High-Performance Graphics (2025)
A. Knoll and C. Peters (Editors)

Real-time rendering of animated meshless representations

Pacôme Luton and Thibault Tricard
INRIA / LJK (CNRS, UGA, INP-G), France

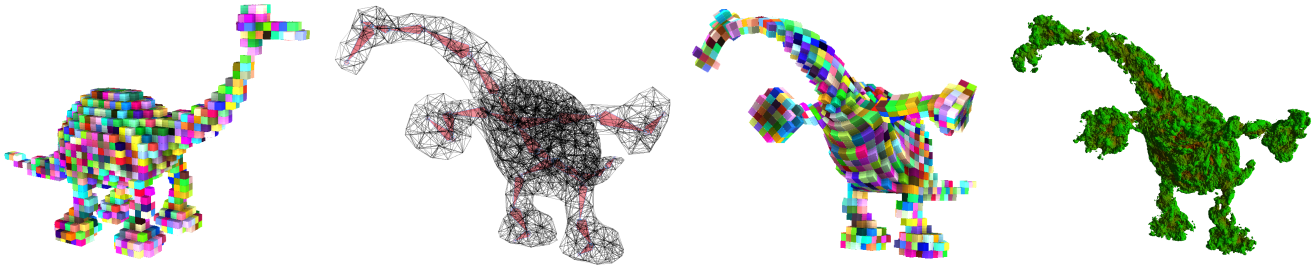

Figure 1: ***Left to right****: Voxel model in rest pose space. Tetrahedral cage animated with linear blend skinning. Voxel model implicitly animated using our method (3.2 ms at 1920×1080). Complex SDF animated by our method (13.8 ms - see supplementary video). For visualization purposes, we use coarse voxels and tetrahedra.*

Abstract

Meshless representations, such as implicit representations (Signed Distance Fields, procedural density fields, etc.) and 3D textures, are important representations in Computer Graphics. Implicit representation allows the representation of geometry with an infinite resolution and a low memory cost. 3D textures are an explicit representation that can store shape information in a regular 3D grid of voxels, allowing for simple anti-aliasing, mipmapping, and dynamic editing. Recent works have improved both representations' rendering performances, making them viable for real-time rendering. However, their animation remains a tedious task, limiting their adoption. In this work, we propose a data structure and a rendering pipeline that allows for animating meshless geometric representations. To achieve that, we encase the meshless representations into a coarse tetrahedral mesh, rigged as we would have for a typical articulated character. At rendering time, we apply the deformation of the rest pose to the full volume using interval shading *[Tri24]. Our method can be directly integrated into a classical rasterization-based rendering pipeline, allowing for the real-time animation of meshless representations using pre-existing animation software.*

CCS Concepts

• ***Computing methodologies*** → ***Animation;*** *Rasterization;*

1. Introduction

Shape representations and design tools are important topics in computer graphics, and many approaches are available. However, for usability in production workflows, they have to be compatible with animation. In practice, surfaces are generally represented as or converted to triangular meshes, which are compatible with most animation methods (e.g., linear blend skinning, free form deformation).

Other shape representations, such as implicit geometry and voxel grids (i.e., 3D texture), have gained popularity for real-time applications due to increased GPU performance and improved rendering algorithms. Despite this popularity, these meshless representations remain extremely difficult to animate, limiting their adoption.

Animating meshless representations presents a significant challenge due to the fundamental nature of their representation. Unlike traditional surface-based models, implicit geometry defines shapes using functions of space (typically, iso-value) that, once deformed, can not be expressed with the same family of functions (e.g., an arbitrarily deformed sphere is not a sphere). Thus, the sampling of the deformed implicit has to be done via a rest pose.

In the case of 3D textures (i.e., volumetric data), there is no trivial way to attach bones or control points, which makes traditional rigging-based animation techniques inapplicable. Deforming a volume requires altering the internal structure of the data field itself,

DOI: 10.2312/hpg.20251172

often in a non-rigid and spatially complex way, making intuitive control extremely challenging and manipulation extremely costly.

A standard solution to animate meshless representations is to use Cage-based deformation [STH*24]. Cage-based deformation is a geometric modeling technique that enables the deformation of an object by embedding it within a deformation cage (a coarse triangle mesh that encloses the object). In this case, the deformation is defined by the offset of the cage's control vertices, and the object inside is deformed according to an interpolation of these displacements. Unfortunately, the expressivity of the deformation depends on the design and complexity of the cage, and this complexity directly impacts the computational cost of the deformation. Furthermore, if the deformation created by the cage is not a bijection (space overlap i.e. self-intersection), it cannot be rendered correctly. Thus limiting their applicability.

To solve these issues, we propose to deform meshless representations using the *juxtaposition of Tetrahedral cages* (close to FFD Lattices [Coq90; CJ91]) to limit the deformation cost while maintaining the expressivity: It is easy to control element size, density, and subdivision level in a tet-mesh, and local linear interpolation is a lot cheaper. Plus, with a high enough tetrahedron count, we can closely approximate a given deformation while keeping the benefit of using linear elements. Moreover, tetrahedra are linear elements; thus, their deformation cannot self-intersecte, however adjacent tetrahedra can intersect without creating issues.

At run time, we render the tetrahedral mesh using *interval shading* [Tri24] using the classical rasterization pipeline to propagate the animation of each tetrahedron to the meshless shape independently (See Figure 2). Doing so allows us to handle each tetrahedron separately, compute all ray-shape intersections in parallel, and rely on the raster operations pipeline to keep only the ones that contribute. This approach mimics the way opaque triangle surfaces are usually rendered and is embarrassingly parallel, thus taking full advantage of the parallel nature of modern GPUs.

Our contributions are:

- A pipeline that allows for the animation of meshless representations in real time, that supports self-intersection, which can directly be integrated into the classical rasterization-based rendering pipeline (See Section 3.1).
- An extension of the *interval shading* method [Tri24] that allows interpolation of vertex attributes, plus a low-level optimization improving its performance (See Section 3.4.2).

