Spline Based Shell Mapping for Smooth Voxel Deformation

Vincent Groen* Max Rensen* Elmar Eisemann Mathijs Molenaar

Delft University of Technology

Abstract

Current techniques for animating volumetric models are often computationally expensive or suffer in quality for smooth animations. This paper presents a method for smoothly deforming volumetric objects in real time by making use of Cubic Hermite splines. Instead of deforming the volume directly, our method transforms coordinates according to a spline-based deformation. A heuristically adapted distance field, along with a tight intersection algorithm are provided as optimization techniques. Additionally, a shading method using deformed normal vectors is presented.

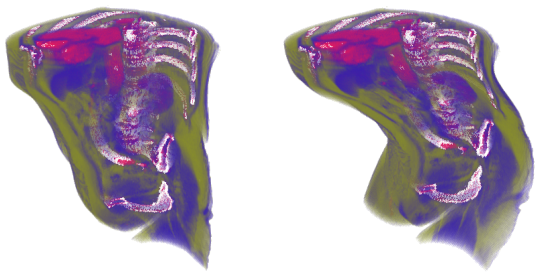
1 Introduction

Voxels are widely used, typically in medical contexts, but they also find relevance in real-time applications such as games or simulations. Usually, voxel representations are static or consist of animations that are pre-captured per time step. Nevertheless, in order to enable the step from a static representation to a useful primitive that can be animated in a real-time application, specialized methods are required.

A common method comprises of splitting the volumetric object into parts, applying affine transformations to these parts and rendering them individually [1]. This method is computationally inexpensive and sufficient for many types of animations, but lacks the ability to perform more organic animations.

VoxMorph [2] is another technique for animating volumes, which remaps voxels by approximating a mesh around the volume, transforming that mesh and finally determining for each voxel in the grid of the deformed mesh, the corresponding voxel in the original volume. The technique is able to arbitrarily deform volumes, similar to how polygonal models are deformed. However, each deformation requires a large amount of pre-processing time, which makes it unsuitable for arbitrary real-time animation.

Gagvani and Silver proposed a skeletal animation technique that exclusively makes use of voxels, yet allows the user to employ traditional tools intended for skeletal animation of polygonal objects [3]. Their method works by thinning a volumetric object into a skeleton, manually animating that skeleton with the aforementioned traditional tools and finally reconstructing the original model from the deformed skeleton. Though this method works well for organic animations of human models and can likely be extended to more general models, the reconstruction is demonstrated to be improper, which might be undesirable for some use cases.



(a) Without deformation **(b)** With spline deformation

Figure 1: A 3D voxel model of a torso without deformation and with *spline map* deformation.

*Equal contribution.

In this paper, we propose the *spline map*, a novel method for smoothly deforming volumetric objects, inspired by the *shell map* technique originally proposed by Porumbescu et al. [8]. An example of this deformation is shown in Figure 1. Our technique makes use of splines to smoothly deform the space the object lies in, which results in the voxels appearing to smoothly curve with the spline. The algorithm runs in real time and can be integrated with traditional ray-based techniques, the major limitation being that splines must be monotonic with respect to a predefined base.

A concrete definition of the *spline map* and a description of the methods is given in Section 3. More details on the implementation are provided in Section 4, followed by the results in Section 5.

2 Related Work

The smooth deformation of volumetric objects is typically accomplished by modifying the ray marching algorithms used for rendering. This can be categorized into two groups, namely methods that deform the object space and methods that deform the rays.

Shell maps [8] belong to this first category and are used to add surface detail to polygonal meshes by sampling from 3D volumes. The main principle of *shell maps* is to map coordinates from *shell space* to texture space. This shell space is defined by prisms that are extruded from a triangular base corresponding to triangles of the original polygonal model, or a curved triangular base in a later extension [4]. Within the so-defined volume, barycentric coordinates are used to associate positions in the prisms with a location in a volumetric texture.

