Interacting with Volume Data: Deformations using Forward Projection

Simon Walton, Mark Jones
Department of Computer Science
Swansea University
Wales, UK
cssimon@swan.ac.uk, m.w.jones@swan.ac.uk

Abstract

We present a rendering algorithm for the forward projection of volume deformations. Spatial deformations are modeled by allowing the user to define curve skeletons inside the target object. Our rendering method utilises modern consumer graphics hardware and is integrated into an easy to use interface. In addition, we give an analysis of spatial transfer functions computed on the GPU, and give a solution to the problem of cracks appearing in image space when samples are pulled apart.

1 Introduction

This paper investigates the problem of providing an interactive user interface in order to allow real-time deformation of volume datasets for the purpose of improving user interaction during biomedical visualisation. Emphasis is placed upon the design of algorithms and GPU implementations to give the desired interactivity, and the main goal is to provide a user interface that allows the user to just grab and manipulate parts of the volume domain they are working with.

Many volume deformation algorithms have been devised that allow for manipulation of the data obtained from CT/MRI scans. However, these algorithms often suffer from a lack of interactivity (caused by computational complexity), or from an unintuitive methodology presented to the user; the user should not be expected to predict the effects of pulling one voxel on the rest of the dataset.

We present a fully interactive deformation tool based on the highly-intuitive Volume Wires methodology which was introduced with a backward-mapping algorithm [14]. The new GPU forward-projection rendering algorithm presented here gives higher interactivity, and provides an effective image-space solution to the general problem of cracks appearing during forward projection.

2 Background and Related Work

Early research demonstrated that forward-projection methods for volume rendering were more suited to interactive viewing of volume datasets. Splatting was introduced by Westover [16] as a forward-projection solution to volume rendering, whereby voxels are projected towards the screen and the contribution of the voxel for each pixel is based upon its pre-computed footprint. Point-based rendering has seen recent growth due to systems such as QSplat [11] which has an efficient software renderer and Point Shop 3D [17] which allows for real-time manipulation of the point data. Methods for rendering point-based datasets need to account for the gaps between neighbouring samples (see Figure 3(d)) to construct a continuous surface in image space – e.g. surface splatting [18] renders object-space ellipses, with the overlap between ellipses closing the holes between samples.

Point-based rendering has been implemented on the GPU using many different data encoding and manipulation techniques [2, 1]. Modern GPUs are very well suited to dealing with the large amount of vertices and repetitive, independent processing of these vertices. Most GPU point-based techniques use a two or three-pass approach to correctly blend overlapping splats [3], as current cards do not offer α-buffer functionality. Multipass approaches sometimes write intermediate data to a texture and then use this texture data in a subsequent fragment pass to complete any additional calculations. With a *visibility splatting* approach [6], the first pass generates a correct z-buffer for the current view. Then, in order to blend visible overlapping splats, the second pass shifts the viewpoint back slightly such that the only splats now passing the z-test are those at the front of the view for each pixel. This technique has the added advantage of allowing the GPU to cull any fragments failing the depth test before entering the fragment program.

Our contribution to the area is a forward-mapped approach to volume deformation. We design and implement a GPU-based forward-projection rendering algorithm that

evaluates the spatial mapping of voxels on the GPU, and provides effective image-space splat correction to solve the problem of cracks appearing as voxels are pulled apart.

3 Concepts and Terminology

In the context of deformation for discretely sampled objects, a **forward mapping** function $\Phi : \mathbb{E}^3 \rightarrow \mathbb{E}^3$ defines, for each $p \in \mathbb{E}^3$, the spatial transfer, Φ, of point p. Typically these functions are evaluated at render time by applying $\Phi(p) = p'$ and using p' in place of p in subsequent render stages. Forward-mapping deformation methods can be utilized for forward-projection based renderers (as in [5]), by applying Φ to each voxel and rendering the new position using a forward-projection volume or point-based rendering model such as splatting.

The problem with forward mapping is that adjacent points p may not map to adjacent pixels p' in the framebuffer, which leads to visual cracks. Other methods also introduce artefacts where bounding volumes around objects of interest move independently and therefore open up linear cracks. We remove these artifacts using a fragment program running on the GPU.