Our representation is animation-controlled similarly to classical characters, real-time, and interoperable with classical scene graphs (including intersections and self-intersections) and scene operations (e.g., casting and receiving shadows). Still, it can engulf any kind of meshless representations or details. We demonstrate our results and detail the performance in Section 5 and in the supplementary video.

2. Related Work

In this section, we detail the previous work on animating meshless representations and the methods we build upon.

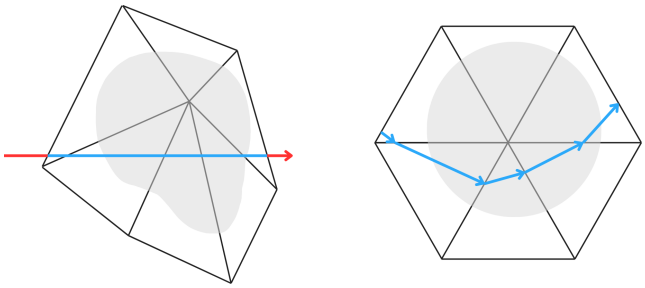

Figure 2: *Visualisation of how the deformation is applied to the path of a single ray.* ***Left****: In world space where the shape is deformed, the ray path is a straight line.* ***Right****: The ray is curved in rest-pose space. We assume it is linear by part, while remaining* $\mathcal{C}^0$ *continuous.*

2.1. Direct deformation of implicit geometry

In some cases, an implicit shape can be a function that uses user-defined handles such as control points (e.g.,skeletons) or scalar parameters (e.g., a sphere defined by a position and a radius). Doing so allows the user to create a complex shape by combining multiple simple ones (Constructive Solid Geometry and its smooth extension [Ini13]). In this case, the animation can be designed by manipulating the user-defined handles, positions, and rotations of those simpler shapes. Examples can be found on platforms like *Shadertoy* [Ini]. However, this kind of animation is difficult to control, and the animation strategy for animated handles is based on fine-tuning from the artist and is different for each implicit shape. An approach to homogenize the animation of SDF is to define them with a handle that is close to traditional rigs, such as skeleton-based implicit [AZ21; CD97]. Unfortunately, as this kind of approach is only compatible with a subset of implicit surfaces, the range of shapes available is also limited. Recently, a method has been proposed to differentiate the parameters of an SDF to find the parameters that produce a specific deformation [RMP*24], allowing an artist to directly control the animation.

These types of methods all have an identical downside: when used for animation, the range of possible deformation is limited by the expressivity of the latent parameter defining the SDF.

2.2. Tesselation and Sampling

Of course, a simple method to animate implicit surfaces is to convert them into a triangle mesh [LC87; JLSW02] and to animate it traditionally. However, using these methods forces the user to choose a sampling resolution; a high resolution will conserve the expressivity of the meshless shape but will create a heavy output mesh , a smaller resolution will result in a loss of detail and might create artifacts when dealing with thin shapes [SJNJ19]. Similarly, 3D texture can be explicitly animated iteratively [Kap02] similarly to what is done for fluid simulation. However, it is very costly and not suited for character animation.

Figure 3: *Summary of the different steps of our rendering pipeline and of the task they perform.* ***Animation Stage****: Compute the deformation of the animation on each tetrahedron vertex as well as the Jacobian of the deformation 3.3.* ***Clipping Stage****: Clip the tetrahedron according to the nearest plane and cull the tetrahedron outside the frustum.* ***Mesh shading stage****: Generate triangle proxies with vertex attributes that encode the entry and the exit point of a ray coming from the camera 3.4.2.* ***Rasterizer****: Computes the entry and exit point for each ray that goes through each tetrahedron and produces a fragment containing these informations.* ***Fragment Stage****: Propagate the animation of the tetrahedron to the meshless shape 3.4.3.*

2.3. Deforming space around the shape

Another approach to animating meshless shapes is to deform the space in which the shape is defined. In this case, the deformation is done using a Cage-based Deformation [STH*24], where the shape is encased in a coarse triangle mesh that can be easily manipulated vertex by vertex. Then the deformation of the cage is used to deform the space inside it, thus propagating the animation of the cage to the meshless shape. Doing so allows for the deformation of any shape but requires adapting the rendering algorithm, those approaches can even be adapted to deform NeRFs [XH22; PYL*22].

An example of this is the curved ray tracing approaches [KK89; Fab96; SJNJ19], which propose to ray march the shape in its rest pose and to curve the ray according to the Jacobian of the deformation of the space. This allows the animation of SDF as long as the deformation of the space is a bijection (i.e., invertible), which limits the range of animation available. Furthermore, computing the deformation of space at rendering time is often too costly, and the deformed meshless representation is tessellated or stored in a 3D texture (See 2.2).

This kind of approach can solely be used to deform a shell around the surface, as is done for shell map-like approaches [PBFJ05; JMW07; Oga23; Rit06]. That limits the deformation to a meshed subset of the volume. Shell maps are more suited for real-time rendering, as part of the calculation can be done using the rasterization pipeline. However, Shell maps remain limited to creating effects near the surface.

Recently, an approach was proposed to animate implicit functions using Articulated Distance Fields [TTT*17]. This approach proposes the construction of a coarse tetrahedral mesh that allows implicit deformation of the underlying SDF. This method can deform a shape bounded by a small number of tetrahedra by deforming the tetrahedral mesh. This approach then proposes to ray-march the deformed shape by querying the deformation of every tetrahedron at every marching step. While interesting, this approach does not scale well for a high number of tetrahedra, limiting the expressivity of the depicted animation.

