Trivariate Functions
in
Solid Modeling, Medical Imaging and
Computer Graphics

Sagi Schein
Trivariate Functions

in

Solid Modeling, Medical Imaging and
Computer Graphics

Research Thesis
Submitted in Partial Fulfillment of the
Requirements for the
Degree of Doctor of Philosophy

Sagi Schein

Submitted to the Senate of the
Technion - Israel Institute of Technology

IYAR, 5766 HAIFA May 2006
This Research Thesis was Done Under the Supervision of Professor Gershon Elber in the Faculty of Computer Science

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED
## Contents

1 Introduction

1.1 Volumetric Model Representations ....................................... 4
1.2 Placement of Deformable Objects ......................................... 6
1.3 Bounding the Deformation Error of an FFD .............................. 7
1.4 Real-time Free-form Deformation Using Programmable Hardware ... 9
1.5 Real-time Geometric Deformation Displacement Maps Using Programmable Hardware ...................................................... 11
1.6 Discontinuous Free-Form Deformation ................................. 13
1.7 Silhouette Extraction from Volume Data .................................... 14

2 Placement of Deformable Objects ........................................ 17

2.1 Introduction ........................................................................... 18
2.2 Related Work ......................................................................... 22
2.3 The Placement Algorithm ..................................................... 25
   2.3.1 The Deformation Function ............................................... 25
   2.3.2 The Mass-Springs Model ................................................. 30
2.4 Some Extensions ................................................................. 33
   2.4.1 Further Manipulating the Trivariate Function ..................... 33
   2.4.2 Refinement Algorithm ..................................................... 34
   2.4.3 Animation .................................................................... 38
2.5 Examples .............................................................. 39
2.6 Conclusions and Future Work ................................. 41

3 Handling the Errors in Piecewise Linear Mappings Through FFDs 45
  3.1 Introduction ....................................................... 45
  3.2 Bounding the Deformation Error of Planar Geometry in FFDs ...... 48
    3.2.1 Bounding the Deformation Error for General Triangular Meshes 51
    3.2.2 A Parametrization-invariant Deformation Error Bound ........ 54
    3.2.3 Using Subdivision to Control the Deformation Error ......... 56
  3.3 Conclusions ..................................................... 58

4 Real-time Free-form Deformation Using Programmable Hardware 59
  4.1 Introduction and Background .................................... 59
  4.2 Evaluation of Trivariate B-splines .......................... 61
    4.2.1 Single-Phase FFD Evaluation .............................. 63
    4.2.2 Two-Phase FFD Evaluation ............................... 64
  4.3 Normal Estimation .............................................. 66
  4.4 Efficient Evaluation of B-spline Basis Functions .......... 67
  4.5 Results ........................................................... 69
  4.6 Conclusions and Future Directions .......................... 71

5 Real-Time Geometric Deformation Displacement Maps Using Programmable Hardware 73
  5.1 Introduction ..................................................... 74
  5.2 Deformation Displacement Mapping .......................... 77
  5.3 Hardware-based DDM .......................................... 79
    5.3.1 The Preprocessing Phase .................................. 80
    5.3.2 The Interactive Phase ..................................... 81
  5.4 Extending the Basic HDDM .................................... 84
# List of Figures

1.1 A vector tensor product trivariate B-spline function ........................................ 6  
1.2 Placement of deformable objects ........................................................................ 8  
1.3 Hardware-based FFD for real-time deformable animation ................................. 10  
1.4 Hardware-based DDM for real-time geometry detailing ................................... 12  
1.5 Discontinuous free-form deformation for global model splitting and local surface incision ............................................................. 14  
1.6 Silhouette rendering from volumetric datasets .................................................... 16  

2.1 A variation on the Reptiles picture of M. C. Escher. Multiple deformable objects are placed inside the scene. ............................................................... 19  
2.2 The control lattice of an FFD ............................................................................... 26  
2.3 Constructing two offset curves from a sketch ...................................................... 27  
2.4 The generated extrusion volume .......................................................................... 28  
2.5 The ruling method versus the offset method for the construction of FFDs........ 29  
2.6 Mass-springs optimization for FFD volumes ...................................................... 32  
2.7 Spherical manipulation tools for FFD ................................................................. 34  
2.8 Applying manual manipulation tools on a model of a crocodile......................... 35  
2.9 The deformation error in FFD ............................................................................ 36  
2.10 Controlling the deformation error of a deformed cube model using subdivision ................................................................................................................. 38  
2.11 Animation of the locomotion of ants on an arbitrary terrain ............................. 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.12</td>
<td>Placement of a crocodile laying on the back of a horse.</td>
<td>40</td>
</tr>
<tr>
<td>2.13</td>
<td>A sequence of deformed Porsche cars.</td>
<td>41</td>
</tr>
<tr>
<td>2.14</td>
<td>A snake is suspended in mid air above two virtual objects.</td>
<td>41</td>
</tr>
<tr>
<td>2.15</td>
<td>A deformed “CGGC” group logo.</td>
<td>42</td>
</tr>
<tr>
<td>2.16</td>
<td>Three frames from an animation sequence of a bulldozer parking.</td>
<td>42</td>
</tr>
<tr>
<td>3.1</td>
<td>Approximating smooth surface with a single triangle</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Bounding the length of a vector in a triangle</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Bounding triangle by its circumcircle</td>
<td>53</td>
</tr>
<tr>
<td>3.4</td>
<td>A triangle refinement algorithm</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>A mapping of points through an FFD function.</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>A two-dimensional texture representing the control lattice of an FFD.</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>The single-phase HFFD algorithm</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>The two-phase HFFD algorithm</td>
<td>65</td>
</tr>
<tr>
<td>4.5</td>
<td>Real-time HFFD whales example.</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>Real-time HFFD ants example.</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Real-time HFFD Porsche example.</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>A geometric tile of a flower and its DDM mapping function</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>A block diagram of the HDDM algorithm</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Position and normal texture maps of an HFFD</td>
<td>81</td>
</tr>
<tr>
<td>5.4</td>
<td>Illustrating the HDDM mapping of a single tile.</td>
<td>82</td>
</tr>
<tr>
<td>5.5</td>
<td>The position and normal maps for the camel model.</td>
<td>86</td>
</tr>
<tr>
<td>5.6</td>
<td>Mapping thorn-like tiles over a face model using HDDM</td>
<td>90</td>
</tr>
<tr>
<td>5.7</td>
<td>Four snapshots from a real-time animation sequence of a Buddha tile.</td>
<td>91</td>
</tr>
<tr>
<td>5.8</td>
<td>Metamorphosis of texture tiles in HDDM</td>
<td>91</td>
</tr>
<tr>
<td>5.9</td>
<td>HDDM of polygonal base surfaces.</td>
<td>92</td>
</tr>
<tr>
<td>5.10</td>
<td>HDDM with non-covering tiles</td>
<td>92</td>
</tr>
</tbody>
</table>
5.11 A human face mask covered with hexagonal tiles using HDDM. . . . 93
5.12 Reducing HDDM into a simple displacement mapping. . . . . . 94
5.13 A set of robots walking on the Utah teapot . . . . . . . . . . . . 95

6.1 The cross-section of obstacle $O_i$ in plane $W_k$. . . . . . . . . . . 109
6.2 The silhouette of a human face. . . . . . . . . . . . . . . . . . . . . . 111
6.3 Closing an open model by stitching the open edges of a DFFD function 112
6.4 A model of a sphere is squashed in the vertical direction and wrapped
    around a cylindrical obstacle. . . . . . . . . . . . . . . . . . . . . . . . . 113
6.5 A cross-section of a local DFFD function. . . . . . . . . . . . . . . . 114
6.6 A local DFFD function. . . . . . . . . . . . . . . . . . . . . . . . . . 115
6.7 A cylindrical tool slicing through the Stanford bunny model . . . . 117
6.8 Modeling three incisions in the cheek and nose regions of a human face 118
6.9 Snapshots from an animation sequence of a bulldozer moving through
    a forming hole in a tree. . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.10 Three non-vertical bars are used to model a DFFD function . . . . 119
6.11 The robot model is cut by bars of different orientations. . . . . . . 120