4 System Overview

Our system uses the Volume Wires deformation methodology [14], allowing the user to define *wires* to manipulate the dataset. The *base wire* acts as a curve skeleton for the object, and the associated *control wire* defines the deformation of this skeleton.

Section 4.1 discusses our method of encoding the deformation, while section 4.2 discusses how this encoding is used at render time. The remaining parts of this section discuss the user interface, the effects that can be achieved, and details of our forward-projection algorithm for interactively viewing the deformation.

4.1 The Mapping Field

The mapping field is a volumetric dataset (γ, ϵ) where γ corresponds to the voxel's nearest base wire and ϵ corresponds to the voxel's nearest offset along that wire (see Figure 2).[1] The field is created using a distance propagation technique, which requires an initialisation of distances to be propagated. Voxels close to a wire in the scene are initialised with the closest distance d to that wire, along with the (γ, ϵ) pair representing the wire reference and offset.

[1] Implementation note: for efficiency, the wires in the system are sampled as a finite set of points, therefore ϵ maps the voxel to a point index rather than a particular floating-point t-value to be fed into the parametric equation of the wire.

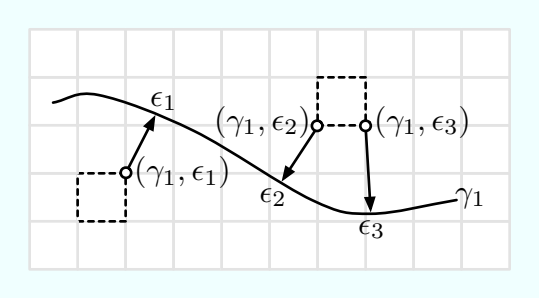

Figure 2. The mapping field (2D example)

To begin, the distances are initialised to ∞. Each base wire is then incrementally followed through the field (voxelized). For each voxel cube (bounded by eight voxels) that the wire traverses, the voxel's current d-value is compared to the distance between the current wire point and the voxel. If the new distance value is less, then the voxel is updated with the new wire reference γ and offset ϵ, and finally the new distance.

These distances and (γ, ϵ) attributes are then propagated using the EVDT vector transform [8] (giving higher accuracy than a distance transform) to create a vector field with additional attributes (γ, ϵ) at each voxel. Once completed, the intermediate vector values are discarded from the field as they are not required by the system in subsequent stages, thus saving a great deal of memory.

4.2 Mapping Process

In order to render the current deformation, a mapping must be established between dataset space (where the base wires are defined) and world space (where the associated control wires are defined). To forward-map the deformation, each voxel is essentially associated with its nearest wire point. The difference in both the world position and wire tangent determine where the voxel is translated to its new position in world space, and how its normal is perturbed.

The mapping process also allows for effects such as rotation (twisting effect) and scaling (warping effect) along the trajectory of the wire, and we denote such operations below as having a matrix E which performs the operation for a particular wire offset. If $T(p \in \mathbb{E}^3)$ gives a translation matrix for point p, we now express the composite mapping matrix for the nearest control wire point $p_{control}$ and nearest base wire point p_{base} as :

$$M_{base \rightarrow control} = T(p_{control}) \cdot R_c \cdot E \cdot R_b \cdot T(p_{base})$$

that is, a mapping taking a point p and computing the geometrical transformation of the point based on its nearest wire offset. The effect matrix E is composed of $R_z(r)$