2.4. Interval Shading

Recently, the interval shading [Tri24] proposed an approach to compute ray intersections with a set of tetrahedra at rendering time in the scope of a regular rasterization pipeline. The method achieves this by encoding tetrahedra such that the front and back faces emit a single fragment containing both depths, allowing for recomputing their exact position in world space, which is close to what we need. Unfortunately, this approach does not support vertex attributes, preventing us from accessing information about the deformation of the tetrahedra in the fragment shader.

Our work proposes to use an approach similar to Articulated Distance Fields [TTT*17] by using a tetrahedral cage to deform the space around a shape and proposes to extend the *interval shading* [Tri24] method to be able to propagate the rest pose position and the Jacobian of the deformation to the fragment shader in order to apply the deformation of the tetrahedral cage to a given meshless shape in real-time.

3. Method

3.1. Overview

Our goal is to allow artists to animate meshless shapes using the classical animation workflow so that the shapes can be integrated into a traditional scene graph. In this section, we describe an overview of our method and indicate how it can be integrated into an animation workflow and a real-time rendering workflow.

3.1.1. Creating an animation

To animate a meshless shape with our method, we must define a tetrahedral cage with baked animation weights, a rig, and key poses, much like the regular skinning workflow.

In an ideal workflow, an artist could define the meshless shape, the tetrahedral cage, the rig, and the animation weights in a single software as one would do for a triangle mesh. As tetrahedral meshes are not yet supported in skinning software, we propose a method to generate the tetrahedral cage and the animation weights in Section 4.1.

Once the animation weights are baked into the vertices of the tetrahedral cage, the meshless shape, the cage, and the animation can be passed to our rendering pipeline.

3.1.2. Rendering pipeline

At the render time of each frame, our rendering pipeline is separated into five stages (figure 3):

- *The animation stage:* Apply the current frame animation on each vertex of our coarse tetrahedral mesh in a compute shader (See Section 3.3).
- *The clipping stage:* Apply a frustum culling and a near-plane clipping on tetrahedra.
- *The mesh shading stage:* Encodes each tetrahedron as a set of triangles as it was proposed in *interval shading* [Tri24], and bakes the position in rest pose space as well as the Jacobian of the deformation of each vertex as attributes (See Section 3.4.2).
- *Rasterizer:* Decodes the triangles produced by the previous state and generates, for each tetrahedron, fragments with attributes containing the entry and exit point of the ray that goes through the tetrahedron, their positions in the rest pose space, and the Jacobian of deformation at these coordinates.
- *The fragment shader stage:* Marches the meshless shape in its rest pose space within the bounds of the entry and exit point given by the *interval shading*, and uses the Jacobian of the deformation to reconstruct the normal of the surface with respect to the deformation (see Section 3.4.3).

3.2. Notation

Symbol	Description
p_{xyzw}	coordinate x,y,z and w of the point p
p^R	coordinate in rest pose space
p^W	coordinate in world space
p^H	homogenous coordinate in clip space
p^P	normalize device coordinate in clip space
λ_a	screen space interpolation coefficient
$A(p^R)$	animation function apply on a point p^R
J_p	Jacobian of A at point p

We introduce the following notation to distinguish the space in which we express the coordinates of our points. First, we have the point position in the rest pose space p^R. Then we can define the animation function of a given frame $A(p^R)$ and its Jacobian at point J_p at point p.
We note the coordinates of a point in world space:

$$p^W = \text{Model} \cdot A(p^R)$$

And the homogenous coordinate of the point in clip space:

$$p^H = \text{Projection} \cdot \text{View} \cdot p^W$$

And the normalized device coordinate in clip space:

$$p^P = \frac{p^H_{xyz}}{p^H_w}$$

3.3. Animation Stage

The animation is applied to the vertices of the tetrahedral cage in a compute shader (Figure 3). In this compute shader, we apply the deformation given by the function $A(t,v)$ of each vertex v at the time t and store their position in the rest pose space: v^R and the Jacobian of the deformation J_v as vertex attributes.

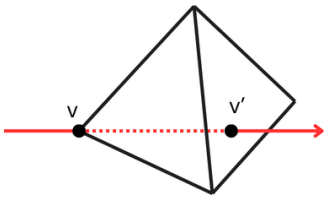

Figure 4: *A vertex v and its projection on the back faces v′.*

3.4. Rendering pipeline

3.4.1. Propagating the deformation

By applying the *interval shading* method [Tri24], We use a mesh shader to encode the tetrahedra as a set of triangles, with each fragment generated by their rasterization having attributes encoding the entry and exit points of the camera ray in the tetrahedron.

To achieve this, the method proposes to encode each tetrahedron as a set of triangles where each triangle encodes a front and a back face. Each vertex of given triangle v is encoded as $\{v^P_x, v^P_y, 0, 1\}$ with the following vector as an attribute $\{v^P_z, v'^P_z\}$ where v^P_z is the depth of the vertex, and v'^P_z is the depth of the vertex projected on the back faces (see Figure 4).

Once rasterized, these triangles generate fragments with attributes encoding the coordinates of the entry and exit points in the projected space. However, to correctly compute the intersection of the camera ray with the shape representation contained by our tetrahedron, we need to find the entry and exit points in the rest pose space, as well as the Jacobian of the deformation, to be able to reconstruct the normals of the deformed shape.

These attributes arrive baked as vertex attributes of the input tetrahedra; unfortunately, in the formulation presented above, setting w to 1 is necessary to encode the front and back faces together. Unfortunately, this prevents vertex attributes from being interpolated correctly by the rasterizer as it requires the w component to interpret the perspective correctly [Khr]. To correct this issue, we propose in Section 3.4.2 a extension of *interval shading* to encode vertex attributes that allow us to send the information we need to access in the rendering to the fragment stage.