The second category contains ray deflectors [5], a flexible method applicable to volumes, which redirects the trajectory of a ray based on various factors, such as the depth of the ray and the type of deflector. This method can be extended to support almost any arbitrary deformation of volumes, though this generality could come at the cost of performance and manual adjustments.

Our work belongs to the first category and is directly inspired by *shell maps*, transforming coordinates in deformed shell space to texture space. However, deflectors could conceivably be used to achieve similar transformations, though it is outside the scope of this paper.

3 Spline Map

This section will concretely define the *spline map*, together with the steps required to go from a point in *spline space* (the inside of the *spline map*, analogous to shell space[8]), to the corresponding point in texture space. Relevant implementation details of the *spline map* are omitted and instead elaborated upon in Section 4.

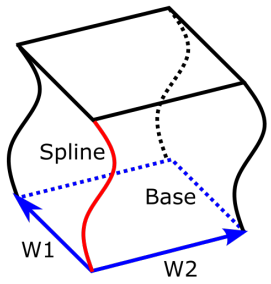


Figure 2: *Spline map* definition, consisting of a base in the form of a quad with two spanning vectors $W1$ and $W2$ (blue), and four horizontal edges consisting of identical splines (reference spline shown in red).

3.1 Spline Map Definition

In this paper, a *spline map* is defined by a base and a spline (see Figure 2).

The base is a quad, characterized by a vertex and two spanning vectors, $W1$ and $W2$; however, any form able to characterize a finite plane suffices for the base.

The spline is a standard Cubic Hermite spline, where each dimension of the spline is described by the following cubic equation.

$$at^3 + bt^2 + ct + d|_{t\in[0,1]} \tag{1}$$

The three-dimensional spline can be represented with the following matrix.

$$S(t) \equiv \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \tag{2}$$

Four identical copies of the spline start at every vertex of the base, at $S(0)$. At the end of these splines, at $S(1)$,

a copy of the base is constructed to complete the *spline map*. These four splines, together with the two bases, define the shape of the *spline map*. However, only a single spline and base are necessary for the computations.

To simplify intersections, we restrict the spline to satisfy the following assumption:

- For every $t \in [0, 1]$, if a plane parallel to the base is translated to $S(t)$, it must intersect with the spline exactly once, i.e. at $S(t)$. This means the spline must be monotonic along the normal vector of the base plane.

3.2 Spline Plane Intersection

Obtaining the intersection between a spline and a plane is a vital part of the *spline map*, and its use will be explained in later sections. Finding the value for t such that the point $S(t)$ lies on the given quad P is done by performing the following steps:

- Use a transformation M to transform P so that it is aligned with the xz-plane, and centered at the origin. This transformed plane will be referred to as P'.
- Apply transformation M to the y component of spline S. This transformed cubic will be referred to as S'.
- Compute the roots of the cubic equation S'. This will give up to three values t for which S' intersects the xz-plane.
- For each resulting value for t, discard it if it is not in the range $[0, 1]$ or if $S(t)$ is not within the bounds of the quad P.

Spline Transformation

A spline can be transformed using a standard transformation matrix, but it merits an explanation. When the spline in matrix form (see Equation (2)) is treated as a transformation matrix, it is clear that d represents the translation and a, b and c comprise the remaining transformation. As such, we apply a translation only on the d component, whilst a rotation is applied to all four components.

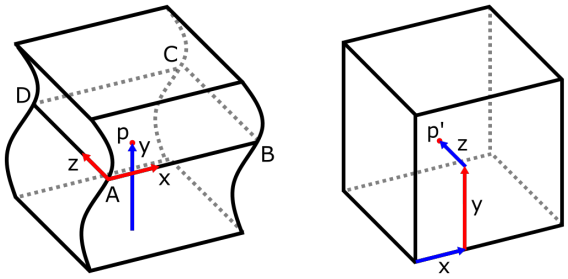
Computing the Roots

To determine the intersection of a spline with any plane, the roots of the aforementioned cubic equation are to be computed. This is done by first constructing the matrix M that transforms the base to be aligned with the xz-plane, as described by Pique [7]. This means that for every point on the base plane, $y = 0$. Next, the cubic equation of the y component of the transformed spline is solved for 0, using a depressed cubic, in order to find its roots with values t. The resulting solutions are filtered to be within $[0, 1]$, and subsequently the intersection positions can be determined by computing the position on the original spline $S(t)$ for t, using Equation (2).