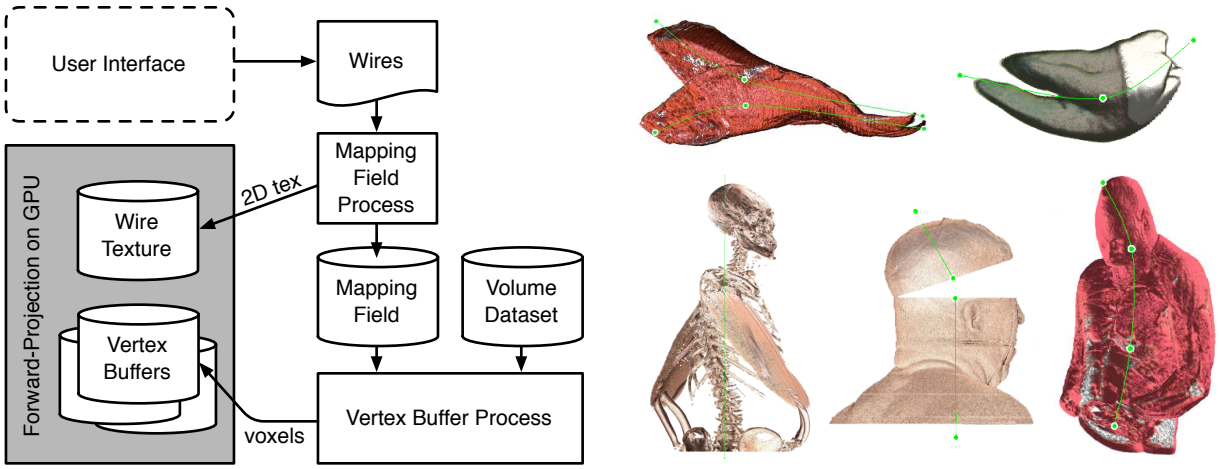

Figure 1. Left : An overview of our forward-projection system for deformation. Right : some examples of interaction with a variety of datasets (carp, tooth, visible human), all manipulated and viewed in real time.

which performs a rotation by r around Z, and $S_{xy}(s)$ which performs scaling of the voxel in the XY plane by s.

Using a point-based approach rather than converting to a polygonal mesh [7] gives visually superior images due to correct blending of overlapping splats, and allows for dynamic thresholding and transfer function modification. An appropriate splat size is determined adaptively based on the viewing parameters and the density of voxels.

The process of building the mapping field needs to be completed only when the base wires in the system are defined or modified. Once the field is constructed, the user is free to deform the volume object(s) by pulling the control points of the control wires and viewing the volume object deformation in real time (>**5 frames per second**).

The interface additionally allows for a context menu to be invoked on the control points of a wire and a rotation and/or scaling operation to be applied easily with the mousewheel or arrow keys (a rotation/twist effect has been applied to the bottom left image in Figure 1(right)). The context menu also offers a cutting tool. When invoked on a control point, the cutting tool splits the wire into two separate wires. This allows for accurate splitting effects to be achieved by moving the two wires apart at the point of the cut. A dataset split is shown in Figure 1(right) bottom middle.

4.3 Forward-Projection Algorithm

The wires are discretised into 1024 points and encoded into a 2D texture with a 32-bit unclamped internal format. Each wire occupies one row in the texture, and each column texel represents a wire point encoded into the $< R, G, B >$ values. The rotation and scaling effects defined on the wire are encoded into a 2D texture in a similar manner to the wires and sent to the GPU as additional input to the mapping operation – the scalar values representing rotation angle and scaling factor are encoded into the $< R, G >$ components of the texture. Since Catmull-Rom splines are evaluated in $[0, 1]$ between control points, the discretised values in-between control points can be linearly interpolated, producing effects that smoothly reduce in magnitude further away from the point. The encoding of effects into the wire texture means that the user can apply these twisting and scaling effects in real-time.

For each non-background voxel in the dataset, a vertex is defined with associated attributes and uploaded to the graphics card using vertex buffer objects [9]. Each VBO stores an $< R, G, B >$ float triple. `glBindBuffer` is used to instantiate 2 VBOs – the position and colour buffers, which are then bound to the position and colour binding semantics in the vertex shader. The below table shows our VBO arrangement.

Attribute	**Type**	**Buffer Used**
Voxel XYZ position in dataset space	3 floats	Vertex Buffer
Voxel scalar value	1 float	Colour Buffer
(γ, ϵ) pair for the voxel	2 floats	

The $< R, G >$ color buffer values for each voxel are obtained from the mapping field, which gives the (γ, ϵ) parameters to the forward-mapping function to be computed by the vertex shader. The colour buffer is updated with the new attributes when the mapping field is created or mod-

ified by obtaining a direct pointer to the VBO data using `glMapBuffer`.