3.4.2. Interpolation of attribute with Interval Shading

In the rasterization pipeline, for a point p in the triangle (abc) such that:

$$p^P_{xy} = a^P_{xy}\lambda_a + b^P_{xy}\lambda_b + c^P_{xy}\lambda_c \tag{1}$$

We know, by definition [Khr], that for any attribute f defined on the vertex of the triangle, we can interpolate linearly (in view space) from information in screen space :

$$f_p = {f_p}^H \cdot {p_w}^H \tag{2}$$

where:

$${f_p}^H = \frac{f_a}{a^H_w}\lambda_a + \frac{f_b}{b^H_w}\lambda_b + \frac{f_c}{c^H_w}\lambda_c \tag{3}$$

$$\frac{1}{p_w^H} = \frac{1}{a_w^H}\lambda_a + \frac{1}{b_w^H}\lambda_b + \frac{1}{c_w^H}\lambda_c \tag{4}$$

With *interval shading*, we know we can coerce the rasterizer into computing Equation 3 and 4.

To do so, given a vertex attribute f, we need to add the following vertex attributes to the triangle generated by the method:

- f_v/v_w^H and $f_{v'}/v'^H_w$ that encode the attributes for the vertex v and its value at v' projection of v on the back faces of the tetrahedron.
- $1/v_w^H$ and $1/v'^H_w$ that encode the deformation of the tetrahedron created by the perspective for the vertex v and its value at v' projection of v on the back faces of the tetrahedron.

Then, in the fragment stage, we receive the result of Equation 3 and 4 for the front and back faces, and we can use Equation 2 to get the perspective accurate interpolation of the attributes f.

In our context, we want to know the position of the entry and exit points in the rest-pose space and the Jacobian of the deformation at those points. Thus, we need to encode them as described above. Which gives us, for each vertex v, the following list of attributes:

$$\{\, v_z^P,\ v'^P_z,\ \frac{v^R}{v_w^H},\ \frac{v'^R}{v'^H_w},\ \frac{J_v}{v_w^H},\ \frac{J_{v'}}{v'^H_w},\ \frac{1}{v_w^H},\ \frac{1}{v'^H_w} \,\} \tag{5}$$

and we decode each attribute using Equation 2.

In practice, the result of Equation 4 can be derived from the fragment coordinate and the inverse of the projection matrix in the fragment shader to limit the number of vertex parameters. Thus, using FP32 precision, we have 104 bytes attributes per vertex, representing less than one-quarter of the available fragment input components on modern GPUs.

Attribute name	v_z^P	v'^P_z	$\frac{v^R}{v_w^H}$	$\frac{v'^R}{v'^H_w}$	$\frac{J_v}{v_w^H}$	$\frac{J_{v'}}{v'^H_w}$
Size (octet)	4	4	12	12	36	36

3.4.3. Rendering in rest pose space

Following the explanation in Section 3.4.2, we can access the entry a^R and exit point b^R in the rest pose space of the ray intersecting the tetrahedron. From there, we can apply a marching algorithm in rest pose space to render the shape that intersects a ray R defined as:

$$R(s) = (1-s)a^R + sb^R \tag{6}$$

for $s \in [0,1]$.

If a hit is found for a given s, to do accurate shading, we need to re-express the position of the hit p^R and the normal n^R in world space. As the deformation created by a tetrahedron is linear, we can get the position of the hit in world space using:

$$p^W = (1-s)a^W + sb^W \tag{7}$$

Then, we compute the Jacobian of the deformation at the hit position in the rest position p^R with:

$$J_p = (1-s)J_a + sJ_b \tag{8}$$

With J_a and J_b being the Jacobian at the entry and exit points that came as vertex attributes (see Section 3.4.2). Finally, we can compute the normal at the hit point with:

$$n^W = \texttt{inverse}(Model \cdot J_p)^T \cdot n^R \tag{9}$$

from there, we can use the position p^W and the normal n^W to compute our shading.

Smooth normal: The reason why we interpolate the Jacobian of the transformation within each tetrahedron instead of having a single Jacobian matrix per tetrahedron is to ensure a $\mathcal{C}^0$ continuity of the normal [Pho98] in world space as visible in Figure 5.

3.4.4. Alternative approaches to fetch the Jacobian

Multiple alternative solutions exist to get the Jacobian of the transformation in the fragment shader.

- Per tetrahedra: For each tetrahedron, we can store the Jacobian of the deformation in a buffer and read it when needed in the fragment shader. This would reduce the number of vertex attributes but lead to a discontinuity of the normal, creating visual artifacts.
- Per vertex: Similarly, we can store a Jacobian by vertex in a separate buffer, then fetch and interpolate the required Jacobian in the fragment shader. This would reduce the number of vertex attributes but requires four noncontiguous memory fetches of a 3 by 3 matrix and will increase the interpolation cost.
- In voxels: Finally, the Jacobian of the deformation can be stored in voxels in the rest pose of the shape and be sampled using the rest pose coordinates. Doing this would enable bilinear interpolation of the Jacobian to get $\mathcal{C}^1$ normal. However, it will drastically increase the memory cost and add multiple texture fetches.

In comparison, our approach increases the number of vertex attributes but uses the rasterizer to compute a large part of the Jacobian interpolation, which spares us potentially expensive memory fetch in the fragment shader.

4. Implementation

In this section, we describe how we implemented the approach presented in Section 3

4.1. Animation

Our method is compatible with all animation methods as long as the deformation can be defined for each vertex of a tetrahedral mesh at all times. We used two animation methods in our examples: implicit deformation function and rigging animation.

Linear blend skinning: To achieve linear blend skinning, we start by defining a rig. We do so in Blender using the boundary of the tetrahedral mesh as a previsualization proxy. Then, to compute the animation weight on the tetrahedral mesh, we use the *libigl* implementation[JP*18] of the bounded biharmonic weights (BBW) method [JBPS11]. We resolve the animation at rendering time as described in Section 3.1 and in Figure 3.