7.1 Visibility cones’ coverage. . . . . . . . . . . . . . . . . . . . . . . . 130
7.2 Silhouettes of a jaw bone. . . . . . . . . . . . . . . . . . . . . . . . . 136
7.3 Shaded jaw bone with the silhouettes overlaid on top. . . . . . . . 136
7.4 Neck bones’ silhouettes. . . . . . . . . . . . . . . . . . . . . . . . . . 137
7.5 Shaded foot bones with silhouettes of the bones and skin overlaid. . 137
7.6 Silhouette curves of an iron protein molecule. . . . . . . . . . . . . 138
7.7 Direct volume rendering of an iron protein molecule . . . . . . . . 139
7.8 An electron density map of a silicium crystal . . . . . . . . . . . . . 139
7.9 A color-coded silhouette of a foot. . . . . . . . . . . . . . . . . . . . 140
7.10 Multicolored silhouette of the knee joint. . . . . . . . . . . . . . . . 141
7.11 The silhouette of an electron density map . . . . . . . . . . . . . . . . 141
7.12 Three different iso-levels of the electron density map. . . . . . . . . 142
8.1 DMS-spline surface . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
8.2 Computing FFD using a trivariate DMS-spline function and a trivari-ate cubic simplex spline . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
List of Tables

5.1 Timing statistics for the HDDM algorithm. 89

7.1 Timing of silhouette calculation for various data sets 134

7.2 Memory consumption and preprocessing time for various datasets . 134
Abstract

Over the past decade, *volume graphics* has been recognized as an important field of research. Most early research in volume graphics was focused on the problem of rendering of volumetric models. Only recently, *volume modeling* had been suggested. Volume modeling encompass the tools, data representations and algorithms that are needed to describe and manipulate the inner parts of a model, as well as its outer shell. Volume modeling can applied in wide range of fields such as material sciences, medical imaging, the movie-making industry, etc. As with surface-based geometric modeling, volume modeling seeks to supply the designer with a set of robust, fast and predictable manipulation tools for designing volumetric models.

Several alternative representations can be used to model volumetric data. The most common is that of voxels. This representation is well understood and more suitable for volume rendering. However, for modeling purposes, voxels are probably not the best of choices mainly due to their bulkiness. For example, even a moderate-size CT scan may contain $256 \times 256 \times 256$ scalars with a 16 bit-per-voxel resolution. Such a model requires more than 128 MByte of storage and even with today’s powerful computers, interaction with models as large as this can become cumbersome and slow. Moreover, since our aim is to develop tools for volumetric modeling, we should consider the state-of-the-art in surface-based modeling. In computer aided geometric design (CAGD), tensor product B-spline functions are, perhaps, the most commonly used surface representations. Therefore, we consider their extension into three-dimensional parametric space as a data representation for volume modeling.

This thesis focuses on tools and algorithms for modeling with trivariate tensor product B-spline functions. These functions are used to solve several geometric modeling and computer graphics problems. We present diverse and novel applications
such as placement of deformable objects inside virtual scenes, incision simulation of
surface models using trivariate functions and silhouette extraction from volume data
where trivariate tensor product B-spline functions are effectively used. In an attempt
to make trivariate tensor product B-spline functions a better deformation tool, we also
supply bounds on the deformation error of polygonal models inside free-form defor-
mation. Lastly, in order to reduce the increased computational burden of trivariate
functions, relative to the B-spline surfaces, we propose fast evaluation techniques
that employ programmable graphics hardware. These evaluation techniques are used
for real-time model deformation and for adding arbitrary surface details in realtime
applications.
Chapter 1

Introduction

Historically, the field of computer graphic and geometric modeling has been more concerned with the boundary representation (B-Rep) of objects than with their internal structure. This can be explained by the fact that the visual appearance of most real-life objects is unambiguously defined by the shape and material properties of their outer surfaces, while objects appeared, most of the time, homogenous. On the other hand, volumetric representations, which describe the inner parts of a model as well as its outer shell, were largely relegated to the fields of medical imaging and scientific visualization. In these fields, volumetric representations are essential since the data that is being visualized is tri(and multi)-variate in nature. Volume graphics was, until recently, notorious for being slow. Even with today’s powerful CPUs and GPUs, real-time volume graphics is still challenging.

At the same time, the field of volumetric modeling has received much less attention. Volumetric modeling relates to the process of designing and manipulating volumetric models. As its surface modeling counterpart, volumetric modeling seeks to supply the designer with a set of intuitive, fast, robust and flexible methods for constructing a model. Better methods for volumetric modeling could have a profound impact, for example, in the context of surgical simulations where they can be
used as part of medical simulators, used by surgeons to improve their technical skills. Similarly, in the animation and movie-making industries, volumetric functions can be used for object deformation.

Although they possess great flexibility as a modeling tool, trivariate functions have yet to become a leading modeling tool. This can be explained by the fact that trivariate functions are hard to grasp and complex to manipulate. Moreover, the evaluation of these functions is usually an order of magnitude slower than their bivariate siblings. As a result, their evaluation times could become prohibitive when incorporated into interactive modeling applications.

The goals of this thesis are to explore novel applications of trivariate B-spline functions in the fields of geometric modeling and computer graphics and tackle some of the limiting factors of trivariate functions. It is our belief that by demonstrating the usefulness of trivariate functions as a modeling tool, we could spur future development in this direction. In the rest of this introductory chapter, we survey the results of this research in brief. The following chapters elaborate on these results. The introduction also lays out some common definitions that will later be used in the remaining of this manuscript.

1.1 Volumetric Model Representations

There are many ways to represent volumetric models, each with its inherent strengths and weaknesses. The oldest and most common technique for representing a volumetric model employs voxels. Voxels (volume elements) are the three-dimensional generalization of pixels. A volumetric model is represented as a three-dimensional lattice of piecewise constant values, such that a continuously varying material is approximated using a discrete representation. This simple representation of volumetric models is very common due to its simplicity [57]. However, and in contrast to their 2D relatives (the pixels), voxels are much more complex to render. Volume rendering, the
construction of 2D images from its volumetric representation, is a topic of great import-
ance both for research purposes and from a practical point of view [62, 77, 60, 98].

Although simple to comprehend, voxels are also too cumbersome and less appeal-
ing for volumetric modeling tasks [70]. Modeling with them would be like painting an image by hand pixel by pixel, or specifying individual triangles inside a polygonal model. Additionally, in order to construct complex volumetric models that are smooth and contain both small-scale and large-scale features, voxel-based models have to contain large amounts of data. The sheer size of the voxel-based volumetric data representation makes it very complex to work with. Alternative representations such as octtrees and PDE-based [31] representations also exist but are not the focus of this work.

In this thesis, we focus on a single volumetric representation, trivariate paramet-
ric functions. We explore the usefulness of this functional representation for solving common geometric modeling and computer graphics problems. Parametric trivariate functions are a generalization of parametric curves into a three-dimensional paramet-
ric domain. As in the case of surfaces, there is no single way to extend curves into trivariate functions. However, generally tensor product trivariate Bézier or B-spline functions are used,

\[
F(u, v, w) = \sum_{i=0}^{l} \sum_{j=0}^{m} \sum_{k=0}^{n} P_{ijk} B_i^d(u) B_j^d(v) B_k^d(w),
\]

\[(u, v, w) \in [U_{\min}, U_{\max}] \times [V_{\min}, V_{\max}] \times [W_{\min}, W_{\max}],\]

where \(P_{ijk} \in \mathbb{R}^3\) are the control points in the case where \(F(u, v, w)\) is a vector function, or \(P_{ijk} \in \mathbb{R}\) in the case of a scalar trivariate B-spline function, \(F\). \(B_i^d(u)\) is the univariate Bézier /B-spline basis function of degree \(d\), which, in general, can be selected independently for each parametric axis.

B-spline, Bézier and Non-Uniform Rational B-spline (NURBS) surfaces are widely used in geometric design due to their superior properties in modeling. Over the
years, a rich set of standard modeling techniques have been developed; most of these tools are also applicable to the trivariate case. Computational tools such as direct node manipulation, subdivision, refinement, evaluation, derivation and integration are readily available and can be easily adapted for volumetric modeling, in the tensor product form.

1.2 Placement of Deformable Objects

One important geometric modeling tool that takes advantage of trivariate parametric functions is called Free-Form Deformation (FFD) [85]. Here, trivariate parametric functions are used to warp a volume of space, inducing a deformation over any embedded model. FFD has found many applications in the field of computer graphics, e.g. model deformation and animation. It has also been employed in the field of medical imaging for non-rigid registration of volumetric models [78]. Figure 1.2 shows a simple illustration of the usage of trivariate functions for object deformation. In Chapter 2 we demonstrate that FFD can also be used to tackle a tedious chore that

Figure 1.1: A vector trivariate function, $F(u, v, w): D \subset \mathbb{R}^3 \Rightarrow \mathbb{R}^3$, is used to map point, $P_i \in D$ to a new point, $F(P_i) \in \mathbb{R}^3$. This figure shows one available application of trivariate functions in geometric modeling.
is encountered during scene design, the placement of three-dimensional deformable objects [81] inside the scene.

