Graphics Hardware (2007)
Timo Aila and Mark Segal (Editors)

Programmable Shaders for Deformation Rendering

Carlos D. Correa and Deborah Silver

Rutgers, The State University of New Jersey

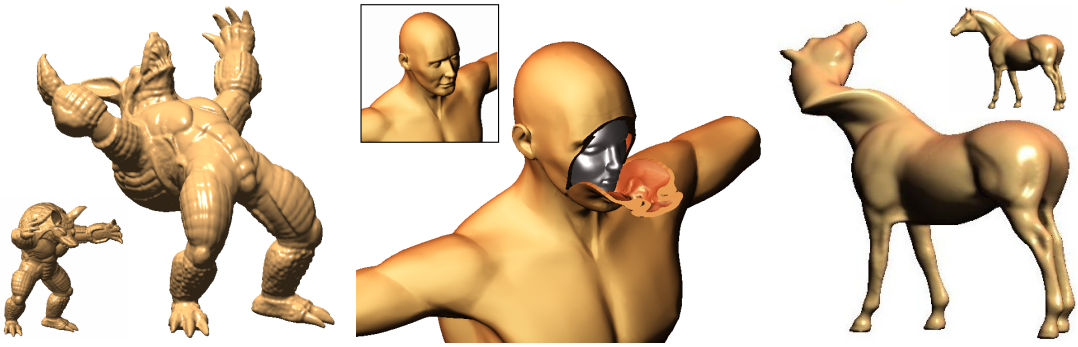


Figure 1: *Example cuts and deformations on geometric models. From left to right: (a) Armadillo with bending deformation (172K triangles) (b) Torso (25,528 triangles) with peeled skin and interior Mask model (10,213 triangles) (c) Horse with twist deformation (97K triangles)*

Abstract

In this paper, we present a method for rendering deformations as part of the programmable shader pipeline of contemporary Graphical Processing Units. In our method, we allow general deformations including cuts. Previous approaches to deformation place the role of the GPU as a general purpose processor for computing vertex displacement. With the advent of vertex texture fetch in current GPUs, a number of approaches have been proposed to integrate deformation into the rendering pipeline. However, the rendering of cuts cannot be easily programmed into a vertex shader, due to the inability to change the topology of the mesh. Furthermore, rendering smooth deformed surfaces requires a fine tessellation of the mesh, in order to prevent self-intersection and meshing artifacts for large deformations. In our approach, we overcome these problems by considering deformation as a part of the pixel shader, where transformation is performed on a per-pixel basis. We demonstrate how this approach can be efficiently implemented using contemporary graphics hardware to obtain high-quality rendering of deformation at interactive rates.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object ModelingGeometric Transformations; I.3.6 [Computer Graphics]: Methodology and TechniquesInteraction techniques;

1. Introduction

In this paper, we present a novel rendering methodology which produces high-quality deformations including cuts and twists, using programmable graphics hardware. Unlike previous approaches to hardware-assisted deformation, which place the deformation at the vertex shader, we place deformation in the pixel shader. This has certain advantages over per-vertex deformation. First, generic deformations can be defined in *deformation textures*, which sample 3D displacements in a 2D or 3D texture. Vertex texture fetch has only been supported in the latest GPUs, and have limited filtering modes. However, pixel texture fetch is still considerably faster, and provides support for multiple filtering modes, such as bi-linear, tri-linear and anisotropic. Second, deformation on image-space, as opposed to object space, provides smoother results, as it is not limited by the resolution of the mesh. In our approach, we deform the object as part of the rendering pipeline. This is different from other traditional deformation methods, where deformation is considered as a modeling problem and a new mesh is required.

GH 2007, San Diego, California, August 04 - 05, 2007
© 2007 ACM 978-1-59593-625-7/07/0008 $ 5.00

Deformation as a rendering process does not produce a resulting mesh, but generates images of a deformed object in real-time and with very high quality. This has a wide number of applications in real-time illustration, animation, gaming and generation of special effects.

In this paper, we propose a generalized notion of deformation shaders, where we exploit the programmability of graphics hardware to render all kinds of deformations, including cuts and breaks. These are more difficult to model using vertex processing, as topology changes are required (i.e., remeshing). However, we show that these can be realized in a pixel shader, by considering a special kind of *deformation textures* which encode discontinuity information. Because we realize deformation as a pixel shader, it is considered as an inverse space warping problem rather than a vertex transformation problem. As a general approach, we use raycasting to find intersections of viewing rays with the deformed surface. This paradigm was introduced earlier for raytracing of deformed objects [Bar86], and we show how it can be efficiently implemented using programmable graphics hardware. Further, we solve a number of additional challenges when incorporating cuts.

2. Related Work

Deformations have been widely used in computer graphics to generate a number of effects, and they have applications in animation, shape modeling and simulation. The literature in deformation is extensive, for which there are a number of surveys [NMK*05, BS07]. Mesh cutting is another aspect of deformation, where the mesh topology is modified to simulate breaks and incisions [BSM*02]. Work closely related to this paper are GPU-based deformation techniques and rendering of implicit surfaces.