Implicit deformation: Our methods also support implicit deformation functions. Given an animation function $A(p,t)$, which for a time t gives the movement of any point p, we can apply this deformation to the vertices of our tetrahedra to create an animation. To

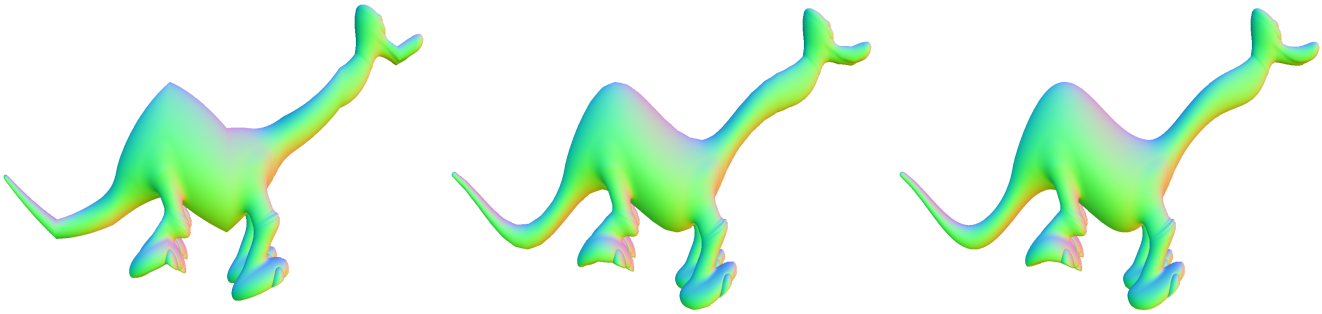

Figure 5: *Visual quality depends on the number of tetrahedra. In the three examples, we deform the geometry defined by the SDF [Ini15] with the deformation $A(x,y,z,t) = (x, y+0.1sin(10x+t), z)$. The varying parameter is the number of tetrahedra in the tetrahedral mesh, from left to right, 692, 10245, and 59171, respectively. The color of the shape is normal on the surface. Here, we can see that normals are $\mathcal{C}^0$ even when the number of tetrahedra is low.*

apply this in our pipeline, we also need to calculate the inverse of the Jacobian J of the deformation A for every point of our tetrahedral mesh. Figure 5 shows an example of this.

4.2. Creating a coarse tetrahedral mesh

In our example, we build a coarse triangle mesh that tightly encloses our geometry while being conservative. To do so, we used the same conservative marching cube algorithm as presented in NSLT [SJNJ19]. Then, we compute the coarse tetrahedral mesh by tet meshing the coarse triangle mesh with the *TetGen* Library [Si15], allowing us to choose the size of the tetrahedra. We manually choose this parameter for our test scenes to limit visual artifacts in the worst case. In our examples, 26k tetrahedra were enough to avoid visual artifacts at any time. Figure 5 demonstrates how the number of tetrahedra affects the rendering quality. The deformed geometry is $\mathcal{C}^0$ on the border of each tetrahedron but keeps the same level of detail and continuity inside each tetrahedron. In practice, reconstructing the normal using Jacobian interpolation allows the number of tetrahedra in a coarse mesh to remain relatively low without creating visual artifacts. The number of tetrahedra might need to be adjusted for more complex animation (higher maximum curvature, more expressive rig, etc.). In a professional system, this should be adaptive.

A criterion on the number of tetrahedra We can give a criterion to choose the minimal number of tetrahedrons so as to have no visible artifact; we want the error between the target deformation D and its discretized piecewise linear deformation D_{Linear} to be smaller than a pixel footprint. To do so, we can define the geometric maximum error inside a volume V:

$$E_V = \max_{p \in V} \|D(p) - D_{Linear}(p)\|$$

As we are doing linear interpolation inside a tetrahedron T, we have a more precise description of this error. If we suppose that the deformation D of a given animation is $\mathcal{C}^2$ in space, one can show that:

$$E_T \leq \frac{3}{8} M \times l^2$$

where l is the longest distance within the tetrahedron and M the maximum norm of the second derivative of f inside the tetrahedron [Jul20].

We can then define the pixel footprint maximum error that depends on the distance of the tetrahedron to the pinhole camera d, and the solid angle of a single pixel f/r (f is the FOV, and r the resolution of the screen):

$$P_T \leq \frac{E_T \times r}{d \times f} \leq \frac{3 \times M \times l^2 \times r}{8 \times d \times f}$$

By applying this error metric to the deformation presented in Figure 5, we find the error to be from left to right 34.3, 5.7, and 1.8 pixels. For this specific case, M is constant in space. Thus, we can choose the discretization by picking the same l value that keeps the error under a given threshold. For more complex deformations, M is not constant in space. Thus, the tet meshing must be adapted to account for that.

Remark One important thing to notice is that the number of tetrahedra necessary to avoid artifacts depends on the deformation, not the object's geometry. As the space deformation is often low frequency compared to the geometry, we need fewer tetrahedra than would require discretizing the meshless representation itself.

4.3. Low level optimization

The Interval Shading [Tri24] method proposed to compute the clipping in the mesh shading stage before encoding the tetrahedra. As a result, the clipped tetrahedron needed to be decomposed into a varying number of tetrahedra depending on how it was clipped. The mesh shading stage registers usage to support this, and the maximum triangle output was adapted to the worst-case scenario. As our approach requires us to use vertex attributes, the register usage would take a toll on performance. To avoid this, we split the clipping part of the algorithm into a separate compute shader before the mesh shader, as shown in figure 3. Consequently, the mesh shader now has a more stable number of triangle outputs and a better low-level resource parallelism of the computation.

5. Results

Our meshless shape is encased in a tetrahedral cage made of 26857 tetrahedra in our scenes. To demonstrate the versatility of our method, we used different shape representations. We used implicit SDF in Figure 7, solid voxels (see Figure 1, Left), SDF stored in a 3D texture (see Figure 6), density field stored in a 3D texture (see Figure 8), or even a hybrid representation where the coarse SDF is stored in a 3D texture, and the fine details are computed via an implicit noise function (see Figure 9). The source code to generate our figures is available here.