3.3 Coordinate Mapping

Given a coordinate in spline space, the coordinate can be mapped to texture space. Subsequently, this texture space coordinate is directly used to sample the 3D volume.

To perform this mapping, a copy of the base quad is first translated to the current coordinate. Then the spline plane intersection algorithm described in Section 3.2 is used to find the intersection position with the spline. Let the translated base be described by $ABCD$, and the coordinate to be mapped by p (see Figure 3a).



(a) Spline space **(b)** Texture space

Figure 3: Mapping of coordinate inside spline space to the corresponding coordinate in texture space.

The x and z components of the texture coordinate are determined by projecting the vector Ap onto the spanning vectors $W1$ and $W2$ of the translated base. That is, the x component is Ap projected onto $W2$, or AB. And the z component is Ap projected onto $W1$, or AD.

After that, the y component is determined as the distance to the base of the *spline map*. Finally, all components are scaled to be within $[0, 1]$, by dividing by the length of the spanning vectors for x and z and the distance between the top and bottom planes for y, in order to find their position in texture space (see Figure 3b).

4 Implementation

In this section, the implementation details for rendering volumes using a *spline map* are provided. Intersection points with the *spline map* are found using a customized ray marching algorithm. Once an intersection is found, conventional ray tracing techniques can be applied.

4.1 Ray Intersection algorithm

In order to render a deformed volume, a ray is cast for each pixel on the screen. Subsequently, this ray is marched, and for each step the coordinate mapping algorithm described in Section 3.3 is applied. When a non-empty voxel is sampled, the appropriate action is taken, depending on the desired result. For example, the voxel color can directly be assigned to the pixel from which the ray originated, it can be reflected for lighting purposes, or if an accumulative volume is being rendered, the ray will continue until it accumulates enough density or exits the *spline map*. In this paper we choose the first of these options. But first, the entry and exit point of the *spline map* need to be determined for a ray.

Spanning Plane

The use of a conservative bounding box that encompasses the *spline map* naturally suffers in performance when the *spline map* does not cover the majority of the bounding box. In such a case, a ray is likely to intersect with the bounding box but only partially with the *spline map*, which results in unnecessary marching and coordinate mapping. To address this problem, a more accurate intersection algorithm is proposed which gives the intersection points of a ray and a *spline map* directly. This algorithm covers two cases, the ray enters or exits the *spline map* either from one of the sides or the top or bottom base.

To solve for the first case, two planes are constructed using the ray as one spanning vector and W_1 or W_2 as the other. Recall that W_1 and W_2 span the base (see Figure 2). Next, the algorithm uses the method described in Section 3.2 to calculate the intersection of each of the two planes with two splines on opposite corners of the base. The resulting intersection points, combined with the knowledge that all splines in a *spline map* are identical, allows two line segments to be constructed which are fully contained in their respective plane and connect two neighbouring splines. These line segments are sufficient to calculate the intersection of the ray with the sides of the *spline map*, namely by finding the intersection of the ray and the line segments, which must exist as the ray spans each plane together with a vector parallel to the corresponding line segment.

Figure 4 illustrates this algorithm on the particular case that the ray intersects only the two opposing sides. In such a case, using a single plane with the correct spanning vector is sufficient for the algorithm to find the entry and exit points.

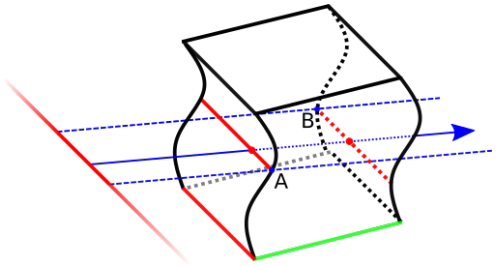


Figure 4: Illustration of the intersection algorithm, A and B indicate the points where the constructed plane intersects a spline.