This placement algorithm first constructs an FFD function, $F(u, v, w)$, which comes in contact with the terrain underneath. Then, an embedded polygonal model is deformed using $F$ by evaluating $F$ at the vertices of the model. As a result, a deformed version of the embedded model is placed on top of the terrain underneath. To achieve a proper placement, the FFD function should be carefully constructed. First, the base-surface or the bottom part of the FFD function should come in contact with the terrain underneath. Then, $F$ should induce a deformation in a way that would achieve a deformation that is both plausible and visually appealing. The specification of such a complex deformation function might require the manipulation of hundreds, if not thousands of control points. Since manual manipulation of this amount of degrees of freedom is not feasible, Chapter 2 offers an automatic construction scheme for the deformation function. The proposed tool integrates a physically-based spring system simulation of the base surface to allow the FFD to better capture the behavior of real deformable object being placed.

In Figure 1.2(a), multiple animals are placed over arbitrary terrain inside a virtual scene. The placement is done at almost interactive rates. Multiple FFD volumes are placed at different parts of the scene and then the different animals were deformed and placed accordingly. In Figure 1.2(b), 3D text is placed on the body of the Utah teapot. In this scene, three snakes are incrementally placed over the previously placed text models.

1.3 Bounding the Deformation Error of an FFD

FFDs can be thought of as a composition of the input surface model $S(s, t) = [u(s, t), v(s, t), w(s, t)]$ and the deformation function $F$. Even for an FFD function, $F$, which is only tri-quadratic and an input surface, $S$, that is bi-quadratic, the composite
Figure 1.2: FFD can be used to place deformable objects inside virtual scenes. In (a), several animals are deformed and placed inside a scene that is based on M. C. Escher’s “Reptiles” picture. In (b), three-dimensional text is placed over the body of the Utah teapot.

Surface $F(S(s,t))$ becomes a surface of a bi-degree of $12 \times 12$. Such high degree surfaces are difficult to evaluate and process. To alleviate this difficulty, FFD is usually applied to the vertices of the polygonal surfaces that approximates the original free-form surface. It is, therefore, interesting to analyze how large an error is introduced into the approximating model during the process of FFD.

In Chapter 3, we consider a triangular mesh $M$ and an FFD function $F$. Then, each triangle $T \in M$ is implicitly treated as a linear Bézier triangle, which is composed with $F$. The difference between the triangle mapped by its vertices and the composed Bézier triangle is considered to be the local deformation error inside the triangle. Even if the original free-form model is carefully tesselated so that its approximation error remains within a prescribed tolerance, this tolerance would not be typically preserved after FFD. In Chapter 3, we define the term deformation error in the context of FFD and give a bound on this error. We build on results from [40], which gave a bound on the distance between triangles with vertices on a surface and the surface itself. [40] gave these bounds for the right triangle case and claimed that the bounds can be
extended to the general case. Chapter 3 explicitly gives the bounds for the general triangular case, and also develops them in the case of triangles that are mapped through an FFD operation. Additionally, these bounds are used to derive an adaptive subdivision algorithm to control the local deformation error inside each triangle.

1.4 Real-time Free-form Deformation Using Programmable Hardware

One key limitation that hinders the widespread use of trivariate functions is their evaluation cost. Trivariate tensor product B-spline functions take an order of magnitude longer to compute than B-spline surfaces. There are several fast evaluation algorithms that can be adapted to the trivariate case. However, their running times still make it difficult to use trivariate functions inside interactive applications.

Over the past few years, graphics hardware, also known as Graphical Processing Units (GPUs), has become a powerful computational platform. Designed for vector-oriented computations and employing multiple computational pipelines, GPUs are specifically tailored for real-time rendering. Moreover, current GPUs also offer extensive programmability support that further improves their usability. Consequently, many complex algorithms, previously non-implementable in real-time graphics, have become viable options. Specifically, expensive geometric modeling operations could take advantage of the expanded performance of GPUs.

In Chapter 4, we propose a framework for evaluating trivariate B-spline functions on the GPU. The proposed framework is then used to compute a deformation of geometric models using FFD at interactive frame rates. We denote this approach as hardware-based FFD (HFFD). The proposed evaluation scheme is currently suggested only for trivariate B-spline functions with uniform knot sequences. Extending these ideas to general knot-sequences is possible but much less appealing, from the
performance point of view. The algorithm embeds all the computational aspects of
the deformation process on the GPU. As a result, incorporating this capability into
existing modeling application becomes very easy. The proposed tool is capable of
handling the evaluation (and deformation) of well over 1 million points per second,
on modern GPUs.

Figures 1.3 (a) and (b) demonstrate the usage of HFFD in the context of a de-
formable placement application. In Figure 1.3 (a), an animation sequence of six
straight-walking ants is reused to construct an animation over the complex geometry
of a Möbius band. In Figure 1.3 (b), HFFD is used to animate a deformable Porsche
over the complex terrain of a general virtual scene.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{HFFD is demonstrated. In (a), a series of six walking ants are placed and
deformed in real-time on a surface in the shape’s of a Möbius band. (b) demonstrates
that HFFD can also be used in the context of a deformable placement application.}
\end{figure}
1.5 Real-time Geometric Deformation Displacement Maps Using Programmable Hardware

Adding fine-scale surface details to existing smooth models is an important problem in computer graphics. Artificially generated models are usually smooth and lack the details that are present in real-life objects. Over the years, tools such as texture mapping [15], bump mapping [8] displacement mapping [25], and others, more advanced variations of these three, were developed. The most general form of these methods is displacement mapping, which modifies the rendered geometry instead of just altering the way in which the models are shaded. One limitation of displacement mapping is that it is homeomorphic to the base surface and hence can only represent a limited set of surface details. In [33], the Deformation-Displacement Mapping (DDM) algorithm was presented. Basically, DDM defines a parametrization of the volume above the surface of the model. Assuming that $S(u, v)$ is a regular surface and \( \vec{n} = \frac{\vec{\partial S}/\partial u \times \vec{\partial S}/\partial v}{\| \vec{\partial S}/\partial u \times \vec{\partial S}/\partial v \|} \) is the unit normal field of $S$, then a trivariate function, $T(u, v, w) = S(u, v) + \vec{n}(u, v)w$, which spans the regions above (and below) the surface can be constructed. The function, $T$, could be used to map geometric details above (and below) the smooth surface of the model. While fully overcoming the homeomorphic limitation of displacement maps, DDM would usually generate huge amounts of geometry, typically in the order of millions of polygons. Such large amounts of geometry could become a limiting factor, when it comes to real-time rendering.

In order to make DDM more useful in real-time applications, Chapter 5 presents a GPU-based variation of the DDM algorithm, denoted Hardware DDM (HDDM). HDDM is capable of computing DDM at interactive frame rates on modern GPUs. To achieve real-time performance, the HDDM algorithm reverses the roles of geometry and texture maps. The original smooth surfaces of the model are represented as pairs of texture maps for position and normal directions, which are stored on the GPU,
while the geometric details are represented as pure geometric objects. To efficiently evaluate the position and normal values of an arbitrary point on the surface, the algorithm takes advantage of the efficient texture sampling capabilities of modern GPUs. Furthermore, by also storing the geometric details on the GPU, high resolution meshes do not have to be explicitly synthesized on the CPU. The proposed method is capable of reproducing the results of traditional displacement mapping, and at the same time, supporting details that have an arbitrarily complex geometry.

Figure 1.4 (a) shows a model of a camel where the original polygons of the model are replaced by a chain-like tile, demonstrating that HDDM can easily be used to increase the amount of details in existing smooth models. In 1.4 (b), six models of Buddha are placed over the body of the Utah teapot. This image shows that HDDM can handle highly complex tiles with ease.

Figure 1.4: Applying HDDM to two geometric model. In (a), the surface of camel model is replaced with a highly complex periodic tile of a chain. In (b), the Utah teapot is shown with six models of Buddha mapped over its body.
1.6 Discontinuous Free-Form Deformation

Virtually all deformation approaches alter the geometry of the deformed object without affecting its topology. This may become restrictive when a designer wishes to incorporate holes or tears into an existing model. Chapter 6 elaborates on a new FFD variant, coined Discontinuous FFD (DFFD) [82], which we only describe briefly here. DFFD can be used to incorporate discontinuities and deform the model properly while automatically allowing it to split and re-form at the proper locations.