GPU Accelerated Mesh Deformation. Most deformation approaches transform the vertices of a mesh directly, via procedural transformations [Bar84], via a proxy geometry [SP86], or via a physically-based simulation [NMK*05]. Recently, physically-based deformation approaches have used graphics hardware for solving partial differential equations required for mass-spring or finite element models [KW03, GWL*03]. In this case, rendering is seen as a different stage to the deformation process. This approach has been extended to the rendering of cuts, especially for surgery simulation [MS05]. Others have applied deformation during the rendering process, in a *vertex shader*. The deformation is procedurally defined [d'E04], defined as a deformation texture [JP02] or a displacement map [SKE05], or as a combination of basis functions [MBK06]. The ability to define deformations as textures is possible thanks to vertex-texture capabilities of recent GPU's. A problem with vertex-shader-based deformation is the reliance on a smooth tessellation of the input mesh for high-quality rendering and the difficulty to model topology changes, required for cuts. As an alternative, deformation can be done in a *pixel shader*. Transforming the shape of an object in image space has been successfully implemented for displacement mapping. Displacement mapping [Coo84] allows the rendering of geometric detail on a base surface. Recent methods use GPU-based ray casting [OBM00, HEGD04, WTL*04, PO06]. However, many of these techniques do not extend easily to general deformation or to large cuts, because the displacement map is usually independent from the input geometry and because it is assumed to be contained within a small volume (shell) surrounding the surface. A displacement was proposed in [CSC06b, CSC06a], where a volumetric object is deformed and cut at interactive rates. Because the volume is represented as a stack of view-aligned slices, this approach is prone to aliasing and undersampling. In this paper, we generalize the notion of displacement map for deformation, including the ability to render cuts in polygonal models in an efficient manner, without the requirement of explicitly changing the topology of the mesh.

Implicit Representations. Implicit surfaces have been used widely in computer graphics as an object representation, with applications in modeling, deformation and animation [Blo97]. One of the challenges with implicit deformations is finding intersections efficiently for interactive rendering or accelerated ray tracing. A number of approaches were proposed very early [Har96, KB89, SP91], for both undeformed and deformed implicit surfaces. Recently, Hadwiger et al. [HSS*05] showed how these methods can be extended to the rendering of arbitrary meshes using the distance field as an implicit representation. In this paper, we exploit the programmability of pixel shaders to represent mesh deformation as a rendering of a deformed distance field. Unlike previous approaches, our method can be extended to the rendering of cuts, which may introduce sharp corners, difficult to render accurately via raycasting. We show how efficient ray casting algorithms can be developed for artifacts-free rendering of deformation-cuts.

3. Overview

Our pipeline for deformation is based on the raytracing of implicit deformable surfaces. Because of the exactness of representation, we choose signed distance fields as a representation. Later, in Section 6.1, we show how this requirement can be adjusted to improve GPU memory efficiency. An implicit surface is defined as an isosurface of a function $f : R^3 \mapsto R$. As a convention, a surface of interest S can be defined implicitly as the set of points $\mathbf{p}$ such that $f(\mathbf{p}) = 0$. Also, as a convention, f is positive inside the object and negative outside. A deformed implicit surface S' is defined as the zero-set $f'(\mathbf{p}) = 0$. Let $T : R^3 \mapsto R^3$ by a spatial transformation, so that a point $\mathbf{p}$ is transformed into $\mathbf{p}' = T(\mathbf{p})$ and its inverse T^{-1} such that $\mathbf{p} = T^{-1}(\mathbf{p}')$. In this paper, we define T^{-1} as a 3D inverse displacement, i.e., $\mathbf{p} = \mathbf{p}' + \mathbf{D}(\mathbf{p}')$, with $\mathbf{D} : R^3 \mapsto R^3$, a displacement field. The deformed implicit

surface can be found as the set of points $\mathbf{p}'$ such that :

$$f'(\mathbf{p}') = f(T^{-1}(\mathbf{p}')) = f(\mathbf{p}' + \mathbf{D}(\mathbf{p}')) = 0 \qquad (1)$$

Since we intend to apply this method to surfaces defined as meshes, our rendering pipeline requires a stage where a signed distance field is obtained from the mesh. However, we do not require a signed distance field for the entire mesh, but rather only the subset of the surface which is to be deformed. Thanks to a number of GPU-based techniques for the generation of distance fields [SPG03, SGGM06], it can be done interactively.

First, the user selects a region where deformation is to be applied using a bounding box. Let $S_D \subseteq S$ be the part of the surface contained in that region, which is to be deformed. The user selects a 3D displacement texture from a pool of pre-defined deformations, and applies it to the surface, much in the way color or displacement textures are applied in current graphics pipelines. Because we know the maximum attainable displacement for a given deformation texture, we can bound the space occupied by the deformed surface within a bounding box $B(S'_D)$, where S'_D is the deformed surface. Then, the rendering can be done in two stages. First, we render the undeformed part of the object using traditional shaders, enabling depth culling of the region defined by $B(S'_D)$, so that the deformed region is not rendered twice. In the second stage, we render the bounding box $B(S'_D)$ using a deformation shader. The deformation shader is a shader program that enables the rendering of deformed implicit surfaces in image space rather than object space. Finally, once both are rendered, we blend the color near the seams so that both parts (the one using an explicit surface and the one using an implicit surface), are integrated without artifacts. This is possible as long as deformation vanishes as it gets near the boundaries. This could be enforced in all boundaries of the deformation cube, but, in practice, this is only enforced in certain faces of the cube, depending on the deformation. Twisting, for instance, is constrained to be zero at all the faces except the top of the deformation box. This means that it should be placed so that the top face do not intersect the object.

