Interactive free form deformer for point-based objects by GPU acceleration

A. Beddiaf[1] and M.C. Babahenini[2]
LESIA Laboratory, University Med Khider of Biskra
Biskra, ALGERIA
1 *alibeddiaf@gmail.com,* **2** *chaouki.babahenini@gmail.com*

W. Puech
LIRMM Laboratory,University of Montpellier II
Montpellier, FRANCE
william.puech@lirmm.fr

***Abstract*—the point-based representation constitutes a recent useful alternative approach for modeling enormous and large 3D polygonal models which contain usually millions of faces. However, and in order to allow the artists and the designers to work with such representation, a new mechanisms of FFD (Free Form Deformation) must be offered and that especially with an interactive response time or real time response (in the best case). This paper presents an acceleration of FFD for point-based objects and particularly splat-based with interactive response time, and that by exploiting the programmable GPU features of the modern graphics cards.**

***Keywords*— Interactive, free form deformation, point-based objects, GPU, shader.**

I. INTRODUCTION

With the constant increasing of scenes complexity, and technological progress, it appears that the using of polygonal meshes for rendering has become less and less attractive compared to the direct manipulation of points. In addition, the hidden faces culling become much expensive when the scene is complex. A new paradigm for 3D computer graphics, called point-based rendering emerged in the computer graphics field. The principle of this paradigm is to model 3D objects in the scene like a point cloud without an explicit connectivity, the final image will be generated by a simple projection of this cloud, combined with techniques to fill the holes between the points projected to give an impression of continuous surface. Zwicker et al. [1] have presented the surface splatting which appears the best approach to render the point cloud by replacing each point by a circular or elliptical colored disk called splat. Nevertheless, the lack of the connectivity information yields difficulties to apply to this family of objects, some conventional techniques such as geometric deformation tools. Pauly et al. [2] introduced a free deformation tool point-based that allows the user to interactively deform the surface by specifying a deformation smooth field. The user firstly defines a region on the surface of the deformable model, and marks portions of this region as a control handle. The surface might be deformed by push, pull or twist the handle. These interactive operations from the user will be translated to a continuous tensor field. In effect, the interactivity is the main obstacle for the using of deformation tools, in addition, it can't be maintained when the complexity of the 3D model exceeds a certain threshold related to the power of the machine. Boubekeur et al. [3] Proposed to solve this scalability problem by introducing a streaming system based on a sampling / reconstruction approach. Firstly, a fast out-of-core adaptive simplification algorithm is performed in a pre-processing step for the construction of a simplified version of the model. The resulting model can then be submitted to FFD tools, which allows an interactive response because its size is reduced. Secondly, a post-processing step performs a reconstruction of preservative characteristics deformation by applying the deformation to the original model in its simplified version. Accelerating the free form deformation by the GPU is quite new, and very recent works have been proposed [6] on this topic.

II. RELATED WORK

A. Deformer

A deformer is an operation that takes as input a collection of vertices and generates new coordinates for those vertices (the deformed coordinates). In addition, new normals and tangents can also be generated. This task is perfectly suited for vertex programs, where some requirements must be fulfilled (like no creation or deletion of vertices or edges, no influencing of a given deformed vertex on another, and the determinism of the deformation). Most deformers have control parameters: user-defined parameters that give the deformer some variation.

In the most general sense, we think of a deformer as a vector function: $f(x, y, z)$.

Or more precisely it's three scalar functions:

$$(f_x(x, y, z), f_y(x, y, z), f_z(x, y, z))$$

where f_x, f_y *and* f_z are the component functions of *f*. This function *f* takes the coordinates *(x, y, z)* of a vertex and returns the deformed coordinates of that vertex. To deform a mesh, we take each vertex, input its coordinates into *f*, and store the output as the new vertex coordinates. The deformer function *f* could be defined for all the vertices, here we talk about **global deformation,** or for some subsets, in this case we talk about **local deformation**.

978-1-4577-0908-1/11/$26.00 ©2011 IEEE

Linear transformations (translation, rotation and scaling) are not a particularly interesting set of deformers since we know simply how to compute the deformed normals of the vertices (by multiplying them with the inverse transpose of the transformation matrix). But in general, a given deformer can't be written in matrix form if it is not a linear transformation and new techniques for transforming the normals must be found.