The proposed algorithm takes advantage of a knot insertion procedure to incorporate potential $C^{-1}$ discontinuities into the parametric domain of a trivariate B-spline function, $F(u, v, w)$ of degree $d$ in each of the parametric axes $u,v$ and $w$. The result of inserting $d$ knots into the $u$ (or $v$ or $w$) direction at $u = u_0$, would be that now $F(u, v, w)$ interpolates the iso-surface, $F(u = u_0, v, w)$. By manipulating control points that are on or near this iso-surface, the shape of the discontinuity can be modeled.

When DFFD is being used to deform polygonal models, polygons that cross the discontinuity would not mapped properly. Since the embedded model is deformed by applying $F(u, v, w)$ to its vertices, the crossing edges and faces would not properly reflect the discontinuity. To achieve this, such polygons should be split along the discontinuity. Luckily and since this split is done in the parametric domain of $F$ and hence along axis-parallel planes, computing this splitting is not complicated.

In Chapter 6, the DFFD operation is demonstrated in two different applications. The first splits a deformable object so it wraps around an obstacle(s) in the scene, only to re-merge behind the obstacle(s). Figure 1.5(a) shows a snapshot from an animation sequence of a walking robot. When the robot approaches the bars, it splits open and avoids the collision. The model is re-merged after passing through the bars. The second application demonstrates that DFFD can be used as a general framework for real-time insertion of incisions into the surface of geometric models. Figure 1.5(b),
shows the usage of DFFD as part of an incision simulation application. An incision is made to the cheek of a virtual model, simulating a plastic surgery procedure.

1.7 Silhouette Extraction from Volume Data

Trivariate tensor product B-spline functions can be affectively used in the context of medical imaging to represent and analyze scalar volumetric models. Modern imaging devices such as Computerized Tomography (CT) and Magnetic Resonance Imaging (MRI) systems have become widespread. These tools are very effective in imaging hard and soft tissues. However, the interpretation and comprehension of such images is still more of an art than a science. Physicians usually have to envision an imaginary three-dimensional object to fully analyze the significance of the data. This is mainly due to the fact that volumetric images are packed with details, occlusions and semi-transparent parts. Additionally, volumetric objects may not possess a well-defined inner surface structure, which makes marching-cube-like approaches less suitable for

Figure 1.5: Applying DFFD operation on two geometric models. In (a), the global version of DFFD is used to model three cylinders that are used to split a walking robot. In (b), the local version of DFFD is used to embed incisions into the surface of a human face.
the visualization of such models. To improve the comprehension of such complex data sets we may resort to methods that are borrowed from Non-Photorealistic Rendering (NPR).

NPR covers a plethora of methods that can be used (among other things) to reduce the visual complexity of images. One effective method, which is commonly used by professional illustrators, is to emphasize the silhouettes of an object. Rendering silhouette lines is expected to give the viewer strong cues as to the shape of an object and facilitate its comprehension.

**Definition 1.** A surface point $P$ of object $O$ is a silhouette point with respect to a viewing direction $V$ if $\langle V, N_p \rangle = 0$, where $N_p$ is the normal vector of $O$ at $P$.

Chapter 7 suggests the use of scalar trivariate B-spline functions to approximate scalar volumetric data, captured by modalities such as CT, MRI or simulation data. To extract the silhouette from the data we use a subdivision-based method. This extraction method is used to robustly and efficiently compute silhouettes that have a superior look compared to voxel-based silhouette extraction schemes such as [63]. Since the proposed method is time consuming, the data set is arranged into a view-dependent data structure that is indexed by the viewing vector. Using this data structure, only the relevant portions of the data, those that may yield a silhouette point, are processed. The use of this data structure improves the running time of the algorithm. Naturally, this comes at the expense of larger memory consumption and an offline preprocessing step that is needed to construct the data structure.

In Figure 1.6 (a), the algorithm is used to extract the silhouette of a single iso-level out of a CT scan of a human leg. The same iso-level is also visualized by a polygonal approximation using the marching cubes algorithm. The two renderings are overlayed so that the result can be better comprehended. In Figure 1.6 (b), the output of a simulation of an electron distribution in a molecule is visualized by only rendering the silhouette. In this image, the angular distance of each point from the
exact silhouette is used to construct a depth perception.

Figure 1.6: In (a), the silhouettes from a CT scan of a human leg are shown. One isosurface is extracted using a marching cubes algorithm and rendered normally. The silhouette is overlayed on this rendering to enhance the output image. In (b), the silhouettes are extracted from the results of a simulation of electron distribution in a molecule. Gray-scale encoding of the normal direction is used to enhance depth impression.
Chapter 2

Placement of Deformable Objects

This chapter includes the paper "Placement of Deformable Objects". This paper was published in the journal: The Computer Graphics Forum [81].
2.1 Introduction

The seemingly simple task of placing three-dimensional objects in a scene is fundamental to the field of geometric design and computer graphics. During the process of scene design, a lot of time and effort is spent on placing models inside the scene, switching between views and making small adjustments to the relative position of objects. The problem becomes far more difficult when soft or deformable objects are involved in this placement process.

**Definition 2.** A deformable object is made of an elastic material. The shape of a deformable object is influenced by external factors such as the shape of the surrounding objects coming in contact with it and by exerted forces and by internal properties such as elasticity.

The proper placement of a deformable object needs to take into consideration not only the relative position of other objects inside the scene, but also the deformations that newly placed non-rigid objects undergo when in contact with other objects in the scene. Deformable objects are common in everyday life and viewer familiarity makes the task of simulating the placement of a deformable object much more challenging.

**Definition 3.** The placement of a deformable object is a process in which the deformable object assumes a shape that is influenced by interaction with other objects in the scene, taking into consideration internal properties and external factors.

Exact placement of deformable objects requires the simulation of the physics of the interaction, which mandate the employment of powerful analysis tools such as finite elements. Achieving interactive design rates with such tools is limited by the size of contemporary models. Nevertheless, one can go a long way with a far simpler approach, an approach that will be quite convincing even for the sensitive human eye that is used to seeing deformable objects, such as animals and plants, cloth and
upholstery, rubber and Jell-O-like materials. Specifically, the placement of animals over arbitrary terrain typically requires the contact of legs and arms and, sometimes body and tail, with the environment underneath.

In this work, we will show that Free Form Deformation (FFD) [85] can be employed toward proper and convincing placement of non-rigid objects. FFD is a powerful tool that offers a global deformation scheme to manipulate three-dimensional objects. FFD is typically defined as a mapping, \( F : \mathbf{v} \subset \mathbb{R}^3 \rightarrow \mathbf{V} \subset \mathbb{R}^3 \). A point \( p \in \mathbf{v} \) is mapped into a new location \( P = F(p) \in \mathbf{V} \). In the computer graphics literature, trivariate Bézier [85] or B-spline [49] functions are the common representations for
FFDs. The placement algorithm first construct an FFD deformation function which comes in contact with the terrain underneath. Then, the model is deformed using the deformation function. As a result, the deformed object is placed on top of the terrain underneath.

When a designer tries to place a rigid object in a scene, the result needs to be physically plausible. Objects must never float in mid air and chairs had better be standing on their four legs. Likewise, when placing deformable objects over an arbitrary terrain several constraints need to be observed. First, the object needs to rest upon the upper surfaces of the objects on which it is placed. Second, the bending and stretching of the object should be limited and controlled. Toward this end, a mass-springs model, commonly used for cloth simulation [93], is employed to handle the interaction between the deformation function and the objects underneath. In the context of cloth simulation a mass-springs model is used to animate the temporal behavior of cloth and its complex interaction with rigid objects. Here, a mass-springs model is used as a constrained smoothing method that would place a contact surface on top of an arbitrary geometry with a partial contact constraint.

In spite of their expressive power, FFDs are difficult to exploit. The difficulties stem, mainly, from the fact that designing an FFD function that performs a specific deformation is complex, counter-intuitive, and necessitates the manipulation of a volumetric control lattice in $R^3$.

Recognizing these difficulties, this work offers an automatic construction scheme for the FFD function. The designer is only required to sketch a 2D curve that is projected on the scene and set a few parameters, such as the width and height of the mapping function, in order to fully prescribe the shape of the FFD volume. Moreover, the user does not even have to be aware of the existence of this FFD function unless he/she wishes to modify it directly.