The deformation shader then consists of a ray traversal procedure. Each fragment generates a ray in the view direction. For each sample point we perform the following steps, as shown in Fig.(2):

1. **Warping.** Each sample along the ray $\mathbf{p}'$ is *inversely warped* by sampling the deformation texture, and using the values as a 3D displacement. That is:

$$\mathbf{p} = \mathbf{p}' + M(\mathbf{D}(M^{-1}(\mathbf{p}'))) \qquad (2)$$

   where $\mathbf{D}$ is a displacement field (stored as a deformation texture), and M is a coordinate transformation between texture space and object space.
2. **Sampling.** The warped coordinate $\mathbf{p}$ is used to sample the implicit representation f.

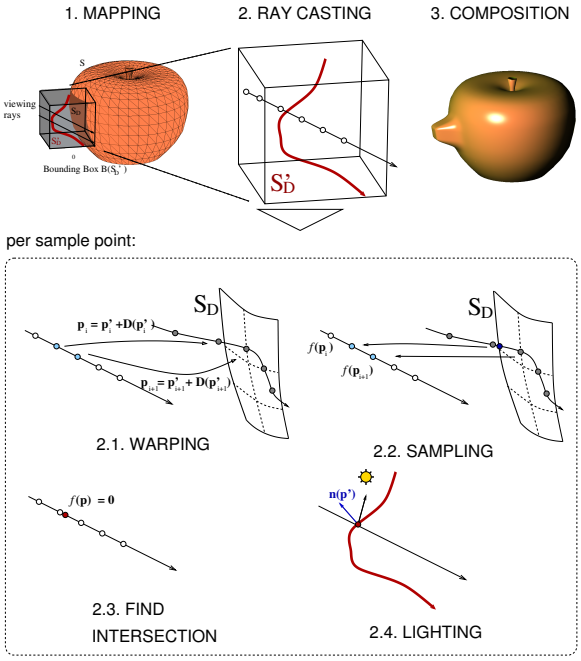


Figure 2: *Deformation of an apple. S, the original surface is rendered with a traditional shader. A 3D displacement/deformation texture is chosen and added to the object. The bounding box $B(S'_0)$ is rendered as an implicit representation using ray casting. The result of both shaders are composited together.*

3. **Find Intersection.** The ray traversal is stopped whenever an intersection is found, as the zero-set $f(\mathbf{p}) = 0$. A number of approaches can be applied for the determination of a stopping criteria and ray traversal, as discussed in section 3.1
4. **Lighting.** Finally, when an intersection is found, color and lighting attributes are obtained by estimating the normals to the deformed surface at the intersection point.

3.1. Ray traversal and Intersections

In order to find the closest intersection of a ray with the surface, this stage traverses the ray at sampling positions:

$$\mathbf{p}'_i = \mathbf{p}_0 + t_i\mathbf{v} \qquad (3)$$

for $t_i \in (0, t_{end})$ and $\mathbf{p}_0$ the entry point of the ray $\mathbf{v}$ into the bounding box $B(S'_0)$. This linear search has been used for interactive rendering of displacement maps. For more accurate intersections, linear search is usually accompanied with a binary intersection refinement, such as in [HSS*05], and an adaptive sampling mechanism. For speed up, linear and binary search gives good results for smooth surfaces. Surfaces with narrow details, however, results in artifacts due to missing an intersection. In the context of deformation, there are two additional challenges: the introduction of large deformations, and the introduction of sharp cuts, both of which

can introduce narrow details into the new deformed surface. These problems are discussed in the following sections.

3.1.1. Adaptive Sampling

One of the challenges with the rendering of implicit surfaces is accurately finding the first intersection with the surface. With linear search, regions of high spatial frequency might be skipped, resulting in artifacts near edges. A number of solutions have been proposed. Previous approaches aiming toward ray tracing require "guaranteed" intersection-finding algorithms, based on Lipschitz constants [KB89]. Hadwiger et al. use adaptive sampling based on a user-controlled threshold [HSS*05]. In our paper, we are interested in the cases where adaptive sampling is needed in the case of deformation. For this purpose, and without loss of generality, we assume that the original undeformed surface can be rendered robustly using one of the above techniques. With deformation, additional conditions must be met so that the deformed surface is rendered properly. For instance, a long narrow pull or a large twisting, may increase the required sampling of the deformed surface. Let us define a ray in deformed space $\mathbf{p}_0 + t\mathbf{v}$, where $\mathbf{v}$ is the view direction and $\mathbf{p}_0$ is the ray entry point. This ray corresponds to a 3D curve $\mathbf{R}(s)$ in undeformed space. We can consider $s = T^{-1}(t)$, for an inverse transformation T^{-1}, such that $\mathbf{R}(s) = T^{-1}(\mathbf{p}_0 + t\mathbf{v})$. To avoid missing intersections in the case of a sharp or large deformations, we ensure that samples in the deformed space correspond to uniform samples along the curve in the undeformed space. Taking the derivative of s with respect to t yields:

$$\frac{ds}{dt} = |\mathbf{J}_{T^{-1}}\mathbf{v}|$$

where $\mathbf{J}_{T^{-1}}$ is the Jacobian of the transformation T^{-1}. Since T^{-1} is a displacement, $\mathbf{J}_{T^{-1}} = \mathbf{I} + \mathbf{J}_D$, where $\mathbf{I}$ is the identity matrix and $\mathbf{J}_D$ is the Jacobian of the displacement field. Then, assuming a constant sampling distance δs, an adaptive sampling (which we call Jacobian sampling) is obtained as:

$$\delta t_i = \frac{1}{|(\mathbf{I} + \mathbf{J}_D(t))\mathbf{v}|}\delta s \qquad (4)$$

and points along the ray direction can be found as $\mathbf{p}_{i+1} = \mathbf{p}_i + \delta t_i \mathbf{v}$. Fig.(3) shows an example of a narrow pull on the golf ball model. Because adaptive sampling can be costly, we allow the programmer to define whether to use a threshold-based sampling as in [HSS*05], Jacobian sampling or no adaptive sampling at all.

3.2. Warping

Warping of the sampled positions is done via inverse displacement using Eq.(2). Previous approaches to programmable deformation define the warping as a hard-coded statement within a vertex or pixel shader. This method, however, is difficult to generalize and some deformations, such as peels, require a complex procedural definition that may be

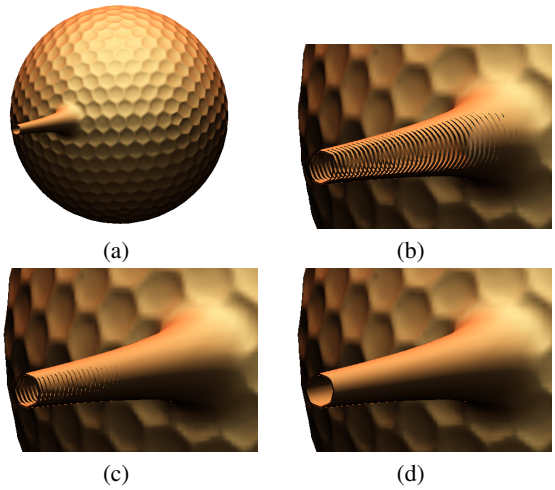


Figure 3: *(a) Narrow Pull over golf ball model (245K triangles). (b) Linear search with binary refinement results in missed intersections (c) Adaptive sampling based on threshold (thresh=0.04) cannot resolve all misses (d) Jacobian sampling finds the intersections properly.*

costly to compute. Instead, complex deformations are sampled into *deformation textures*. In Eq.(2), we allow the definition of a spatial transformation M, which represents a mapping between object space and displacement space. This type of mapping may be used to map the deformation in cylindrical coordinates, for example, to render "round" cuts. Another use is the interactive control of the spatial extents of the deformation, as an affine transformation $A(\mathbf{p}) + \mathbf{u}$, where A contains a rotation and scaling, and $\mathbf{u}$) is a translation. This allows the user to translate and rotate the deformation texture to simulate interactive manipulation of the deformation parameters, *without* the need to modify the deformation texture on the fly. For instance, translating a twist in the y direction, has the effect of increasing the strength of the twist. Similarly, a translation along the x direction of a peel deformation simulates the effects of progressive peeling (Fig.(4)).

3.3. Lighting

After an intersection point is found, color properties are obtained depending on the lighting parameters. For this purpose, we need to estimate the normal at that point. The normal can be reconstructed from the gradient of f' on the fly, or, for speed up, from a transformation applied to the original gradient of f. In traditional deformation, this can be encoded into the vertex shader by multiplying the original normal by the inverse transpose of the Jacobian of the *forward* transformation function T [Bar84]. In our approach, we require a transformation based on the inverse displacement. From vector calculus, we have that for an inverse transformation, $\mathbf{J}_T^{-\top}(\mathbf{p}) = \mathbf{J}_{T^{-1}}^{\top}(T(\mathbf{p}))$ [Dav67], where $\mathbf{J}_T$ is the Jacobian of a transformation T. For $T^{-1}(\mathbf{p}) = \mathbf{p} + \mathbf{D}(\mathbf{p})$, the Jacobian is

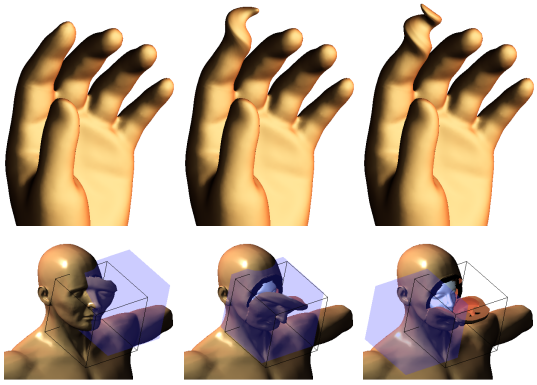


Figure 4: *Example of applying an affine transformation to the deformation texture. Top: A progressive twisting of a finger is obtained by applying a translation along the y axis. Bottom: A translation of the deformation space (blue box) along the z direction results in progressive peeling. This is encoded in the* mapping *stage of the deformation shader.*