B. Deforming normals

The first trivial solution to compute the deformed normals is to operate by a second pass after the vertices coordinates deformation pass. In this second pass we recompute the normals based on neighboring vertices in the new deformed mesh. Unfortunately we don't have that option with vertex program. Each vertex being processed in a vertex program has no knowledge on its neighbors, and there is no second stage where we can look around and do computations. The second solution is an approximate numerical technique where the deformed normal can be achieved by deforming three points for every input vertex. We find two new points very close to the input vertex. Add a very small multiple of the tangent to the vertex coordinates to get the first point. Add a very small multiple of the binormal to the vertex coordinates to get the second. These three points define two vectors whose cross product is the original normal. We can then deform all three of these points by the deformer, and these three deformed points will define two vectors whose cross product is approximately the deformed normal. The closer together these three points are, the more accurate the approximation will be, however numerical accuracy problems will appear if the points are too close together.

The third solution is based on the Jacobian matrix. If a deformer $f(x,y,z) = (f_x, f_y, f_z)$ has continuous first-order partial derivatives, then we can compute a matrix for each vertex called the Jacobian matrix **J**. as known, everywhere a function *f(x, y, z)* is smooth, *f* can be approximated by a linear transformation **J**, plus a translation $f(x) \approx J(a)(x-a) + f(a)$, for some point a, and $x=(x, y, z)$. This follows from the generalized Taylor's theorem. The Jacobian matrix is defined by the equation (1) (note that the Jacobian matrix is often denoted $\partial f(x)$):

$$J(x,y,z) = \begin{bmatrix} \partial \mathrm{fx}/\partial \mathrm{x} & \partial \mathrm{fx}/\partial \mathrm{y} & \partial \mathrm{fx}/\partial \mathrm{z} \\ \partial \mathrm{fy}/\partial \mathrm{x} & \partial \mathrm{fy}/\partial \mathrm{y} & \partial \mathrm{fy}/\partial \mathrm{z} \\ \partial \mathrm{fz}/\partial \mathrm{x} & \partial \mathrm{fz}/\partial \mathrm{y} & \partial \mathrm{fz}/\partial \mathrm{z} \end{bmatrix} \quad (1)$$

Where all partial derivatives are evaluated at *(x, y, z)*.

We can transform the normals using the Jacobian matrix in two equivalent ways:

1- The inverse transpose of J(x,y,z) can be computed per vertex in the vertex program and used to transform the normal directly. If **N** is the input unit normal, then the deformed normal is **N'**:
$$\mathbf{N'} = (J(x,y,z)^{-1})^{T} * \mathbf{N} \quad (2)$$
2- If a unit tangent vector **U** is given per vertex then we can generate a unit binormal **V**= **N^U**. We can transform the tangent and binormal to the deformed space, using the matrix **J**(x,y,z). Then the new deformed normal is:
$$\mathbf{N'} = (\mathbf{J}(x,y,z)*\mathbf{U}) \wedge (\mathbf{J}(x,y,z)*\mathbf{V}) \quad (3)$$

Inverting 3x3 matrices is not a particularly suitable task for a vertex program, so the second method is generally easier to implement and faster to compute.

C. Graphics pipeline

In 3D graphics rendering, graphics pipeline or rendering pipeline refers to the stages required to transform a 3D primitives from the object space into a 2D image on the screen space. The stages are responsible for processing information initially provided as primitives properties (coordinates, color, material, etc) used to describe what is to be rendered, and the typical primitives in 3D graphics are vertices, lines and faces.

The stages of the OpenGL (as an open standard cross-language and cross-platform API for 3D rendering) rendering pipeline are illustrated in Fig. 1.

Initially the old OpenGL rendering pipeline has been fixed functionalities (meaning that the coder doesn't have a great control on the way by which the input data are processed), but with the introduction of the modern GPU (Graphics Processing Unit), several stages become programmable through the shader model, using a different languages such as GLSL (openGL Shading Language), HLSL (High Level Shading Language), Cg (C for Graphics). This allows not only the high level control of the rendering effects, but the most important thing is the offloading the vertex and the fragment dependent computations from the CPU to the GPU, in order to have an efficient real-time rendering.

III. Formulation of the free form deformer

The basic idea behind the free form deformation is very simple to understand. It involves wrapping the object in some space then distort this last. The deformations made on the space are then applied to the wrapped object.