Self intersect animation Our method supports self-intersection of the deformed shape. This was an important limitation of the curved ray tracing approaches [SJNJ19; Fab96]. In Figure 7, the head and foot are animated into the same position in the world space, making them intersect without issues (see Supplemental Video).

Dynamic editing of the geometry In figure 9, the SDF is defined by two components, a base SDF in the form of a dinosaur [Ini15] stored in a 3D textures and a FBM (Fractional Brownian Motion) that can be updated dynamically each frame as can be seen in the Supplemental video.

Transparency As our method uses the regular rasterization pipeline to compute the ray tetrahedra intersection, scattering effects are not natively compatible with our approach, as it would require sorting the fragments before blending them. However, optical depth base rendering is possible (See figure 8). Here, the shape is a density field encoded in a 1024^3 3D texture.

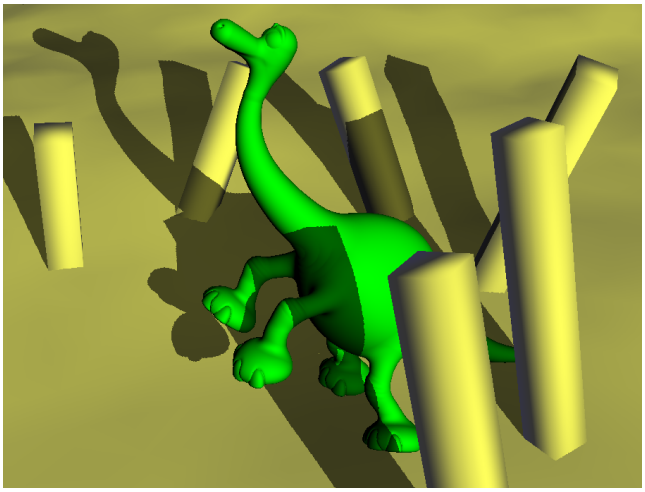

Figure 6: ***Shadow*** *Example of an SDF animated by our method integrated into a traditional scene made of a triangle mesh using a shadow map. Here, Arlo is marked by the scene and masks part of the scene. Similarly, Arlo casts and receives shadow from the scene and itself.*

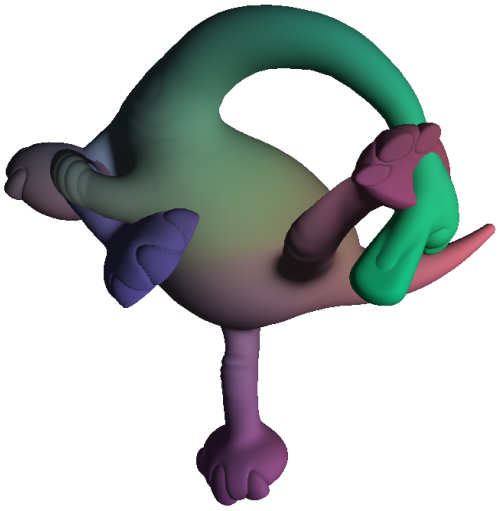

Figure 7: ***Self intersection****: Self intersection of the deformed shape. The head and the left foot are animated into the same position to create an intersection.*

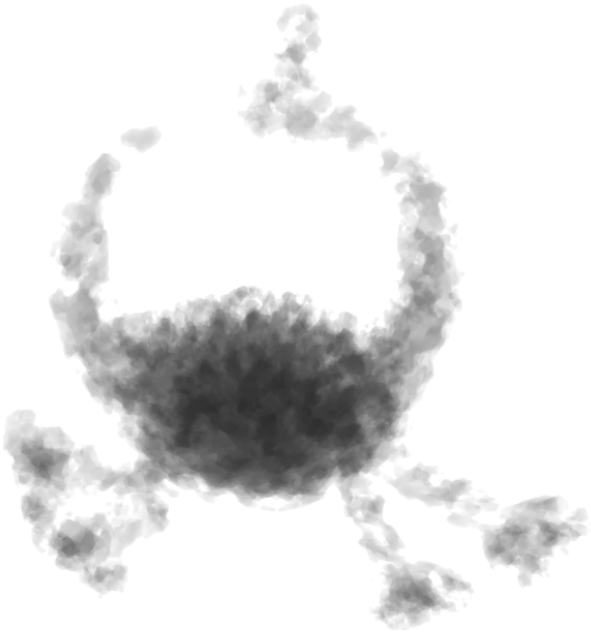

Figure 8: ***Transparent****: Our method applied to a density field. This showcases the compatibility of our method with transparent shapes.*

5.1. Performances

In all our tests, we used a computer with RTX 4080 16 GiB GPU and AMD Ryzen 9 79000X 12-Core CPU and rendered in Full HD (1920×1080). The performance of our method is shown in Table 1. In this table, we can see that our method remains real-time in all our tests. The cost of our method is discussed in the next section 5.2.

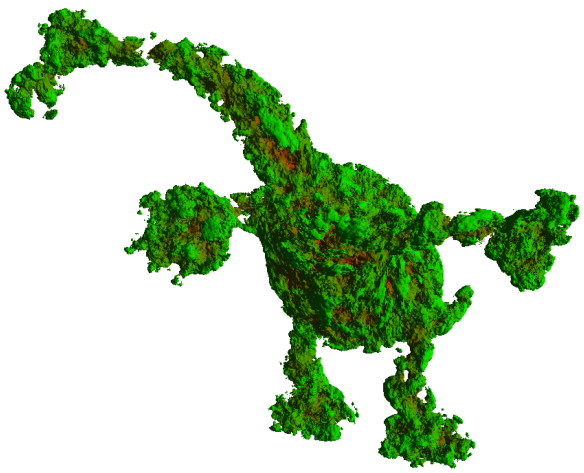

Figure 9: ***FBM****: example of an SDF enriched with an animated FBM (Fractional Brownian Motion).*

Table 1: *Performance of our method.*

Scene	figure	cost (ms)	FPS
Voxel model	1	3.2	310
FBM	1	9.3	107
Shadow	6	3.9	250
Self Intersect	7	1.3	780
Transparent	8	1.5	660

5.2. Cost of our method

To evaluate the performance of our method, we compared the cost of rendering the meshless shape without our pipeline to our result. To do so, we compare the performance of:

- A reference method with a meshless representation in rest pose space rendered using ray marching without our method.
- A scene using our method by setting the animation function to an identity $A(p^R) = p^R$ with the same point of view as the reference image. Doing so allows us to have the same pixel coverage in the resulting images. (Figure 10).