defined as $\mathbf{J}_{T^{-1}}(\mathbf{p}) = \mathbf{I} + \mathbf{J}_D(\mathbf{p})$, where $\mathbf{J}_D(\mathbf{p})$ is the Jacobian of the displacement and $\mathbf{I}$ is the identity matrix. Therefore, normals can be obtained as:

$$\vec{\mathbf{n}}'(\mathbf{p}) = normalize\left((\mathbf{I}+\mathbf{J}_D(\mathbf{p}))^{\top}\vec{\mathbf{n}}(\mathbf{p}+\mathbf{D}(\mathbf{p}))\right) \quad (5)$$

where $\vec{\mathbf{n}}'(\mathbf{p})$ is the normal to the deformed surface at point $\mathbf{p}$ and $\vec{\mathbf{n}}(\mathbf{p})$ is the normal to the undeformed surface at point $\mathbf{p}$. Note that this is a different expression from the usual normal estimation formula which has been derived from direct deformation. Eq.(5) applies for inverse warping.

4. Definition of Deformation Textures

Deformation Textures are defined as a sampling of an inverse displacement in a regular grid. There are a number of ways of obtaining a deformation texture: procedurally, as the result of inverting a forward transformation, via control points, or via a simulation. In this paper, our deformations are obtained from a procedure. For a twist deformation, for instance, a forward transformation is given by the expression: $\mathbf{D}_F(x,y,z) = (x\cos\alpha(z)+y\sin\alpha(z)-x, -x\sin\alpha(z)+y\cos\alpha(z)-y, 0))$. The inverse displacement is simply $\mathbf{D} = -\mathbf{D}_F$. For a bend deformation, the inverse transformation can also be found, as described in [Bar84]. Complex deformations, however, may be difficult to define procedurally. Further, finding a close form solution to the inverse may prove difficult. In this case, deformation can be obtained using a finite set of control points, which are deformed interactively by the user, and using inverse weighted interpolation in order to find the values at grid points [RM95]. Once a deformation texture is created, it can be re-used into a number of surface models and applications, much in the way of color or image textures. Further, deformation textures based on displacements are algebraically easy to combine. Addition of deformation textures can be used to add deformation "detail" to a deformation texture.

5. Cut shader

The above method is essentially for continuous deformations, where the displacement function $\mathbf{D}$ is defined everywhere in the domain of the deformation texture. Deformation cuts, such as incisions, create discontinuities into the surface, without removing material. Other cuts, such as the removal of parts (i.e., a CSG operation), can also be realized with this method. To implement a cut shader, a number of changes are made to the different stages of the deformation pipeline. First, we must encode the cut information into our deformation texture. Second, finding intersections is different, as the introduction of cuts implies a change in the continuity of the surface. Further, when simulating a solid model, a ray may intersect the object at the interior, on a new surface that appears due to the cut. And finally, because a new surface may appear, the normal information must be adapted to the shape of the cut geometry as well. These cases are described below.

The deformation texture, which contains a discrete sampling of $\mathbf{D}$ must be extended to contain discontinuity information. This is achieved by creating an alpha map $A(\mathbf{p})$ (scalar field), which tells whether a point is inside or outside of a cut region. As a convention, we assume that a point $\mathbf{p}$ is cut, if it lies in the region $A(\mathbf{p}) > 0$, or not, otherwise.

5.1. Ray Intersection with Deformation Cuts

The ray intersection process differs from the original shader in that a ray may not intersect the surface if it is cut, or it may intersect a new surface that appears at the interior of a solid object. The appropriate ray intersection method depends on whether the object is assumed to be hollow, solid, or a hollow thick shell. Although surfaces are modeled as thin shells, a deformation cut shader allows the rendering of any of these three types of objects, as selected by the user.

5.1.1. Rendering of Hollow Surfaces

In the simplest case, we may consider a surface as a thin shell. Cutting a surface reveals the empty space inside an object, and provides a view of the backside of parts of the object. In traditional deformation, cutting a surface implies a change in topology, which is difficult to do in a vertex shader. However, this can be done using our pixel-based deformation shaders, by modifying the ray intersection algorithm, as follows: Whenever an intersection $\mathbf{p}'$ is found, a test for visibility is performed, using the alpha map which encodes the shape of the cut. If $A(\mathbf{p}') < 0$, the point is not within a region of the cut and the intersection is found. Otherwise, the point is discarded and the ray traversal continues until another intersection with the surface is found, or the stopping criteria are met. This is shown in Fig.(5a). The intersection

point 1 is discarded and only 2 is used for rendering. Note that these intersections are on the interior or underside of the model. We can use the normals to render the interior in a different color. Because of false intersections, this approach may be slow. Another approach is to use the original ray intersection algorithm over a combined implicit representation of the surface minus the geometry of the cut (CSG operation). However, this approach may result in intersection misses for sharp cuts. With the approach described above, however, sharp cuts can be represented without artifacts.