The second case, where the ray enters or exits the *spline map* from the top or bottom base, is handled by determining the possible intersections of the ray with both bases.

Performing all the previously outlined steps results in a set $T = \{t_1, \ldots, t_k\}$ containing up to six possible t values for the intersection points on the ray. The entry point t_{entry} and the exit point t_{exit} can be found by taking the minimum and maximum of the set T respectively.

4.2 Optimization

The ray marching process described in Section 4.1 makes use of a constant step size. In order to prevent skipping over voxels, this step size could be scaled with the voxel dimensions. However, this would result in many wasted steps for large unfilled areas. A possible way to resolve this is using an unsigned distance field [6], heuristically modified to account for *spline map* deformations.

This is achieved by first retrieving the value in the distance field for the current position in the ray marching algorithm. Subsequently, this distance is added in the direction of the ray to determine the target position in the non-deformed volume. The difference in deformation between the two points is computed, for which a negative difference is subtracted from the distance, and a positive difference is added to the distance in the following manner:

$$\vec{D} = (d\vec{r} + \vec{\delta})^2 \cdot sign(d\vec{r} + \vec{\delta}) \tag{3a}$$

$$d' = \sqrt{max\left(0, \sum_{n=\{x,y,z\}} D_n\right)} \tag{3b}$$

Where d is the original distance, d' is the new distance, $\vec{r}$ is the normalized ray direction in texture space and $\vec{\delta}$ is the difference in deformation. This formula makes sure that a negative deformation difference on any axis n where $-\delta_n > d_n$, will subtract from the distance, instead of adding to it. Note that there is no deformation on the y-axis, so $\delta_y = 0$.

This new distance d' is then used for the distance field rendering algorithm. It is important to realize that this optimization is a heuristic and extreme deformations might result in certain voxels being skipped (when $d' \leq 0$). However, this has not been perceived in practice and is unlikely to happen for reasonable deformations. A depiction of this algorithm applied to a 2D deformation is shown in Figure 5.

4.3 Shading

In order to add shading to the rendered models, normals are computed for each visible voxel using a gradient field. This gradient field holds the gradient of the volume for each axis of a voxel, indicating the direction of the surface. The normals of an object have to deform when the object does, which is achieved by determining the slope of the spline at the current voxel position and rotating the normal according to the angle of this slope. This is only done for the x and z-axis, since the y-axis is not deformed, resulting in the following angles.

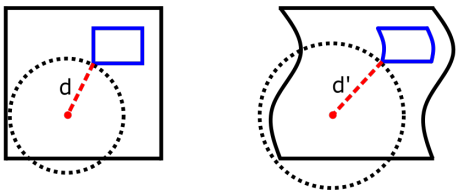


Figure 5: Example of a distance field correction from the original distance d to d' for a 2D deformation.

$$\begin{aligned} \phi_x &= arctan(S'(y)_x) \\ \phi_z &= arctan(S'(y)_z) \end{aligned} \tag{4}$$

Where $S'(t)$ is the derivative of the spline function $S(t)$, and $y \in [0, 1]$ is the normalized y-coordinate of the voxel in texture space. These angles are then used to compute the new normal n' as follows.

$$\begin{aligned} n'_x &= n_x \cdot cos(\phi_x) \\ n'_y &= n_y \cdot + n_x \cdot sin(\phi_x) + n_z \cdot sin(\phi_z) \\ n'_z &= n_z \cdot cos(\phi_z) \end{aligned} \tag{5}$$

Finally, these normals can be used to apply shading.

5 Results

With the techniques described in Section 4, results regarding the quality and performance of the *spline map* can be presented. These results were determined with an implementation using OpenGL 4.5 on a computer with an AMD RX 480 and an Intel i7 7700K.