Inspired by non-linear deformations of Barr [4], Sederberg and Parry [5] proposed a simple and user-friendly deformation model called: the free form deformation. The process is summarized in three main steps:

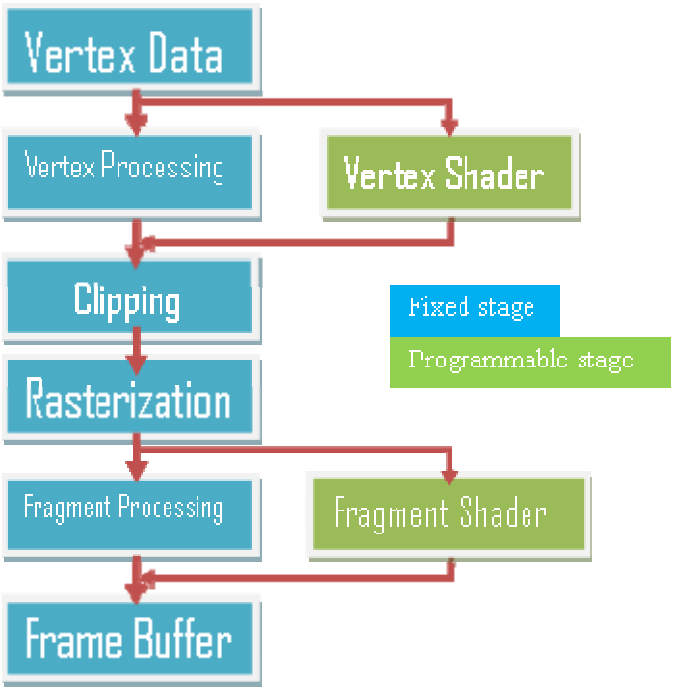

Fig. 1. Simplified OpenGL rendering pipeline with the introducing of the shader model.

1. Create a volume as a parallelepiped around the object and impose local coordinates at each vertex of the object to deform.
2. Fix a grid of control points on the parallelepiped.
3. Deform the object by moving the control points of this grid.

Once the parallelepiped wrapped the object is computed, where its origin is X_0 and the three sides are S, T and U, we will compute the local coordinates *(s, t, u)* of the object's vertices and that according to the equation (4):

$$X = X_0 + sS + tT + uU \quad (4)$$

Then we can fix the grid of control points that contains *(l+1)* points at the axis S, *(m+1)* points at the axis T and *(n+1)* points at the axis U, so *(l+1)*(m+1)*(n+1)* points in total. The coordinates of each control point are computed by the equation (5):

$$P_{ijk} = X_0 + \frac{j}{l}S + \frac{i}{m}T + \frac{k}{n}U \quad (5)$$

Now the deformation could be introduced and that by modifying the coordinates of the grid's control points, then the vertices will be deformed according to the deformation made on the grid, and that using the Bernstein polynomials as illustrated in the equation (6, 7, 8, 9) :

$$X' = f(X) = (f_x, f_y, f_z) = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}B_i^l(s)B_j^m(t)B_k^n(u) \quad (6)$$

$$f_x = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^x B_i^l(s)B_j^m(t)B_k^n(u) \quad (7)$$

$$f_y = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^y B_i^l(s)B_j^m(t)B_k^n(u) \quad (8)$$

$$f_z = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^z B_i^l(s)B_j^m(t)B_k^n(u) \quad (9)$$

Where $B_i^l(s), B_j^m(t)\ et B_k^n(u)$ are the Bernstein polynomials defined by the equation (10) :

$$B_i^l(s) = \binom{l}{i}(1-s)^{l-i}s^i \ , \ B_j^m(t) = \binom{m}{j}(1-t)^{m-j}t^j \ , \ B_k^n(u) = \binom{n}{k}(1-u)^{n-k}u^k \quad (10)$$

Until now, we have only computed the deformed coordinates of the vertices. And in order to compute the deformed normal associated at each vertex we have to compute the Jacobian matrix at each vertex, and this will be done by computing the first-order partial derivatives as the following equations:

$$\frac{\partial f_x}{\partial x} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^x BP_i^l(s)B_j^m(t)B_k^n(u) \quad (11)$$

$$\frac{\partial f_x}{\partial y} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^x B_i^l(s)BP_j^m(t)B_k^n(u) \quad (12)$$

$$\frac{\partial f_x}{\partial z} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^x B_i^l(s)B_j^m(t)BP_k^n(u) \quad (13)$$

$$\frac{\partial f_y}{\partial x} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^y BP_i^l(s)B_j^m(t)B_k^n(u) \quad (14)$$