Usually, models deformed through a FFD $F$ are represented as polygonal meshes.
Denote by \( m \) a polygonal model, a polygon or a triangle \( t \in m \) would be mapped to a free-form polynomial triangular patch \( T = F(t) \), provided that \( F \) is polynomial. It is typical for \( T \) to be approximated by mapping only the vertices of \( t \) through \( F \). The edges of the deformed polygons remain linear, resulting in errors being introduced into the deformed model. Finer tessellation of the polygons in \( m \) would result in a smaller deformation error. Herein, we will propose a scheme that selects a subdivision location in a way that reduces the amount of added geometry for the same level of error.

The contributions of this work, in short, are

- Its proposal of a set of simple, interactive and intuitive tools to properly place deformable objects over arbitrary complex terrain, and animate their locomotion.

- Its demonstration of the usage of a mass-springs system to place a deformable object, while controlling its bending and stretching stiffness properties, in a physically plausible manner.

- Its development of precise bounds on the error introduced during the mapping of a polygonal model through an FFD function.

In Section 2.2, we refer to related work and research that developed some of the tools that are about to be used here. In Section 2.3, we define the tools and algorithms toward FFD based placement. In Section 2.4, we describe a few extensions to the basic algorithm. We define and present a more accurate adaptive refinement algorithm and also consider a few extensions that include giving more control over the construction stage of the trivariate deformation function and animation of deformable objects. A few more examples are presented in Section 2.5, and finally, we conclude in Section 2.6 and discuss possible future directions.
2.2 Related Work

The problem of object placement stems mainly from the need to use 2D input and output devices when trying to manipulate objects in 3D. During the manipulation process depth perception is typically lost, and hence, even the simplest of tasks, such as placing a vase on a table, become painstaking tasks. The question of automatic placement of rigid objects inside a scene was investigated, for example, by Xu et al [103]. Xu et al [103] tries to minimize the amount of user intervention during the process of placing multiple rigid objects inside a scene. Their system uses a combination of physical and semantical constraints to enable the user to drop objects into the scene. The system then finds the most suitable placement, physically and semantically, for the objects. The proposed system only handled the placement of rigid objects.

Automatic placement of rigid objects was also tackled in the context of augmented reality (AR). Breen et al [9] proposed a system that places virtual objects into a scene of real objects. The two main issues on which the system focused were occlusion of real objects by virtual objects and vice versa, and the placement of the virtual objects on top of real objects in the scene. The proposed system also dealt solely with rigid object placement.

One of the most common technique for deforming an object is free-form deformation. The idea was first introduced by Sederberg and Parry [85] who proposed the use of trivariate tensor product Bézier functions as the deformation functions. Griesmair and Purgathofer [49] introduced trivariate B-spline functions as the deformation function for FFD. Later work tried to elevate FFD into a design tool and remove some of its restrictions. Coquillart [26] proposed the use of prismatic and cylindrical control lattices to define the deformation map. Chadwick et al [16] introduced the combination of FFD and a Mass-Springs system to control the deformation of fatty tissues in animated characters. MacCracken and Joy [64] used a volume subdivision
scheme to generate a control lattice of arbitrary topology. Other results in this area were more interested in finding new applications for FFD. Coquillart and Jancne [27] used FFD to animate objects inside the deformation function. Elber [33] proposed the usage of Deformation Displacement Maps (DDM) to generate three-dimensional textures and place them on top of curved surfaces. A good survey on the topic of deformable object modeling and FFD can be found in Gibson and Mirtich [46].

FFD is not the only available deformation method. Witkin and Welch [100] defined a physically based global deformation technique. The vertices are treated as masses and a simulation of the object’s dynamics is applied to the deformed points. Baraff and Witkin [1], later, used that physical framework to simulate non-penetrable flexible objects. Both these works consider the boundary representation of the objects when calculating the object deformation, not taking into account the materials from which the deformable objects are made.

On the opposite side of the spectrum of deformation tools come purely geometric methods. Axial Deformation, presented by Lazarus et al [59], attaches a three-dimensional curve, denoted as the axis of the deformation, to the vertices of the model. A manipulation to the shape of the axis induces a change in the position of the vertices, and defines the deformation. Wires, proposed by Singh and Fiume [90], further explored the curve based deformation technique. Wires [90] is a local deformation scheme by nature, and enables the removal of some of the limitations of the axial deformation method [59] while retaining the benefits of an interactive manipulation rates. These methods are often much faster than the physically based deformation approaches but do not guard the designer from specifying deformations that are physically unnatural. These works [100, 1, 59, 90] do not address deformable object placement as a stand-alone problem. While physically based approaches [100, 1] could be adapted to solve this problem, describing the interaction of the deformable object with the scene would be much more cumbersome using the pure geometric
methods [59, 90].

Using FFD for the task of placement of deformable objects must be done with care so that the deformed object retains its original topology. Inept handling may result in an unnatural deformed object with self intersections. Gain and Dodgson [43] presented a scheme to test the injectivity of the FFD volume is proposed, by analyzing \( \det(J) \), the determinant of the Jacobian matrix of the FFD trivariate function \( F \). In that paper, they also claim that an exact computation of \( \det(J) \) is too costly for interactive use.

Modeling the contact surface between deformable objects is a problem that was approached from two general routes. The first route is purely geometrical. For example, methods that use implicit function formulation to model a precise contact surface between deformable objects are presented by Gascuel [44] and Desbrun and Gascuel [30]. These efforts guarantees a smooth contact surface between the deformable objects but can not handle cases where the designer wants to place a general polygonal model on top of another polygonal model. Other work concentrated on adding fine details to a smooth surface. For example, Barghiel et al [4] concentrated on adding highly detailed spline surfaces to a base surface while retaining continuity constraint. Another recent example, Biermann et al [7], in the context of subdivision surfaces, extends this method to multi-resolution cutting and pasting operations. The second general route that is often taken is of using some form of a physically based simulation that solves the motion Ordinary Differential Equation (ODE) and a collision detection system. This physical simulation is used to model the complex interaction between rigid bodies and deformable surfaces. This approach is often seen in the field of cloth simulation. For example [14, 93, 2, 10, 18], to name just a few. These results sought to animate the creases and folds that are created when cloth interacts with a rigid object. The results vary in their details of how the motion ODE is solved and how collisions are handled.
2.3 The Placement Algorithm

The placement algorithm we are about to present is closely related to FFD functions. In fact, the function $F$ is the FFD that will map the geometry of a deformable object $\mathbf{m}$ residing on a planar terrain to a new object $\mathbf{M} = F(\mathbf{m})$ over arbitrary terrain. In Section 2.3.1, we present an automatic scheme to construct the FFD function from the terrain underneath it and represent the FFD as a trivariate B-spline function. In Section 2.3.2, a mass-springs model is employed, augmenting the quality of the contact layer or the surface that is in contact with the terrain.

2.3.1 The Deformation Function

Consider $F$, a trivariate B-spline function of order $o$, defined over a uniform knot sequence with open-end conditions [24](also known as clamped knot sequences) $\tau$, in all three axes:

$$F(u, v, w) = \sum_{i=0}^{l,m,n} P_{ijk} B_{i,\tau_u}^o(u) B_{j,\tau_v}^o(v) B_{k,\tau_w}^o(w),$$

(2.1)

where $(u, v, w) \in [0, 1] \times [0, 1] \times [0, 1]$, $P_{ijk} \in \mathbb{R}^3$ are the lattice of control points of the trivariate B-spline function and $B_{i,\tau_u}^o(u)$ is the $i^{th}$ B-spline basis function. See Figure 2.2.

For the task of placing deformable objects on their bottom part, one of the boundary surfaces of $F$ plays a major role in the placement process:

Definition 4. The contact surface of $F$ is the surface defined by $F(u, v, 0)$.

The contact surface, as the bottom layer of $F$, serves as the contact layer with the scene underneath. The selection of open-end condition for the FFD function in Equation (2.1) is crucial in the context of this placement application. It coerces the FFD function to interpolate the envelope of the control lattice, and specifically, the
Figure 2.2: FFD’s control lattice with its corresponding parametric axes. The control points of the FFD function are denoted by $P_{ijk}$, where the triple index $ijk$ denotes the location of the point in the lattice.

contact surface. Therefore, points $p$ of model $m$ of the form $p = (u, v, 0) ∈ v$ are guaranteed to be mapped to the contact surface $F(u, v, 0)$ and hence, be in contact with the terrain underneath.