Due to the dynamic nature of the system (modifications of the voxel positions), building an octree to determine a guaranteed front to back (or vice versa) traversal per frame is not feasible. Similar to [6], we use a two-pass visibility splatting approach. In the first pass, the depth test is enabled, depth writing is switched on, and the color buffer switched off. For the second pass, depth writing is switched off (but depth comparison left on), and the viewpoint shifted back by an arbitrarily small amount. The depth buffer from the first pass is now used to blend the splats lying only closest to the viewer – that is, splats surviving the shifted depth buffer.

At each vertex stage, the (γ, ϵ) pair is pulled from the R and G components and used in a vertex texture fetch on the encoded wire texture to calculate the forward-mapping of the deformation for the current voxel. The voxel is now translated to its new position via the forward-mapping operation (detailed in section 4.2). *Note: The vertex texture fetch functionality is only available on ShaderModel 3 capable hardware, and typically introduces a small amount of latency into the pipeline. Care is taken to hide this latency by performing large parts of the forward-mapping process as soon as the required data is available.*

For fragments surviving the visibility cull in pass two, the vertex shader computes the color of the voxel using a 1D transfer function texture, looking up the value encoded in the B component of the color buffer. The voxel's normal (obtained in the fragment shader from the volume dataset 3D texture) is additionally corrected according to the deformation using a similar transformation based on the wire data. Finally, Phong shading is applied to the voxel.

4.4 Adaptive Refinement

Depending on the size of the dataset and number of active voxels, we may wish to choose only a subset of the voxels to send to the graphics card due to memory limitations (either CPU or GPU memory) or the power of the GPU. It is important for large datasets (such as the visible human dataset) that the system dynamically adjusts the workload to ensure an acceptable frame rate for the user. Performing the mapping operation and lighting calculations on millions of vertices can be costly, and the number of further fragments produced need additional processing. Our system continually measures the frame rate, and adjusts the number of samples sent to the card when this rate drops below a particular threshold (below 3 FPS currently). The system therefore requires a mechanism to allow for different densities of sample points to be used without a noticeable delay for the user.

OpenGL allows for arbitrary striding of vertex array data [12] - but only on a 1D basis which would result in artefacts in the volume. Our solution is to maintain two sets of VBOs in CPU memory; a sparse set of voxels is used for very quick feedback, and the full set is used to refine the image once the user has finished interacting. The number of voxels chosen for each set can be automatically decided by the system based on the capabilities of the graphics card and the size of the dataset. If the graphics card's memory has sufficient space, both sets can be kept in GPU memory. If not, the VBO data must be swapped each time with a small performance penalty.

4.5 Cracks & Splat Shape

During volume deformation, two types of crack artefact may be introduced – those that correspond to a joint in the curve skeleton, and those due to the expansion of a deformed region. The latter is investigated in the next section. The former problem has been encountered previously [4] and one solution is that of using mid-plane geometry [13], which may not produce a smooth, continuous connection.

The use of the curve skeletons in this system ensures continuity at joints, and removes the need for joints to be treated as special cases. Our system allows for stretching and compression of the volume object along the wire as a by-product of the deformation process. With a finite set of voxels, cracks will occur as the voxels are pulled apart along the wire trajectory. This effect is especially apparent along the outer edges of a bent object where the curvature of the wire is high (Figure 3(d)).

Point-based rendering algorithms typically do not support deformations and can therefore assume a reasonable sampling density. Where such a density does not exist, the image-space splat shape and/or sizes are modified to close any holes in the final image. We therefore introduce a new method to solve the crack problem in the next section.

4.6 Splat Fragment Correction

To correct the shape of the splat (the voxel's image-space projection), we use the `point_sprite_nv` OpenGL extension to obtain an image-space parameterisation of each voxel. The point size is increased to give a large image-space area per splat. The size of the point chosen is a trade-off between image quality (bigger points give a larger space for splat correction) and speed (bigger points produce a larger number of fragments). In our research, we have found that a point size of 15 gives the most acceptable results for most datasets.