Figure 6 displays various voxel objects along with arbitrary deformations. Figure 6c demonstrate the effectiveness of the heuristic for extreme deformations.

Additionally, a large-scale application is shown in Figure 7, demonstrating an underwater scene with smoothly deforming water plants and a fish.

A performance analysis is shown in Table 1. The frame times were acquired with a step size of 0.001 and animating the objects by changing the spline tangents over time according to a sine wave. These results demonstrate real-time frame rates of the algorithm, even without any optimizations. With the distance field optimization, the computation time per frame decreases, on average, with a factor of 2, without showing any perceivable artifacts in the results.

Average Compute Time

Model (Dimensions)	Standard	DF
Orange $(256 \times 256 \times 163)$	9.60 ms	4.39 ms
Pig $(256 \times 256 \times 180)$	10.44 ms	4.44 ms
Bunny $(512 \times 361 \times 512)$	10.93 ms	5.67 ms
Underwater scene $2\times(64 \times 64 \times 64)$ + $(128 \times 128 \times 256)$	37.13 ms	19.05 ms

Table 1: Benchmark of the average computation time per frame in milliseconds of various models with arbitrary deformations and diffuse shading, with and without the distance field (DF) optimization.

6 Discussion

The results demonstrate that the proposed algorithm can smoothly deform arbitrary volumetric objects in real time on consumer hardware. The advantages of *spline maps* include general purpose smooth animations and deformations that do not have to be redefined for every object. Additionally, its use of ray marching allows it to be integrated into traditional ray-traced voxel scenes with no further modifications.

There are however several limitations of *spline maps*, mainly related to its flexibility. Firstly, the splines must be monotonic along the normal of the base, otherwise results are undefined. Secondly, the splines of each edge must be identical, which limits the possible deformations that can be achieved. Lastly, the base of the *spline map* must be defined by quads, which have straight edges instead of splines, again limiting the possible deformations that can be achieved.

The *spline map* is not a universal solution for animating or deforming volumetric objects. Different solutions, perhaps in combination with *spline maps*, may be more applicable depending on the use case.

7 Conclusion

In this paper, we presented a technique for smoothly deforming volumetric objects using splines. This technique takes a new approach to the deformation of voxel objects, by marching through spline space and converting the coordinates to texture space. Furthermore, with the use of a heuristic distance field and a tight intersection algorithm, real-time frame rates are achieved on consumer hardware. The gradient field for finding the deformed normal vectors also ensures that the use of traditional lighting techniques, such as diffuse shading, is not limited in any way.

Future research could focus on non-trivial extensions to increase the versatility of the *spline map*. For instance, instead of employing splines in the form of a cubic function, any continuous monotonic function could be implemented, provided its roots can be found efficiently. Another area for future work is in assigning discrete splines to each of the edges, instead of identical ones, which would greatly increase the flexibility of the algorithm, but also its complexity.

Acknowledgements

For this work, we would like to thank Prof. Elmar Eisemann for supervising our process and guiding us along the way. Additionally, the help of Mathijs Molenaar, who reviewed with this paper, is greatly appreciated.

References

[1] Asbjørn E. Espe, Øystein Gjermundnes, and Sverre Hendseth. "Efficient Animation of Sparse Voxel Octrees for Real-Time Ray Tracing". In: 2019. DOI: 10.48550/arXiv.1911.06001.

[2] Noura Faraj, Jean-Marc Thiery, and Tamy Boubekeur. "VoxMorph: 3-scale freeform deformation of large voxel grids". In: *Computers & Graphics* 36.5 (2012), pp. 562–568. ISSN: 0097-8493. DOI: 10.1016/j.cag.2012.03.020.

[3] Nikhil Gagvani and Deborah Silver. "Animating Volumetric Models". In: *Graphical Models* 63.6 (2001), pp. 443–458. ISSN: 1524-0703. DOI: 10.1006/gmod.2001.0557.