In order to fully prescribe the shape of $F$, one needs to specify its control lattice along the $u, v$ and $w$ directions. Let us describe this construction process from the user’s perspective. The process starts by selecting a viewing direction for the scene and sketching a planar curve, $C(u)$, on the image plane of the rendered scene along which the deformable shape is to be placed. Two planar offset curves of distances $±d$ are then constructed from $C(u)$, in the image plane,

$$C_o^+(u) = C(u) + d\vec{N}(u),$$

$$C_o^-(u) = C(u) - d\vec{N}(u),$$

(2.2)

where $\vec{N}(u)$ is the unit normal field of $C(u)$; see Figure 2.3. Construct a planar ruled
surface, $R(u, v)$, between $C_o^+(u)$ and $C_o^-(u)$,

$$R(u, v) = vC_o^+(u) + (1 - v)C_o^-(u), \quad 0 \leq v \leq 1.$$  

$R(u, v)$ is in the image plane. In order to place $R(u, v)$ on top of the objects in the scene, or on the underneath terrain, rays originating from points sampled in $R$ are fired into the scene in the viewing direction, and the first hits, if any, are recorded. Toward this end, the parametric domain of $R$ is point-sampled in a rectangular grid, at a density that is controllable by the user. The denser this grid, the finer the terrain details that can be sensed.

The grid of projected points in the $u$ and $v$ directions is used to define the contact surface, $S(u, v) = F(u, v, 0)$. A B-spline surface, $F(u, v, 0)$, is re-fitted through this new grid with a parameterization that is arc-length in the $v$ direction and chord-length in the $u$ direction. The chord-length is estimated by the length of the middle column of control points, at $(u_i, 1/2)$. As a result, $S(u, v)$ has a uniform, constant speed in the $v$ direction or $\left\| \frac{\partial S}{\partial v} \right\|$ is fixed for all $(u, v)$. Further, $\left\| \frac{\partial S}{\partial u} \right\|$ is also approximately constant along $S(u, 1/2)$, and varies away from $v = 1/2$ when $C(u)$ presents curved regions.

To complete this automatic derivation of the deformation function $F$, we need to define the rest of the control points along the $w$ direction, above the contact surface. We explored two alternatives. In the first, we constructed an extruded volume by extruding the contact surface along a prescribed unit direction $\mathbf{V}$ while $h$ being the

![Figure 2.3: Constructing two offset curves, $C_o^\pm(u)$, from a designer’s sketch, $C(u)$, only to rule a surface $R(u, v)$, between these two offset curves.](image)

Figure 2.3: Constructing two offset curves, $C_o^\pm(u)$, from a designer’s sketch, $C(u)$, only to rule a surface $R(u, v)$, between these two offset curves.
amount of the extrusion:

\[ F_1(u, v, w) = F(u, v, 0) + whV, \quad 0 \leq w \leq 1. \]  

Figure 2.4 presents an extrusion volume constructed for the contact surface of Figure 2.3. The resulting deformed volume \( F_1 \) may be later viewed from an arbitrary direction.

As an alternative, a ruled volume is built between the contact surface and its offset surface, by amount \( h \),

\[ F_2(u, v, w) = F(u, v, 0) + wh\bar{n}(u, v), \]  

where \( \bar{n}(u, v) \) is the unit normal field of the contact surface \( F(u, v, 0) \). Due to the square root normalization in \( \bar{n}(u, v) \) the resulting ruled volume is not rational, and thus, we only approximate the ruled volume by offsetting the control mesh of the contact surface by an amount \( \frac{h}{l-1} \) where \( l \) is the size of the control mesh in the \( w \) direction and \( k \) prescribes the \( k^{th} \) layer of control mesh in the \( w \) direction s.t. \( P_{ijk} = \frac{h}{l-1} \bar{n}(u, v) + P_{i,j,0} \).

Figure 2.4: The generated extrusion volume along a prescribed viewing direction for the contact surface presented in Figure 2.3.
These two approaches are exemplified in Figure 2.5. In (a), the extrusion scheme of $F_1$ is presented and compared to the offset approach in (b).

The approach creating $F_1(u, v, w)$ will never result in a self intersecting trivariate function, provided that $S(u, v)$ is self intersection-free and $\mathcal{V}$ is tangent to $S(u, v)$ at no place. Nevertheless, the result is skewed toward $\mathcal{V}$ and can be less natural than the second alternative. In contrast, $F_2(u, v, w)$ might generate self intersections in the volume where the contact surface has highly curved concave regions, which present a radius of (principal) curvature smaller than $h$. However, its intrinsic construction scheme has no biased direction $\mathcal{V}$, and therefore can be used to place objects on surfaces with an angular span of more than $180^\circ$.

Figure 2.5: An extrusion volume of the contact surface in (a), using the $F_1$ scheme is compared to the normal offset in (b), using the $F_2$ scheme in constructing the FFD.
2.3.2 The Mass-Springs Model

Once the ray-shooting stage is complete, as described in Section 2.3.1, all the control points of the contact surface \( S(u,v) = F(u,v,0) \) are in full contact with the scene underneath. The resulting contact surface emulates a very light fabric material with little bending and stretching resistance. One can employ a mass-springs (MS) model, drawing on ideas presented in [93] or [18], to better control and mimic the behavior of more resistive material in the constructed deformation volume. For the task at hand, we are not interested in the intermediate state of the MS model, only in the final constrained resting state.

Using an MS model, the stretching of the contact surface can be governed. Consider the control points of \( S(u,v), P_{i,j,0}, (i,j) \in [0,n-1] \times [0,m-1] \), where \( l \) and \( m \) are the maximal indices of the control points in the \( u \) and \( v \) directions of the control lattice, respectively.

Each control point, \( P_{i,j,0} \), assumes a mass of \( M \), and is connected to its four direct neighbors, \( P_{i \pm 1,j \pm 1,0} \), with four springs, of equal spring coefficient, \( k \). Unlike traditional MS simulations, \( P_{i,j,0} \) here has a single, directional degree of freedom along which to move, \( \vec{V}_{P_{i,j}} \), a directional constraint that is in either the global extrusion direction, \( V \) (for \( F_{1} \)) or the direction of \( \vec{n}(u_{i}, v_{j}) \) (for \( F_{2} \)). \( \vec{V}_{P_{i,j}} \) is assumed to be a unit vector. The simulation is applied until the system reaches an equilibrium state. Collision detection is used to prevent the masses from penetrating the scene underneath.

For every mass, \( M \), positioned at \( P_{i,j,0} \), Hooke’s law is used to calculate the mass’ acceleration at time \( t + \Delta t \). Here, we assume zero-resting length springs. Handling a mass-springs model with different resting lengths is also possible. Denote by \( \vec{D}_{i,j}^{k,l} \) the Euclidean distance vector from \( P_{i,j,0} \) to \( P_{k,l,0} \) and let \( \rho \) be a velocity damping factor.
Then, the acceleration $\ddot{a}_{i,j}$ of mass $P_{i,j,0}$ equals

$$\ddot{a}_{i,j} = \frac{k}{M} \sum_{k = i \pm 1} \sum_{l = j \pm 1} \vec{D}_{k,l}^{i,j} + \rho \vec{v}_{i,j},$$

with $\vec{v}_{i,j}$ denoting the velocity of $P_{i,j,0}$. Point $P_{i,j,0}$ is restricted to move in the positive or negative direction of $\vec{V}_{P_{i,j}}$. Hence, only $\ddot{a}_{i,j} = \langle \ddot{a}_{i,j}, \vec{V}_{P_{i,j}} \rangle > \vec{V}_{P_{i,j}}$ is considered. From $\ddot{a}_{i,j}$, the new positions and velocities are calculated using an explicit Euler integration scheme, similar formulation as in [18].

The velocity damping term, $\rho \vec{v}_{i,j}(t)$, is needed to stabilize the simulation process and can be thought of as the external friction of the system. $\rho$ is typically negative. We also need to define the termination conditions for the simulation. Toward this end, we identify masses that need further simulation steps. The candidates for further simulations are selected using the following method. For each mass, $M$, at position $P_{i,j,0}$, we consider the four neighboring masses, and calculate the four cross products of vectors toward every consecutive clockwise neighboring pair. If the angles between these four vectors exceed a user defined threshold, we consider that mass as a candidate for further optimization. The same criterion can be also implemented in terms of the dihedral angle between the faces that share vertex that is considered for further simulation. Once no masses are found to be candidates for further optimization, we declare convergence. Such a criterion defines the termination state in terms of the local roughness of the surface of springs and hence stops the simulation when the layer springs represent a smooth surface. As such, it is closer in concept to the application at hand then the more common zero-velocity termination condition.

Assume point $P_{i,j,0}$ needs to undergo an MS simulation. We select a neighborhood of points around it, and consider only these points in the simulation. The neighborhood size enables the designer to influence the final shape of the contact surface by setting the boundary condition of the simulation process, with larger neighborhoods

31
resulting in a less tight contact (see Figure 2.6).