5.1.2. Rendering of Solid Objects

If we assume that the object is solid, rendering a cut reveals the interior of the solid. To render this effect, a ray may intersect a new surface that appears due to the cut. A similar algorithm is used, except that when an intersection with the object is discarded (because it is within the volume of the cut), it may be possible that it still intersects the cut geometry. In such case, the algorithm continues traversing the ray, but this time it searches for intersections with the cut surface, which is found at points where $A(\mathbf{p}) = 0$. A valid intersection with the surface of the cut must be in the interior of the object, i.e., $f(\mathbf{p}) > 0$. If that is not the case, the algorithm continues in the search of intersections. This is depicted in Fig.(5b). Rendering a solid object also changes the lighting stage, since the reconstructed normal depends on whether the point found intersects the surface f or the alpha map A. The following test computes the correct normal:

$$\vec{\mathbf{n}}'(\mathbf{p}) = \begin{cases} \vec{\mathbf{n}}'_f(\mathbf{p}) & \mathbf{p} < \varepsilon \\ \nabla(A) & \text{otherwise} \end{cases} \tag{6}$$

where $\vec{\mathbf{n}}'_f$ is the normal found using Eq.(5), and $\nabla(A)$ is the gradient of the alpha map A.

5.1.3. Rendering of Thick Shells

Finally, another possibility is the case where we have hollow objects with a "thick skin". In this case, there are regions where intersections with the cut surface are needed (the thick part of the outer shell of the object), and other regions where these are ignored . Remarkably, this can be obtained by modifying the surface representation and following the algorithm for solid objects. Let $\tau > 0$ be the thickness of the shell. Then, the new surface representation is:

$$\hat{f}(\mathbf{p}) = \begin{cases} \tau - f(\mathbf{p}) & f(\mathbf{p}) > \frac{\tau}{2} \\ f(\mathbf{p}) & \text{otherwise} \end{cases} \tag{7}$$

The different rendering of cuts can be seen in Fig.(6). In traditional deformation with cuts, it is required to remesh a complex tetrahedral mesh to represent these different types of solids.

6. GPU Implementation

Our approach can be efficiently implemented exploiting the programmability of fragment shaders. A deformation shader

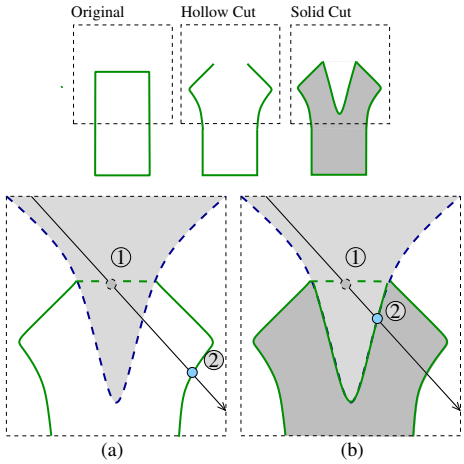
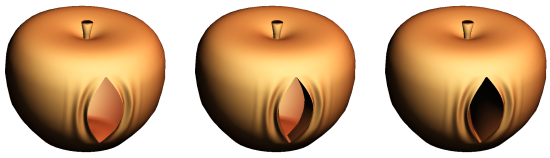


Figure 5: *Ray intersection of cut surface. (a) For a hollow object, we compute intersections with $f(\mathbf{p})$, discarding those inside the cut region (1), and stopping when it is outside the cut region (2). (b) For a solid object, we keep track of intersections with the cut region and the object (point 2).*



(a) Hollow (b) Thick Hollow (c) Solid

Figure 6: *Hollow, Thick and Solid Apple*

is then a realization of the different stages described in Section 3. To render a deformed implicit shader, we first compute the distance field of the partial surface S_D and store it into a 3D texture. We use a *while* loop to traverse the ray along the view direction, and use a conditional statement for early termination. Current GPU's often incur in branch penalties with these type of loops. We believe that branching support of newer GPU's will improve the performance of our approach. Within the loop, we *warp* each sample position according to the displacement stored in a deformation map, and use the result to sample the implicit representation f, stored also as a texture. An interval for a possible intersection is found whenever a change in sign of f occurs. Then, we use a binary refinement to narrow down the intersection coordinates.

6.1. GPU Efficiency

In our approach, we require a distance field representation of an object. Depending on the complexity of the mesh, this may be memory consuming. As an alternative, one may choose smaller representations, trading off the exactness of distance fields. An example is 2D depth maps. A depth or height map is a 2D mapping function $H : R^2 \mapsto R$, which

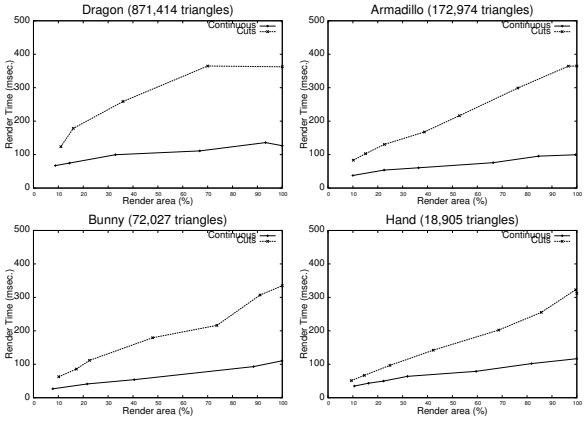


Figure 7: *Rendering time for continuous deformation, in relation to the relative size of the deformation bounding box, as a percentage of the screen area (*512×512*)*