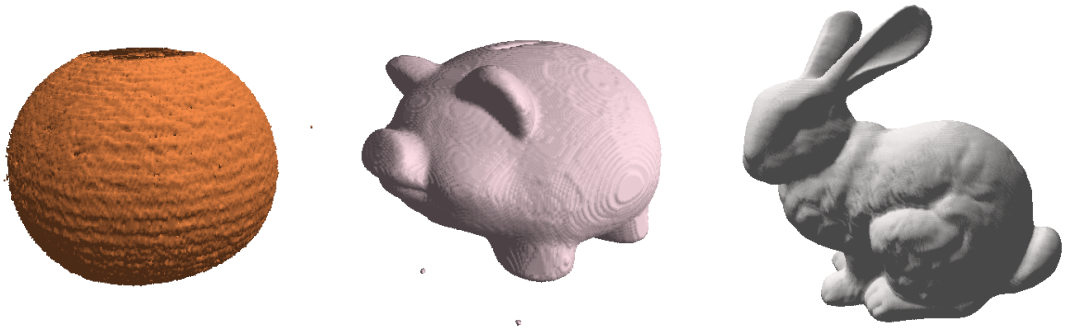
[4] Stefan Jeschke, Stephan Mantler, and Michael Wimmer. "Interactive Smooth and Curved Shell Mapping". In: *Rendering Techniques*. Ed. by Jan Kautz and Sumanta Pattanaik. The Eurographics Association, 2007. ISBN: 978-3-905673-52-4. DOI: 10.2312/EGWR/EGSR07/351-360.

[5] Yair Kurzion and Roni Yagel. "Space Deformation using Ray Deflectors". In: *6th Eurographics Workshop on Rendering 95*. 1995, pp. 21–32.

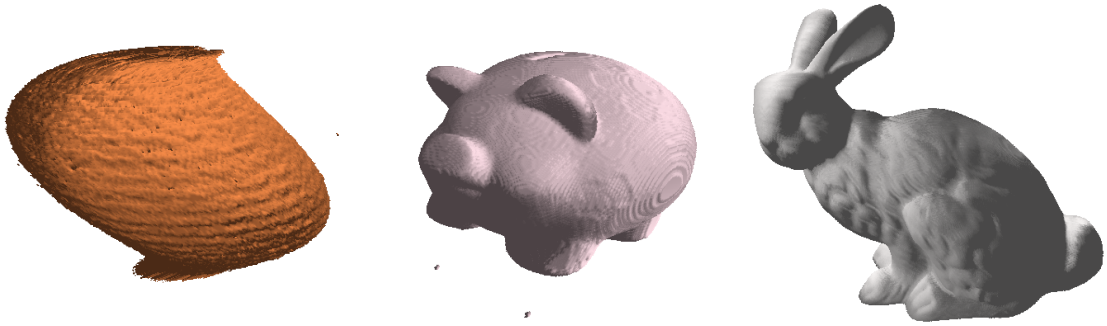
[6] C.R. Maurer, Rensheng Qi, and V. Raghavan. "A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25.2 (2003), pp. 265–270. DOI: 10.1109/TPAMI.2003.1177156.

[7] Michael E. Pique. "Rotation Tools". In: *Graphics Gems*. Ed. by Andrew S. Glassner. Morgan Kaufmann, 1990, pp. 465–469. ISBN: 978-0-08-050753-8. DOI: 10.1016/B978-0-08-050753-8.50108-X.

[8] Serban D. Porumbescu et al. "Shell Maps". In: SIGGRAPH '05 (2005), pp. 626–633. DOI: 10.1145/1186822.1073239.