When the contact surface assumes its final shape, the rest of the FFD volume is calculated as before, either as an extrusion volume along a prescribed direction or a ruled volume along the offset, as was described in Section 2.3.1.

Figure 2.6: MS optimization of FFD volumes. The left volume was generated without any springs optimization, the middle volume employed an optimization using a small neighborhood and the right one used a large neighborhood. The resulting positioned snakes are presented before the trivariate volumes.
2.4 Some Extensions

In Section 2.3, the FFD was constructed automatically from a sketched curve and a few parameters such as sampling density and the deformation volume width and height. Yet, we also provide several additional degrees of freedom that are described in Section 2.4.1. The question of proper refinement of polygonal models embedded in the FFD is discussed in Section 2.4.2. In Sections 2.4.3, we explore the animation possibilities into which the proposed approach can be extended.

2.4.1 Further Manipulating the Trivariate Function

So far, we have described a basic automatic construction scheme of the control lattice of the trivariate B-spline function. This basic approach could be extended in several ways. For example, the designer might wish to stretch the object in some parts of the deformation volume and squeeze it in others. In order to achieve this fine and local tuning ability, we generalize the construction scheme of the offset curves (see Equation (2.2)),

\[ C^\pm_u(u) = C(u) \pm d(u)\vec{N}(u), \quad (2.5) \]

where \( d(u) \) is a newly introduced, second scalar curve with the same parametric domain as \( C(u) \). \( d(u) \) serves as a scaling function, leading to a variable offset curve that enables us to locally control the dimensions of the constructed volume in the \( v \) direction.

Similarly, we enable the designer to manipulate the height of a subset of control points of \( F(u, v, w) \). A scalar height function \( h(u) \) or even \( h(u, v) \) could be directly prescribed, replacing \( h \) in Equations (2.3) and (2.4). Alternatively, interactive intuitive editing tools are provided for minor yet of arbitrary resolution, updates of the lattice points of the FFD. The local changes of control point \( P_{ijk} \) are constrained to the direction \( \vec{V}_{P_{ijk}} \). The designer navigates a semi-transparent editing sphere of
varying radius and presses control points above the contact surface that are found inside the sphere. Three types of influence decay were tested: a Gaussian tool where the influence of the tool decreases exponentially from the center, a hat type tool, where the influence of the tool decreases linearly, and a box-shaped tool that applies the same pressure on all points in the radius of influence. The radius of the tool is a user defined parameter which enables multi-resolution control over the affected area. Figure 2.7 shows the effect of tools of different resolution when applied to the FFD volume. Figures 2.8 shows the effect of a pressing tool in a real scene. In addition to these editing capabilities of the lattice above the contact surface, the control points of the contact surface could be independently pulled and/or pushed along $\vec{V}_{P_{i,j}}$, moving the entire column of control points above it the same amount. That is, the translation of $P_{i,j,0}$ coerces all control points $P_{ijk}$, $k > 0$ to follow along.

### 2.4.2 Refinement Algorithm

One undesired result of mapping the vertices of a polygonal model through a deformation function, instead of the exact composition evaluation, is the fact that neither the edges nor the faces of the model deform. A simple remedy that reduces these deformation artifacts subdivides the edges and introduces new, finer faces into the mesh. Consider two vertices, $p_1$ and $p_2$, of edge $\overline{p_1p_2}$ in some polygonal model. Then,
Figure 2.8: (a) shows the result of the pressing spherical tool. In (b), the sphere of influence of the tool is shown in magenta whereas the trivariate function itself is presented in cyan. Both are drawn transparently.

\[ \epsilon_{1,2} \] measures the error in the middle point of the mapped edge, \( \overline{p_1p_2} \).

\[ \epsilon_{i,j} = \left| F \left( \frac{p_i + p_j}{2} \right) - \frac{F(p_i) + F(p_j)}{2} \right|, \quad (2.6) \]

If the error, \( \epsilon_{1,2} \), exceeds a certain distance threshold, the edge is subdivided at the middle point, \( p_m = \frac{p_1 + p_2}{2} \), and the polygons on both sides of these edges are divided as well. This algorithm is described in Chua and Neumann [21].

This approach is simple but has two major drawbacks. First, its error estimation scheme does not bound the maximal possible error, a maximum that can occur at any point along the edge. In addition and for similar reasons, the middle point is not always the best location at which to split the edge. In fact, if \( F(\overline{p_ip_j}) \) has an inflection point, \( \epsilon_{i,j} \) might vanish even though \( F(\overline{p_ip_j}) \) is far from linear. The location that exerts the maximal error is probably a better candidate that would result in fewer polygons for the same level of accuracy.

Consider, for example, a situation where most of the deformation is concentrated in one end of the edge (see Figure 2.9). Subdividing at \( P_m \) makes little sense since
that part of the deformed edge is almost straight. A superior location for subdivision would be \( P_{\epsilon_{\text{max}}} \), where the error, \( \epsilon_{1,2} \), assumes its maximal value. Subdividing at \( P_{\epsilon_{\text{max}}} \) offers a more rapid convergence and reduces the number of subdivisions and the size of the refined model, while achieving the same accuracy.

Parameterize edge \( p_1p_2 \subset v \) as

\[
e_p(t) = tp_2 + (1-t)p_1, \quad 0 \leq t \leq 1
\]

\[
= (u_p(t), v_p(t), w_p(t)),
\]

for some linear functions \( (u_p(t), v_p(t), w_p(t)) \), and let

\[
E_p(t) = F(tp_2 + (1-t)p_1)
\]

\[
= F(u_p(t), v_p(t), w_p(t))
\]

\[
= \sum_{\substack{i,j,k=0 \\
l,m,n=0}} P_{ijk} B_0^{\omega_{i,\tau_u}}(u_p(t)) B_0^{\omega_{j,\tau_v}}(v_p(t)) B_0^{\omega_{k,\tau_w}}(w_p(t)),
\]

(a) \hspace{2cm} (b)

Figure 2.9: A deformation of a polygonal edge between \( p_1 \) and \( p_2 \) in (a) is approximated by mapping these two points to the gray line shown in (b). The maximal error of this approximation, at \( P_{\epsilon_{\text{max}}} \), is clearly not at the middle of the domain. \( P_{\epsilon_{\text{max}}} \) is indeed a better candidate for further refinement.
be the deformed edge in $V$ (see also Equation (2.1)).

The deformation error function along the mapped edge $E_p(t)$ equals

$$\epsilon(t) = \text{dist}(E_p(t), \overline{P_1P_2})$$

where $\text{dist} (\cdot)$ measures the distance from $E_p(t)$ to line $\overline{P_1P_2}$. The same parameter $t$ that maximizes $\epsilon(t)$, at $P_{\text{max}}$ (see Figure 2.9), should also be selected as the subdivision location of edge $e_p(t)$. $E_p(t)$ is a degree $3(o - 1)$ piecewise polynomial B-spline curve where the distance square function $\epsilon^2(t)$ is a scalar piecewise polynomial B-spline curve of degree $6(o - 1)$. Finding the extremum of $\epsilon^2(t)$ requires the solution of a degree $6(o - 1) - 1$ constraint, $\frac{d\epsilon^2(t)}{dt} = 0$. This computation is exact yet it is also a difficult and time consuming task. Instead, one can examine the control polygon of the scalar polynomial curve of the squared error, $\epsilon^2(t)$, and select the nodal point of its maximal coefficient. The nodal value is a parameter that is considered a close parameter value for the free-form shape to the control point with which it is associated. The nodal points are also known as the Greville abscissa [39]. See [39] for ways to compute these parameters.

Looking for the control point of $E_p(t)$ with the maximal distance from line $\overline{P_1P_2}$ and using its nodal value as the refinement location is significantly simpler and more efficient and would achieve a similar result. Again, $E_p(t)$ is a piecewise polynomial of degree $3(o - 1)$ and hence could be represented as a degree $3(o - 1)$ B-spline curve. Compute the distance of the control points of this B-spline curve to line $\overline{P_1P_2}$ and select the nodal value of the control point with the maximal distance. The rest of the refinement algorithm follows the same lines as the refinement algorithm of Chua and Neumann [21]. Figure 2.10 (a) show the results of deforming a cube with a midpoint algorithm like the one described in Chua and Neumann [21]. Figure 2.10 (b) shows the same cube deformed with an adaptive subdivision scheme. The merit for the adaptive scheme is clear mostly in cases where the initial model contained small number of large polygons, like the cube model. Models that already contained large
number of relatively small polygons will benefit less by this adaptive approach with an obvious higher computational cost.