As a control wire is compressed and expanded, the distance between consecutive points on the control wire is compared to the points on the corresponding base wire to obtain a scaling value s, which is stored in the α compo-

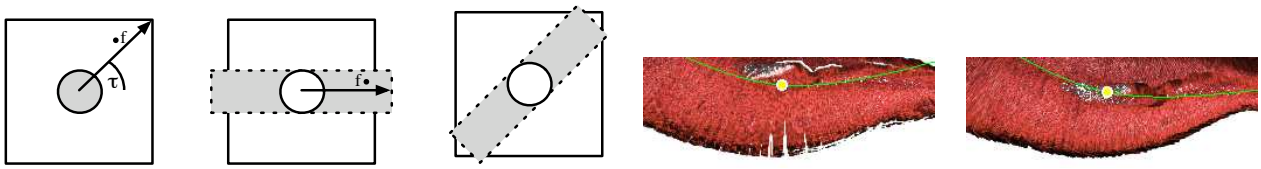

Figure 3. Correcting the splat shape along wire trajectory

nent of the wire texture for each point. We derive s for each wire point as the distance between the current control wire point and its previous point over the distance between the current base wire point and its previous point. The vertex shader takes this value along with the image-space wire tangent vector and computes rotation coefficients to send to the fragment shader.

The fragment shader is given a texture coordinate for the point sprite fragment, and tests if the fragment falls within the target area. Figure 3(a) shows the image-space point sprite with image-space wire normal vector (shown as a solid arrow) and currently processed fragment f. Figure 3(b) shows the fragment rotated by -τ using the coefficients computed in the vertex shader to lie within the hit-box. The test for fragment retention is now reduced to a simple inside-rectangle test. The splat size is used to determine the y-extent of the test. In this case, fragment f survives the test and continues to be processed. Any fragments failing the test are destroyed.

Our choice of having a two-pass splatting algorithm does force the vertex shader to compute the deformation twice between passes, as we cannot save the vertex state before rasterization. We found that due to the heavy use of fragment operations on point sprites, a two-pass approach is more optimal than running all fragments through the fragment program. Additionally, it provides blending of splats, giving a higher-quality image.

Using this algorithm corrects the splat shape problem as seen in Figure 3(d), giving the resulting image shown in Figure 3(e), where the cracks have been corrected.

5 Discussion & Conclusion

The system has been implemented on GNU/Linux x86 with a Geforce 7800GT using C++, and Trolltech's QT Toolkit (version 4) for the GUI and user input. OpenGL is used to communicate with the graphics hardware and Cg is used to write the shaders.

Table 1 gives the amount of time to generate the mapping field for different datasets, along with the resulting size of the field and additionally the time to modify the VBO vertex/color buffer.

Table 2 gives the average FPS (frames per second) for a series of deformations. We measure the FPS while actively deforming the dataset – therefore the timings include the time taken to regenerate the wire texture at a precision of 1024 discrete samples.

Dataset	Field size	Generation time
Visman Torso	$77 \times 47 \times 100$	0.1s
	$155 \times 95 \times 200$	0.76s
	$310 \times 190 \times 400$	6.16s
CT Carp	$64 \times 64 \times 128$	0.14s
	$128 \times 128 \times 256$	1.10s
	$256 \times 256 \times 512$	8.93s

Table 1. Mapping field generation timings

Dataset	# Active voxels	Average FPS
Visman Torso	716,139	8.76
CT Carp	641,022	8.93
Tooth	52,536	30.15

Table 2. Frame rates during interaction

Figure 4 shows one frame from an interactive session investigating the Visible Human male dataset. As part of the tool, an interactive segmentation has associated the arm voxels with their base wires, which now allows them to be repositioned interactively, thus providing the user with unobstructed view. Figure 5 shows a split of the Visible Human male dataset being performed in real-time by simply splitting the wire using the cutting tool, and moving the split wires apart.

The methods introduced here provide a rendering algorithm for interactive volume deformation based on the forward-mapping of voxels computed on the GPU. Our interactive algorithm utilises a forward-projection technique with image-space splat correction to avoid cracks appearing between voxels. We have given a discussion on forward mappings, and a method to compute them interactively on the GPU.