contains depth values to the closest point in the surface along a given vector $\overrightarrow{n}$, normal to the plane where the depth map is defined. Depth maps can be used as an implicit representation of an object as: $f(\mathbf{p}) = H(\mathbf{p}_x) - d(\mathbf{p})$, where $d(\mathbf{p})$ is the closest distance of $\mathbf{p}$ to the plane, and $\mathbf{p}_x$ is the projection of the point into the plane, defined in 2D coordinates local to that plane. Although a depth map is not a complete representation of the closest distance to an object, it can be used as an implicit representation of a convex surface, as long as all points in the surface are visible along the direction normal to the depth map. In Fig. (6), for example, we used a depth map. To overcome the limitations of depth maps, one can approximate a distance field based on depth maps along the six planes defined by a cube embedding the object. We call this a *depth map cube*. For the purpose of rendering, an implicit representation can be obtained by combining the corresponding f functions for the six faces. Given two representations f_1 and f_2, a new representation f can be found as: $f(\mathbf{p}) = \rho \cdot min(|f_1(\mathbf{p})|, |f_2(\mathbf{p})|)$, where the sign ρ is obtained as positive for the cases where $f_1(\mathbf{p})$ and $f_2(\mathbf{p})$ are positive, i.e., when the point is in the inside of both representations. This combination is then repeated for the six faces of the cube. As an example of the memory savings, a 3D layered representation of resolution 128^3 requires 8 MB of texture memory while a depth map cube of the same resolution requires only $6 \times 64KB = 384KB$. However, there is a visibility trade off, and the depth map cube can only be used for geometries which do not contain significant concavities.

6.2. GPU Performance Results

Different displacement textures were created to test our approach. In this paper, we include ones simulating a peel (Fig.(1b)), a twist (Figs.(1c) and (4)), a bend (Fig.(1c)), and a knife cut (Fig.(6)). For Fig.(1b), an object (mask model) was placed under the original torso model, so that when it is peeled, the bottom surface is revealed.

Model	Triangles	Continuous	Cuts (no ESK)	Cuts (ESK)
hand	18,905	89.79	225.46	75.77
bunny	72,027	87.32	249.92	114.95
armadillo	172,974	83.89	260.18	126.00
dragon	871,414	119.24	323.20	126.5

Table 1: *Weighted average of rendering time for continuous and discontinuous deformation in milliseconds.*

Since our approach is largely implemented as pixel shaders, its performance depends on the screen area of the deformation region rather than its complexity in terms of number of vertices. We performed a series of experiments with models of varying size and obtained rendering times for different sizes of the deformation region, relative to the size of the viewport (512×512), on a Pentium XEON 2.8 Ghz PC with 4096 MB RAM, equipped with a Quadro FX 4400 with 512MB of video memory. Composite representation in all these cases used 2MB of texture memory. Fig.(7) shows the rendering times vs. the effective screen area of the deformation region, as a percentage of the window area. We measured time for the rendering of continuous deformation (Continuous) and for a solid rendering of cuts (Cuts), as described in Section 5.1.2. We can see the overhead due to extra intersection tests when rendering solid cuts. To overcome the performance cost of "false" intersections when rendering cuts, we may employ an empty space skipping mechanism, where rays are initiated only at the continuous part of the alpha mask used for the deformation. This information can be obtained from a distance field representation of the cut geometry. If a sample point is found to be within the region of the cut, we can safely skip the sample points which are within the distance given by the distance field at that point, without the need of testing for intersections with the object surface. This method proved to be very fast compared to the original method, and rendering time was comparable to that for continuous deformation.

To summarize the rendering time for the two methods in a comparable way, we computed a weighted average on those results. The weighted average is computed as $\sum a_k t_k / \sum a_k$, where $a_k \in [0,1]$ is the relative render area of the deformed space and t_k is the rendering time for a measure k. Table 1 shows the results of this averaging, where rendering time is given in milliseconds.

6.3. Discussion

Previous results are encouraging for future generations of GPUs. As per-pixel processing becomes more powerful, we believe that the speed up on deformation approaches based on ray traversal can be considerable. For applications where real-time is needed, such as games, the deformation pipeline can be improved with a multi-resolution mechanism. First, ray traversal can be performed in a best effort basis, where deformations taking place in a small area of the screen or at very high speed can be rendered using low frequency

sampling. One of features that makes our approach flexible is the ability to store complex deformations as textures. In our examples, deformations only require from 128 KB (peel) to 2 MB (twist). In general, a displacement texture of size $w \times h \times d$ requires $8whd$ bytes. However, highly complex deformations may require large 3D textures to render high quality images. One mechanism to overcome this is to allow the composition of deformations via simpler textures. Another limiting aspect is the implicit representation of complex objects via 3D textures. For instance, the armadillo model (at a resolution of 256^3) required 64 MB of texture memory. This can be reduced by applying alternative representations as seen in section 6.1. However, handling large complex models requires a partitioning of the object into several textures and improved CPU-GPU bandwidth. With the advent of new vertex capabilities, improved vertex support may be available. We believe our approach still has validity and can complement mesh-based deformation. In fact, it would open possibilities for a hybrid approach, where coarse deformation can be implemented on a vertex shader, while fine-grained deformation and cuts can be done at the pixel level.