$$\frac{\partial f_y}{\partial y} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^y B_i^l(s)BP_j^m(t)B_k^n(u) \quad (15)$$

$$\frac{\partial f_y}{\partial z} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^y B_i^l(s)B_j^m(t)BP_k^n(u) \quad (16)$$

$$\frac{\partial f_z}{\partial x} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^z BP_i^l(s)B_j^m(t)B_k^n(u) \quad (17)$$

$$\frac{\partial f_z}{\partial y} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^z B_i^l(s)BP_j^m(t)B_k^n(u) \quad (18)$$

$$\frac{\partial f_z}{\partial z} = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} P_{ijk}^z B_i^l(s)B_j^m(t)BP_k^n(u) \quad (19)$$

Where $BP_i^l(s), BP_j^m(t)\ etBP_k^n(u)$ are the first derivatives of the Bernstein polynomials computed as :

$$\frac{\partial B_i^l(s)}{\partial x} = BP_i^l(s) = \binom{l}{i}((l-i)(-\frac{1}{s})(1-s)^{l-i-1}s^i + i\left(\frac{1}{s}\right)(1-s)^{l-i}s^{i-1}) \quad (20)$$

$$\frac{\partial B_j^m(t)}{\partial y} = BP_j^m(t) = \binom{m}{j}((m-j)(-\frac{1}{T})(1-t)^{m-j-1}t^j + j(\frac{1}{T})(1-t)^{m-j}t^{j-1}) \quad (21)$$

$$\frac{\partial B_k^n(u)}{\partial z} = BP_k^n(u) = \binom{n}{k}((n-k)(-\frac{1}{U})(1-u)^{n-k-1}u^k + k(\frac{1}{U})(1-u)^{n-k}u^{k-1}) \quad (22)$$

IV. Experimental Results

We have applied the described deformer on several objects, here we show two chosen objects; the first is a point-based textured sphere that contains 59650 splats, and the second is a

point-based human face that contains 40880 splats. The grid of the deformer contain 10x10x10 cells, meaning 11x11x11 control points or 1331 control points in total.

The deformer has been implemented onto a computer with the following features : RAM : 1 GB, CPU : Intel Pentium Dual-Core 2.70 GHz, graphics card : NVIDIA Geforce 9500 GT with 1 GB of memory space.

Note that the first implementation is CPU-based implementation, i.e. without exploitation the programmable functionalities of the GPU (only the fixed functionalities).

The deformer has been applied with different control parameters in behalf of generating global/local deformations and also translational/rotational motions. For the global deformation, we apply the given motion on the whole control points, meanwhile for the local deformation, we apply the given motion on a subset from the control points. And the two kinds of motions are :

1. Stretching : is done by translating a control points by a 3D displacement vector $V_d(x_d, y_d, z_d)$ as following formula:

$$\forall i \in [n_1\ n_2],\ P'_i = P_i + V_d \quad (23)$$

2. Twist : is done by rotating a control points around an axis (A), by an angle (α) as the following formula :

$$\forall i \in [n_1\ n_2],\ P'_i = Rot(P_i, A, \alpha) \quad (24)$$

The following figures (Fig.2, Fig.3, Fig.4, Fig.5, Fig.6, Fig.7, Fig.8, Fig.9, Fig.10, Fig.11) illustrate the results obtained by applying the deformer on the sphere and the face with different configurations (global/local deformation, translational/rotational motion). However we observe that this first CPU-based implementation has a FPS (Frame Per Second) around 0.02 (response time around 50 seconds), and this is not interactive at all. This FPS is due to the heavy computation of deformed vertices and normals that has overloaded the CPU.

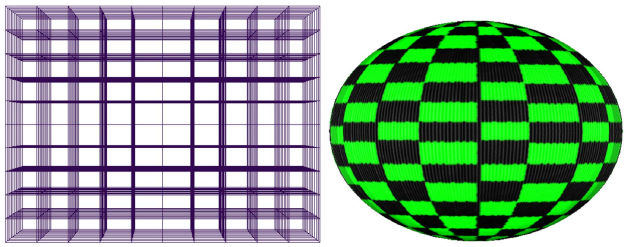

Fig. 2. Normal state of a textured sphere splat-based.