In our method, the tetrahedral mesh ensures that rays start close to the shape. For evaluating density fields, we use fixed-step ray marching and sphere marching for SDF evaluation. In the case of the density field, to create a fair comparison with the reference method, we start the marching at the first intersection between the ray and a bounding sphere and use a sphere marching algorithm to approach the shape. Once the distance between the last step and the shape is under a threshold, we fall back to the same fixed-step ray marching as in our approach. The results are compiled in Table 2. Our method uses the tetrahedral cage as an acceleration structure, so we ensure that our marching is only done close to the shape. Moreover, our marching is split into small intervals. This avoids thread divergence during the marching and helps us achieve better parallelization of our task, as intervals participating in the same pixel are parallelized on multiple SMs. The main additional cost of our method comes from the fact that all tetrahedra are processed in parallel, in random order; as a consequence, our early stop is not optimal: some hidden tetrahedra are still computed. This effect can be seen in the performances of the scene *FBM*, where our framerate is lower than the reference scene.

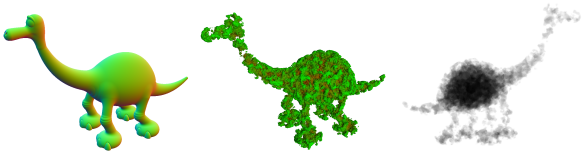

Figure 10: ***Left****: SDF,* ***Middle****: SDF with FBM,* ***Right****: Transparent density field.*

Table 2: *Overhead of our method compared to rendering the shape in rest pose without our method.*

Scenes	reference (ms)	ours (ms)	factor
SDF store in Voxel Grid	2.0	1.3	×**0.65**
Analytic SDF	2.1	5.5	×**2.6**
FBM	5.7	9.3	×**1.6**
Transparent	3.0	1.5	×**0.5**

Finally, we evaluated the impact of animating tetrahedra and encoding them to be about 1 ms. We achieved this measure by rendering our scenes with our method in an image of resolution 1 by 1 to negate the impact of the fragment shader and ROPs.

6. Conclusion

In our method, we proposed a pipeline to achieve the animation of meshless shapes in real time within a traditional rasterization pipeline. Our results and supplemental video demonstrate that our method can deform a meshless shape with traditional animation handles (animation rig) while allowing for a large range of motions. Our method uses the rasterization pipeline and can share its output frame buffer with a triangle rasterization pipeline, which allows us to use traditional on-the-fly effects such as shadow mapping (a GBuffers-based method could also be used). Finally, our method allows for animating shapes that cannot be meshed, such as highly detailed surfaces or density fields.

7. Limitation and Future Works

As mentioned in Section 3.1, current skinning softwares do not support the visualisation and weight painting on tetrahedral mesh, which forces us to rely upon automatic methods [JBPS11] to compute animation weight. This will limit the range of animation possible with our method. However, this problem is only related to the software used to design animation and not our method.

Our approach relies on a piece-wise linear approximation of the deformation, which will create visual artifacts if the number of tetrahedrons is not high enough. If the deformation generated by the animation has a too-strong curvature, our method may require

a large number of tetrahedra, which might impact the performance. At the moment, the resolution of the tetrahedral cage has to be manually chosen by the user and adapted to the animation's properties, as explained in Section 4.2. Furthermore, by defining an adapted subdivision algorithm, the tetrahedral cage could be refined during the animation (at run time) to adapt the curvature of the deformation.

Finally, our method operates unordered, limiting its suitability for complex volumetric effects. However, a potential solution lies in sorting the generated intervals for proper blending. This enhancement could improve our approach's compatibility with various meshless representations, such as radiance fields and volumetric lighting.

Aknowledgement

We thank Fabrice Neyret for his guidance during this project. We thank Laurence Boissieux for helping create the animation used in our examples. Finally, we thank Antoine Richermoz, Mathéo Moinet, and Nolan Mestres for helping proofread the article. We thank Inigo Quilez for allowing us to use his arlo SDF shader.

References

[AZ21] AYDINLILAR, MELIKE and ZANNI, CÉDRIC. "Fast ray tracing of scale-invariant integral surfaces". *Computer Graphics Forum* 40.6 (Sept. 2021). DOI: 10.1111/cgf.14208. URL: https://inria.hal.science/hal-03169283 2.

[CD97] CANI, MARIE-PAULE and DESBRUN, MATHIEU. "Animation of Deformable Models Using Implicit Surfaces". *IEEE Transactions on Visualization and Computer Graphics* 3.1 (1997), 39–50. DOI: 10.1109/2945.582343 2.

[CJ91] COQUILLART, SABINE and JANCÉNE, PIERRE. "Animated free-form deformation: an interactive animation technique". *Proceedings of Siggraph'91* 25.4 (July 1991), 23–26. ISSN: 0097-8930. DOI: 10.1145/127719.122720 2.

[Coq90] COQUILLART, SABINE. "Extended free-form deformation: a sculpturing tool for 3D geometric modeling". *Proceedings of SIGGRAPH'90*. 1990. ISBN: 0897913442. DOI: 10.1145/97879.97900 2.

[Fab96] FABRICE, NEYRET. *Local Illumination in Deformed Space*. Tech. rep. INRIA, 1996. URL: https://inria.hal.science/inria-00073835 3, 7.