7. Conclusions

We have presented a novel pipeline for rendering deformations and cuts using programmable graphics hardware. Unlike previous approaches, which introduce deformation as part of a vertex shader, we define deformation rendering as a pixel shader. This makes possible the rendering of cuts, which are difficult to produce using vertex shaders due to the changes in mesh topology. In our case, we do not need to explicitly split the geometry or remesh along the edges of a cut. It also overcomes the need for a fine tessellation of the mesh, required for smooth rendering of deformation. To implement our approach, we use an implicit representation of the portion of the mesh undergoing deformation, stored in the GPU as a distance field texture. We have shown how displacement maps can be applied to render deformation, similar to image-space displacement mapping, and how they can be extended to render deformation-cuts. Through a number of examples, we show how our method can be used to render hollow and solid objects, without the need for a complex tetrahedral mesh. We believe this approach has applicability in authoring and animation systems, in surgical simulation and interactive applications.

References

[Bar84] BARR A. H.: Global and local deformations of solid primitives. In *SIGGRAPH '84: Proc. of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 21–30. 2, 4, 5

[Bar86] BARR A.: Ray tracing deformed surfaces. *Computer Graphics (Proc. SIGGRAPH 86) 20*, 4 (1986), 287–296. 2

[Blo97] BLOOMENTHAL J.: *Introduction to Implicit Surfaces*. 1997. 2

[BS07] BOTSCH M., SORKINE O.: On linear variational surface deformation methods. *IEEE Transactions on Visualization and Computer Graphics* (2007). 2

[BSM*02] BRUYNS C., SENGER S., MENON A., MONTGOMERY K., WILDERMUTH S., BOYLE R.: A survey of interactive mesh-cutting techniques and a new method for implementing generalized interactive mesh cutting using virtual tools. *Journal of Visualization and Computer Animation 13*, 1 (2002), 21–42. 2

[Coo84] COOK R. L.: Shade trees. *Computer Graphics (Proc. SIGGRAPH 84) 18*, 3 (1984), 223–231. 2

[CSC06a] CORREA C., SILVER D., CHEN M.: Feature aligned volume manipulation for illustration and visualization. *IEEE Transactions on Visualization and Computer Graphics* (September-October 2006). 2

[CSC06b] CORREA C. D., SILVER D., CHEN M.: Discontinuous displacement mapping for volume graphics. In *Fifth Eurographics / IEEE VGTC Workshop on Volume Graphics 2006* (2006), pp. 9–16. 2

[Dav67] DAVIS H.: *Introduction to Vector Analysis*, 3rd ed. Allyn and Bacon, 1967. 4

[d'E04] D'EON E.: *GPU Gems*. Addison Wesley, 2004, ch. 42. 2

[GWL*03] GOODNIGHT N., WOOLLEY C., LEWIN G., LUEBKE D., HUMPHREYS G.: A multigrid solver for boundary value problems using programmable graphics hardware. In *SIGGRAPH/EUROGRAPHICS conference on Graphics hardware 2003* (2003), pp. 102–111. 2

[Har96] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (1996), 527–545. 2

[HEGD04] HIRCHE J., EHLERT A., GUTHE S., DOGGETT M.: Hardware accelerated per-pixel displacement mapping. In *Graphics Interace 2004* (2004), pp. 153–158. 2

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BUHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. In *Eurographics 2005* (2005). 2, 3, 4

[JP02] JAMES D. L., PAI D. K.: Dyrt: dynamic response textures for real time deformation simulation with graphics hardware. *ACM Trans. Graph. 21*, 3 (2002), 582–585. 2

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (1989), pp. 297–306. 2, 4

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (2003), pp. 908–916. 2

[MBK06] MARINOV M., BOTSCH M., KOBBELT L.: GPU-based multiresolution deformation using approximate normal field reconstruction. *ACM Journal of Graphics Tools* (2006). 2

[MS05] MOSEGAARD J., SÃŸRENSEN T.: GPU accelerated surgical simulators for complex morphology. In *IEEE Virtual Reality (IEEE VR)* (2005), pp. 147–153. 2

[NMK*05] NEALEN A., MULLER M., KEISER R., BOXERMAN E., M.CARLSON: Physically based deformable models in computer graphics. In *Eurographics STAR Report* (2005). 2

[OBM00] OLIVEIRA M. M., BISHOP G., MCALLISTER D.: Relief texture mapping. In *Computer Graphics (Proc. SIGGRAPH 2000)*. ACM Press, 2000, pp. 359–368. 2

[PO06] POLICARPO F., OLIVEIRA M. M.: Relief mapping of non-height-field surface details. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), pp. 55–62. 2

[RM95] RUPRECHT D., MÜLLER H.: Image warping with scattered data interpolation. *IEEE Comput. Graph. Appl. 15*, 2 (1995), 37–43. 5

[SGGM06] SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3d distance field computation using linear factorization. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), pp. 117–124. 3

[SKE05] SCHEIN S., KARPEN E., ELBER G.: Real-time geometric deformation displacement maps using programmable hardware. *The Visual Computer 21*, 8-10 (2005), 791–800. 2

[SP86] SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. In *SIGGRAPH '86: Proc. of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM Press, pp. 151–160. 2

[SP91] SCLAROFF S., PENTLAND A.: Generalized implicit functions for computer graphics. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (1991), pp. 247–250. 2

[SPG03] SIGG C., PEIKERT R., GROSS M.: Signed distance transform using graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), p. 12. 3

[WTL*04] WANG L., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Proc. Eurographics Symposium on Rendering* (2004). 2