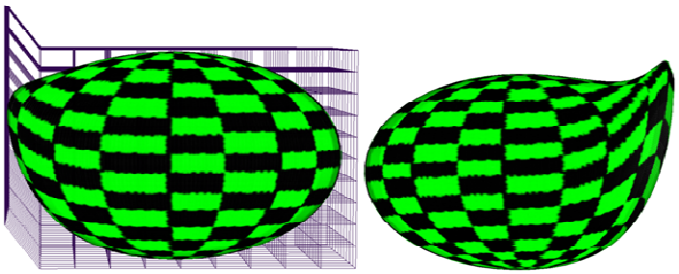

Fig. 3. Local deformation by translational motion (stretching).

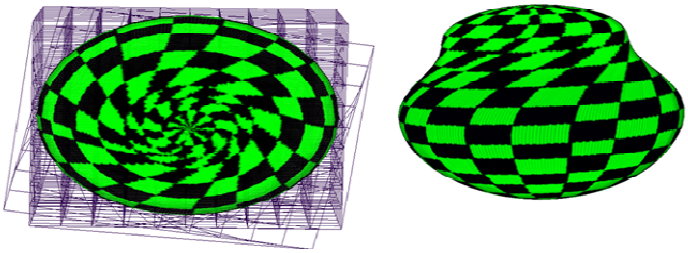

Fig. 4. Local deformation by rotational motion (twist).

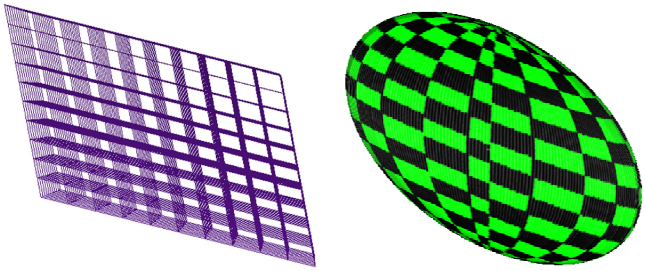

Fig. 5. Global deformation by translational motion (stretching)

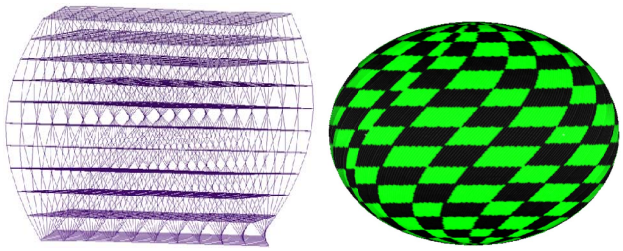

Fig. 6. Global deformation by rotational motion (twist).

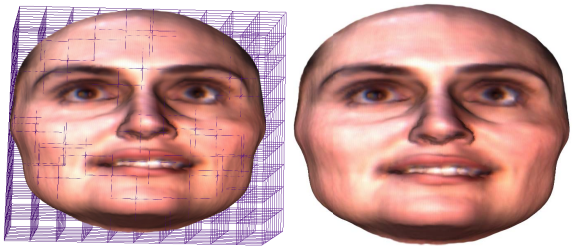

Fig. 7. Normal state of a human face splat-based.

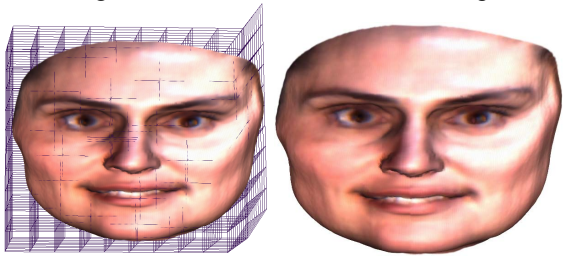

Fig. 8. Local deformation by translational motion (stretching).

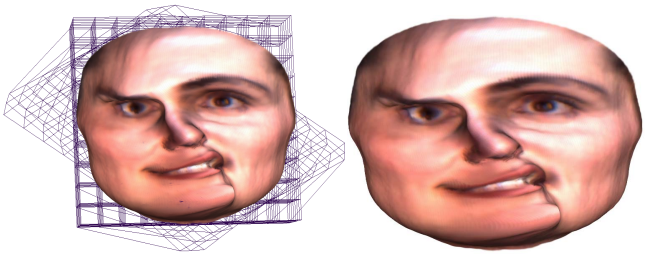

Fig. 9. Local deformation by rotational motion (twist).