[Ini] INIGO, QUILEZ. *IQ shaders*. URL: https://www.shadertoy.com/user/iq (visited on 2025) 2.

[Ini13] INIGO, QUILEZ. *Smooth CSG*. 2013. URL: https://iquilezles.org/articles/smin/ 2.

[Ini15] INIGO, QUILEZ. *Arlo SDF*. 2015. URL: https://www.shadertoy.com/view/4dtGWM 6, 7.

[JBPS11] JACOBSON, ALEC, BARAN, ILYA, POPOVIĆ, JOVAN, and SORKINE, OLGA. "Bounded Biharmonic Weights for Real-Time Deformation". *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)* (2011) 5, 8.

[JLSW02] JU, TAO, LOSASSO, FRANK, SCHAEFER, SCOTT, and WARREN, JOE. "Dual contouring of hermite data". *Proceedings of Siggraph'02*. 2002. ISBN: 1581135211. DOI: 10.1145/566570.566586 2.

[JMW07] JESCHKE, STEFAN, MANTLER, STEPHAN, and WIMMER, MICHAEL. "Interactive smooth and curved shell mapping". *Proceedings of EGSR'07*. 2007. ISBN: 9783905673524 3.

[JP*18] JACOBSON, ALEC, PANOZZO, DANIELE, et al. *libigl: A simple C++ geometry processing library*. 2018. URL: https://libigl.github.io/ 5.

[Jul20] JULIUS, ORION SMITH. *Linear Interpolation Error bound*. 2020. URL: https://ccrma.stanford.edu/~jos/src/Linear_Interpolation_Error_Bound.html 6.

[Kap02] KAPLER, ALAN. "Evolution of a VFX voxel tool". *ACM SIGGRAPH 2002 Conference Abstracts and Applications*. SIGGRAPH '02. 2002, 179. ISBN: 1581135254. DOI: 10.1145/1242073.1242192 2.

[Khr] KHRONOS. Vulkan Documentaion. URL: https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html#primsrast-polygons (visited on 04/2025) 4.

[KK89] KAJIYA, J. T. and KAY, T. L. "Rendering fur with three dimensional textures". *SIGGRAPH Comput. Graph.* 23.3 (July 1989), 271–280. ISSN: 0097-8930. DOI: 10.1145/74334.74361 3.

[LC87] LORENSEN, WILLIAM E. and CLINE, HARVEY E. "Marching cubes: A high resolution 3D surface construction algorithm". *Proceedings of SIGGRAPH '87*. 1987. ISBN: 0897912276. DOI: 10.1145/37401.37422 2.

[Oga23] OGAKI, SHINJI. "Nonlinear Ray Tracing for Displacement and Shell Mapping". *SIGGRAPH Asia 2023*. 2023. ISBN: 9798400703157. DOI: 10.1145/3610548.3618199 3.

[PBFJ05] PORUMBESCU, SERBAN D., BUDGE, BRIAN, FENG, LOUIS, and JOY, KENNETH I. "Shell maps". *ACM Trans. Graph.* (July 2005). ISSN: 0730-0301. DOI: 10.1145/1073204.1073239 3.

[Pho98] PHONG, BUI TUONG. "Illumination for computer generated pictures". *Communication of the ACM*. 1998, 95–101. ISBN: 158113052X. URL: https://doi.org/10.1145/280811.280980 5.

[PYL*22] PENG, YICONG, YAN, YICHAO, LIU, SHENGQI, et al. "CageNeRF: Cage-based Neural Radiance Field for Generalized 3D Deformation and Animation". *Advances in Neural Information Processing Systems*. 2022. URL: https://dl.acm.org/doi/10.5555/3600270.3602547 3.

[Rit06] RITSCHE, NICO. "Real-time shell space rendering of volumetric geometry". *Proceedings of GRAPHITE'06*. 2006. ISBN: 1595935649. DOI: 10.1145/1174429.1174477 3.

[RMP*24] RISO, MARZIA, MICHEL, ÉLIE, PARIS, AXEL, et al. "Direct Manipulation of Procedural Implicit Surfaces". *ACM Transaction on Graphics (SIGGRAPH Asia '24 Conference Proceedings)* (2024). DOI: 10.1145/3687936 2.

[Si15] SI, HANG. "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator". *ACM Transactions on Math. Softw.* 41.2 (Feb. 2015). ISSN: 0098-3500. DOI: 10.1145/2629697 6.

[SJNJ19] SEYB, DARIO, JACOBSON, ALEC, NOWROUZEZAHRAI, DEREK, and JAROSZ, WOJCIECH. "Non-linear sphere tracing for rendering deformed signed distance fields". *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* (2019). DOI: 10/dffm 2, 3, 6, 7.

[STH*24] STRÖTER, D., THIERY, J. M., HORMANN, K., et al. "A Survey on Cage-based Deformation of 3D Models". *Computer Graphics Forum* 43.2 (2024). DOI: https://doi.org/10.1111/cgf.15060 2, 3.

[Tri24] TRICARD, THIBAULT. "Interval Shading: using Mesh Shaders to generate shading intervals for volume rendering". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7.3 (Apr. 2024), 1–11. DOI: 10.1145/3675380. URL: https://hal.science/hal-04561269 1–4, 6.

[TTT*17] TAYLOR, JONATHAN, TANKOVICH, VLADIMIR, TANG, DANHANG, et al. "Articulated distance fields for ultra-fast tracking of hands interacting". *ACM Trans. Graph.* 36.6 (Nov. 2017). ISSN: 0730-0301. DOI: 10.1145/3130800.3130853 3.

[XH22] XU, TIANHAN and HARADA, TATSUYA. *Deforming Radiance Fields with Cages*. 2022. arXiv: 2207.12298 [cs.CV]. URL: https://arxiv.org/abs/2207.12298 3.