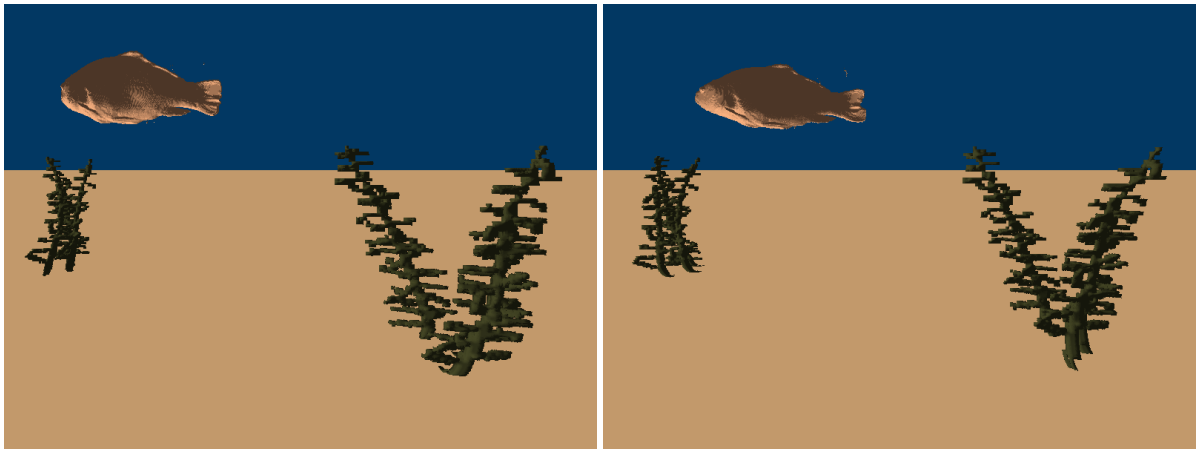
(a) No deformation.



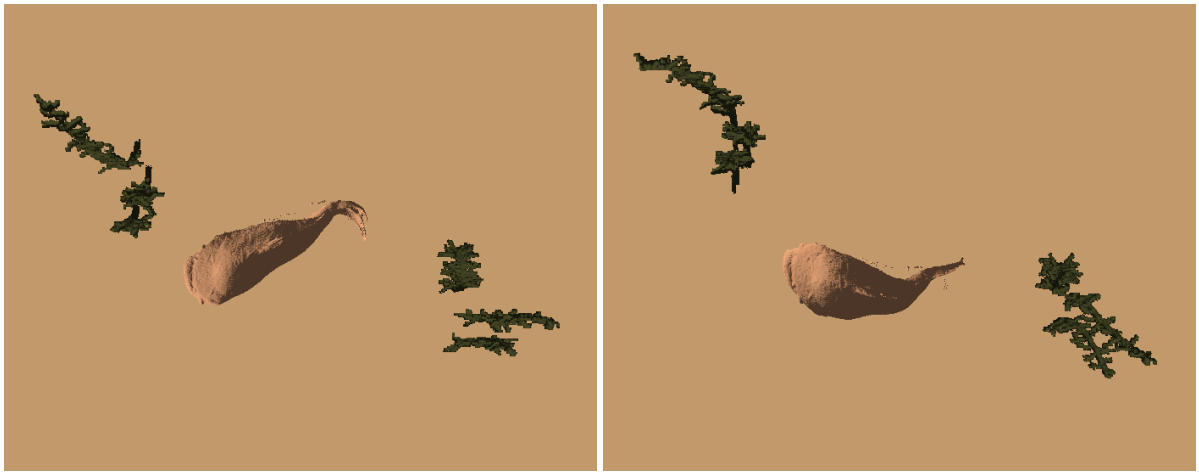
(b) Tangents $\{(3,0,1)^T,(3,0,2)^T\}$, $\{(-2,0,0)^T,(2,0,3)^T\}$ and $\{(0,0,3)^T,(-3,0,4)^T\}$ respectively.

(c) Tangents $\{(13,0,7)^T,(14,0,-4)^T\}$, $\{(-40,0,0)^T,(20,0,-13)^T\}$ and $\{(0,0,-3)^T,(-54,0,-10)^T\}$ respectively.

Figure 6: Models of an orange (left), a pig (middle) and a bunny (right) with various spline tangents and their resulting deformations.



(a) Frames viewed from the side.



(b) Frames viewed from the top.

Figure 7: Two frames of an underwater scene with smoothly animated water plants and a fish, shown from different viewing angles.