In this paper we have concentrated on the algorithmic details to provide such interaction. Future work would be needed to quantify its usefulness in real applications, but the example images demonstrate the ease of use and the useful-

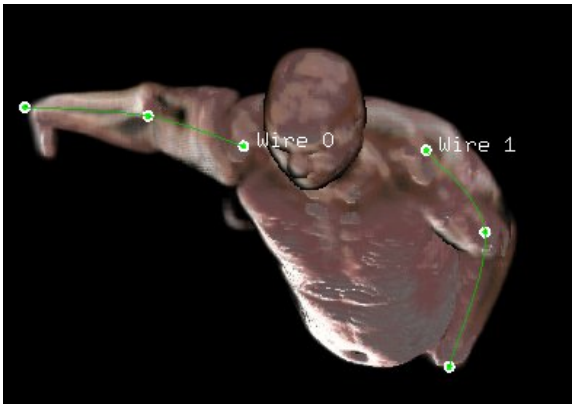

Figure 4. Segmentation-aware deformation of the Visible Human

ness of being able to interact with volume data in real time. Users are able to manipulate the wires and gain immediate feedback of their deformation.

Acknowledgement

Research supported by EPSRC grant GR / S44198. The authors would like to acknowledge the National Library of Medicine's Visible Human Project, and Stefan Roettger's Volume Library [10] for providing an excellent resource of volume datasets for research.

References

[1] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proceedings of the Eurographics Symposium on Point-Based Graphics*, pages 17–24, June 2005.

[2] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, pages 335–343, Washington, DC, USA, 2003. IEEE Computer Society.

[3] L. Coconu and H.-C. Hege. Hardware-accelerated point-based rendering of complex scenes. In S. Gibson and P. Debevec, editors, *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 43–52, Pisa, Italy, June 2002.

[4] N. Gagvani and D. Silver. Animating volumetric models. *Graphical Models*, 63(6):443–458, 2001.

[5] S. Gibson. 3D chainmail: a fast algorithm for deforming volumetric objects. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 149–154, April 1997.

[6] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[7] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

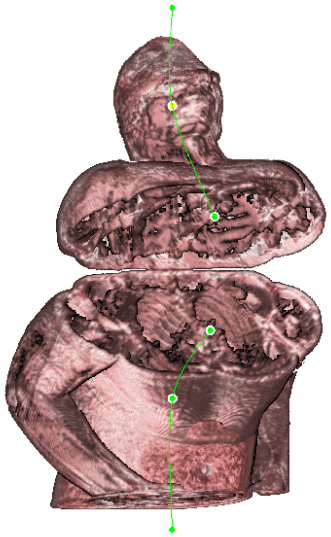

Figure 5. A real-time split of the Visible Human torso

[8] J. C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graph. Models Image Process.*, 54(6):526–535, 1992.

[9] NVIDIA Corporation. *Using Vertex Buffer Objects*, 2003. Whitepaper, ftp://download.nvidia.com/developer/Papers/.

[10] S. Roettger. The volume library, 2006. http://www9.cs.fau.de/Persons/Roettger/library/.

[11] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *SIGGRAPH 2000)*, pages 343–352, August 2000.

[12] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 2.0)*. Silicon Graphics, Inc., 2004.

[13] V. Singh, D. Silver, and N. Cornea. Real-time volume manipulation. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 45–52, 2003.

[14] S. Walton and M. Jones. Volume wires : A framework for empirical nonlinear deformation of volumetric datasets. *Journal of WSCG*, 13:81–88, 2006.

[15] S. Walton and M. W. Jones. Deforming volumes interactively using control skeletons. Submitted to Eurographics 2007.

[16] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, Aug. 1990.

[17] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3D: An interactive system for point-based surface editing. In S. Spencer, editor, *SIGGRAPH 2002*, volume 21, 3 of *ACM Transactions on Graphics*, pages 322–329, New York, July 2002. ACM Press.

[18] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *SIGGRAPH 2001*, pages 371–378, New York, NY, USA, 2001. ACM Press.