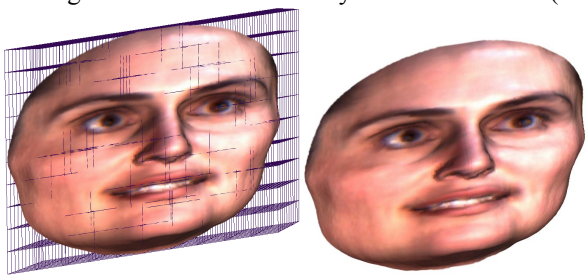

Fig. 10. Global deformation by translational motion (stretching).

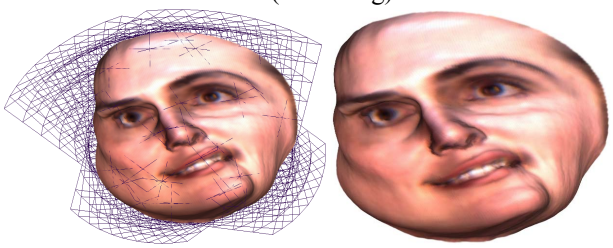

Fig.11. Global deformation by rotational motion (twist)

Our contribution consists of offloading the deformation computations from the CPU onto the GPU of the modern graphics cards, in order to obtain an acceptable interactive response time that is more suitable for the users and the designers.

A. Writing the vertex shader

Once we have written down the deformer (in the previous section), deforming the vertices in vertex shader will be very straightforward. Control parameters are passed as uniform inputs to the vertex shader from the application. In our case, these parameters are the type of the deformation, type of the motion, the displacement vector and the axis/angle of rotation. To compute the new vertex coordinates we simply apply the de previous deformer *f(x,y,z)* on the current vertex coordinates (gl_Vertex) and store the result in the vertex position output (gl_Position) according the GLSL syntax. Thus, the whole deformer will be implemented in the vertex shader. However at the fragment shader level, there is any special processing. Below pseudo-codes of the vertex shader and the fragment shader.

Vertex shader

```
uniform bool deformer;
uniform bool global_local;
uniform bool transl_rotation;
uniform vec3 disp_vec;
uniform float angle_rotation;
uniform vec axis_rotation;
vec4 Vertex;
vec3 Normal;

Void main(){
Vertex=gl_Vertex;
Normal=gl_Normal;

If( deformer )
deformation(Vertex,Normal,global_local,transl_rotation,
disp_vec,angle_rotation,axis_rotation);
gl_FrontColor=gl_Color;
gl_Position=gl_ModelViewProjectionMatrix *Vertex;
}
```

Fragment shader

```
uniform sampler2D color_texture;

void main() {
gl_FragColor = gl_Color*texture2D(color_texture,
gl_TexCoord[0].st);
}
```

The results obtained by this GPU-based implementation give rise to an amazing improvement in matter of FPS up to 5 (meaning with response time around 0.2 second), and this is a suitable interactive response time. Therefore the proposed GPU implementation is almost better 250 times than the CPU one.

V. CONCLUSION

We have presented an acceleration of FFD for point-based objects and particularly splat-based with interactive response time (0.2 second) and a FPS up to 5, which constitutes an improvement about 250 times compared the traditional CPU-based implementation. All that is done by offloading the deformations computation from the CPU to the GPU, at the vertex shader level of the modern programmable graphics cards.

REFERENCES

[1] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In Proceedings of ACM SIGGRAPH 2001, pages 371–378, 2001.

[2] M. Pauly, R. Keiser, L. Kobbelt, M. Gross, "Shape Modeling with Point-sampled Geometry," In *Proceedings of ACM SIGGRAPH 2003 (San Diego, USA, July 27-31, 2003),* pp. 641-650. ACM, July 2003.

[3] T. Boubekeur, O. Sorkine and C. Schlick, "Scalable Freeform Deformation," In *Proceedings of ACM Siggraph 2007 - Sketch Program*, August 2007.

[4] A. H. Barr. Global and local deformations of solid primitives. In H. Christiansen, editor, SIGGRAPH ’84 Conference Proceedings (Minneapolis, MN, July 23-27, 1984), pages 21–31. ACM, July 1984.

[5] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. volume 20, pages 151–160, August 1986.

[6] Marc Modat, Gerard R. Ridgway, Zeike A. Taylor, Manja Lehmann, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Sébastien Ourselin. Fast free-form deformation using graphics processing units. *Comput. Methods Prog. Biomed*. Volume 98, 278-284, June 2010.