Another example of the grouped point handling and joint-based rotation*
*Left : the initial mouse click for the group selection (the joint has already been selected : see blue point) : the whole leg turns green.*
*Right : moving the mouse to rotate the selected points, adding a bit of stretch by moving the mouse a bit further away from the joint*

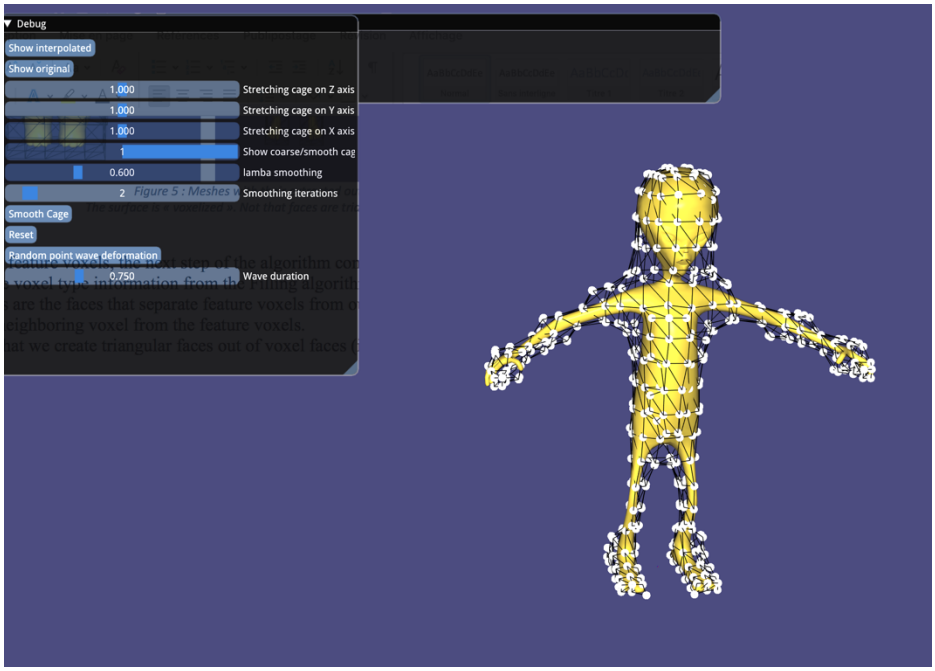



*Figure 10 : The User Interface with the several buttons and sliders*
*Note : the different key and mouse options are not represented*

# CONCLUSION

Mean value coordinates prove to be smooth, realistic and efficient algorithm to interpolate the coordinates of a dense mesh relative to a coarse bounding cage.
Moving the bounding cage then allows to animate and deform the mesh, in real-time.

The main difficult consists in generating sparse and accurate cages that account for the different components of the mesh to deform.
The system I designed to manipulate the cage can lead to quite convincing results, but can be tricky to use for complex cages.
Refining both the hand-modelled and the automatic cage generator implementation (namely by deleting some vertices) would surely lead to more straightforward manipulations.

# APPENDIX

## Appendix 1 : Deforming the cage : user instructions

- Move the sliders with the “Stretching cage on … axis” mention in order to stretch the cage in one of the three dimensions.
- Deform the cage by clicking and dragging points (c.f. figure 8):
  - Select a joint point by pressing ‘J’ and clicking anywhere on the cage : the closest vertex to the click will be set as the joint point. The point will appear
  - Select a bunch of points to move by clicking and dragging the mouse : click at approximately ¾ between the joint and the end of the limb you want to move in order to cover the whole limb. Make sure that the points you wanted to select are indeed turning all green, and that no additional point is turning green.
  Dragging the mouse (while still pressing “S”) will rotate and stretch those selected points in the plane parallel to the image plane.
  - Click anywhere on the mesh while pressing ‘J’ to unselect the joint point, and restart the procedure.

## Appendix 2 : Code structure and manipulation

Important : in order to run the code, the libigl library ("libigl" folder) needs to be at one level higher to the source code.

**Classes**

- The menu initialization, the objects instantiation as well as most of the code responsible for the UI is to be found in main.cpp.
- MeshProcessor.hpp - MeshProcessor.cpp : basic calculation of mesh barycenter, extrema points…
- MeanValueCoordinateController.hpp - MeanValueCoordinateController.cpp : the calculation of the mean value coordinates and the online interpolation.
- CageGenerator.hpp - CageGenerator.cpp : the algorithm for the automatic cage generator.
- DeformCageViewerPlugin.hpp : the main user-cage manipulation and the deformation system. It inherits from the LibIGL class `ViewerPlugin` to facilitate the callback uses.

**Changing the parameters and variables**

In main.cpp :

- Change the string values of `mesh_file_name` and `cage_file_name` with the correct address of the meshes to load accordingly to the hierarchy you have.
- Boolean `compute_automatic_cage` (global) : run the automatic cage generation algorithm, or load an already existing cage.
  `sparseness_cage` (global) : change the value of the sparseness parameter from [1] : increase to generate denser and more refined cages.

# REFERENCES

**[1]** XIAN, C., LIN, H., AND GAO, S.: Automatic generation of coarse bounding cages from dense meshes. In *Proc. SMA* (2009)

**[2]** JU T., SCHAEFER S., WARREN J.: Mean value coordinates for closed triangular meshes. In *ACM TOG 24*, 3 (2005)