2.4.3 Animation

The idea of using FFD for non-linear animations first appeared in [27]. Combining the ideas of non-rigid object placement, presented here, with the ability to animate the deformed object further reveals the capabilities of the proposed method. First, a FFD volume is constructed along a sketched path following Section 2.3.1. Once the FFD volume is in place, the model is translated in the parametric domain of the FFD along its $u$ axis, the axis that corresponds to the direction of the curve that the designer sketched. Recall the chord-length parameterization of $F(u, v, w)$ (Equation (2.1)) and assume that model $m$ spans a fraction $q$ of the $u$ domain of $[0, 1]$. Then, $m$ is animated by placing it inside $F$ along $v$ from $[0, q]$ and up-to $[1 - q, 1]$. Thus, the model is animated along a user defined route in the scene. Typically, the length of the embedded object, spanning $[0, q]$, would be much smaller than the length of the $u$ parametric domain of the FFD. Further, periodic sequences prescribing the

![Figure 2.10](image_url)

Figure 2.10: In (a), a cube model with 12 triangles was subdivided with a mid-point subdivision scheme. After applying the FFD operation and the subdivision, 360 triangles are formed. In (b), the adaptive error control subdivision scheme was used. The algorithm generated 308 triangles from the cube model.
locomotion of objects or animals over planar domain could be used, allowing for a
two level animation - first, of the locomotion of the object or animal itself, such as
walking or sliding, that is combined with a second warping FFD function that adapts
that locomotion to the local terrain. As a result, a locomotion sequence of an object,
such as a moving animal, which was designed on a planar surface along a straight
trajectory could be used to generate an animation of the same object on an arbitrary
terrain and along an arbitrary path. In Section 2.5, we present several examples,
including animation examples.

2.5 Examples

Figure 2.11 presents one snapshot from a short movie that portrays the locomotion
of several ants in a row. The planar locomotion of the ants along a linear path was
adapted to this complex terrain using an automatically created FFD from a single
sketching curve. Figure 2.12 shows the placement of a crocodile on the back of a
horse. This example shows the ability of our deformation algorithm to handle the
placement of an object over a non-planar complex surface. The shape of the horses’
back is mostly hyperbolic. The offset surface placement method is used to achieved
this placement task.

Nothing prevents us from using rigid objects in this placement process. One
could, with similar ease, also allow the bending of rigid objects as if they were soft.
Figure 2.13 contains snapshots from an animated Porsche warped to overcome the
obstacle in this scene, again using an automatically created FFD.

The use of MS model greatly increases the level of realism that can be expected
in the placement process. In Figure 2.14, a snake is placed over a complex terrain.
Only with the aid of MS model, we were able to bridge the deep gap between the
different objects in the scene underneath. A careful examination of Figure 2.13 will
also reveal a snake in the background, placed with the help of a MS model.

39
Font and text are typically formed out of Bézier curves. Herein, we show how three-dimensional models of strings of characters could be placed on top of curved objects and in arbitrary directions. The width and height of the text can easily be controlled by the same methods that were described in Section 2.3. The resulting string of deformed characters can also be animated as proposed in Section 2.4.3. Several examples of this application are shown in Figure 2.15.

Figure 2.16 demonstrates the ability to employ variable offset curves (see Equation (2.5)) in the design of the FFD volume. Figure 2.16 shows three frames from an animation sequence that shows a bulldozer parking itself. The FFD volume that was used for placing the bulldozer was designed such that it will be adapted to the available space between the two red sport cars.
Figure 2.13: A sequence of deformed Porsche cars. The cars’ movement is animated over a complex terrain with the help of the FFD placement tool. Note the placed snake in the background.

Figure 2.14: A snake is placed in the scene. A MS model is used to bridge the gap between the red jar and the yellow dodecahedron.

2.6 Conclusions and Future Work

We have shown that FFD can serve as an intuitive and powerful design tool toward the placement of deformable objects. Combined with mass springs model, a simple, intuitive and yet effective tool is offered to handle the process of placing non-rigid objects over arbitrary terrains.

Currently, objects are placed relative to a contact surface, $F(u, v, 0)$. Using $F(u, v, 0)$ for a contact surface restricts the placed objects to lay on their bottom part. Using other iso-surfaces of $F$ for a contact surface would enable designers to place deformable objects with a general distance from the contact surface. This can be seen as simple extension of the current algorithm.

Since the scene underneath is discretely sampled, sharp features could be missed, resulting in aliasing and rounding of corners with the possible consequence of the penetration of the placed object into these corners. One solution for that problem
Figure 2.15: A deformed “CGGC” group logo and three snakes placed over different surfaces of the Utah teapot. The snakes exemplify the ability of placing objects on top of other previously deformed objects (snakes and text).

Figure 2.16: Three frames from an animation sequence of a bulldozer parking itself between two red sports cars. This image demonstrates using variable offset curves for the construction of the contact surface. The FFD volume is then constructed on top of the generated surface.

is instead of shooting rays, to adaptively sample the geometry on which the contact surface is placed. Such reverse sampling could improve the reconstruction of sharp and/or discontinuous features during the reconstruction of the contact surface.

Potentially disturbing artifacts that might be created during the placement process are self intersections in the deformed object. A necessary condition for a locally self intersection-free FFD, and hence, a locally self intersection-free deformed object, is that the Jacobian of the deformation function, \( J(F) \neq 0 \), never vanishes, as was
proposed in [43]. The computation of $J(F)$ can be performed symbolically using the tools that were developed in [35]. Visualization and possibly correction of these singular locations would enable the designer to further fine tune the construction of the FFD function at these locations. Yet, the question of detection and elimination of global self intersections in trivariate functions remains completely open.
Chapter 3

Handling the Errors in Piecewise Linear Mappings Through FFDs

3.1 Introduction

FFD can be thought of as a composition of a continuous model $S(s, t)$ and a trivariate deformation function $F(u, v, w)$.

Definition 1. A trivariate tensor product B-spline function $F$ of degree $d_F$ in each parametric axis is defined as $F(u, v, w) = \sum_{ijk} P_{ijk} B_i(u) B_j(v) B_k(w)$ over knot sequence $U = \{u_i\}$.

One problem with this approach is that the resultant surface is generally of high degree. Assume that the deformation function $F$ is a trivariate B-spline function of degree $d_F$ in each axis and that the composed surface $S$ is either a Bézier or a B-spline surface of degree $d_S$ in each axis. Then, the composed surface $F(S(s, t))$ will have a multiplicative degree in each direction. Since $S$ is a polynomial surface of degree $d_S$ in each direction, it has a bi-degree of $d_S \times d_S$. Because this surface is arbitrarily oriented inside the parametric domain of $F$, it may be influenced by all the axes of $F$, each is of a degree of $d_F$. This makes the bi-degree of the composed surface
\(F(S(s, t))\), in exponential notation: \((s^{d_s} t^{d_s})^{d_f} (s^{d_s} t^{d_s})^{d_f} (s^{d_s} t^{d_s})^{d_f}\) or a bi-degree of \((3d_s d_F \times 3d_s d_F)\). For example, even if \(S\) and \(F\) are only quadratic in each direction, the composed surface will be a bi-degree of \((12 \times 12)\).

Such high degree surfaces are usually not desirable, due to their potential instability and cost of evaluation. To circumvent the use of high degree surfaces, FFD is usually applied to the vertices of an approximating polygonal model. Due to the fact that planar polygons that are mapped via \(F\) are likely to become non planar, introducing errors, we restrict ourselves in this discussion to triangular meshes, to ease the analysis. In such cases, a deformation error is inevitable. This is the error we try to bound herein.

Let \(F(u, v, w)\) be a tensor-product trivariate B-spline function and let \(T(s, t) = \triangle ABC \in IR^3\) be a triangle through points \(A\), \(B\) and \(C\) that is linearly parameterized by \(s, t, r\), \(0 \leq s, t, r \leq 1\). Also, denote by \(T_F(s, t) = \triangle F(A)F(B)F(C) \in IR^3\) the mapped triangle that results when applying \(F\) to the vertices of \(T\).

**Definition 2.** \(\epsilon_{F,T} = \sup_{s,t} \|F(T(s, t)) - T_F(s, t)\|\) is the deformation error between the exact composition \(F(T(u, v))\) and its linear approximation, \(T_F(s, t)\).

**Definition 3.** The (asymmetric) Hausdorff distance, \(D(\cdot, \cdot)\), between two sets \(A\) and \(B\) is defined as \(D(A, B) = \max_{p \in A} \min_{q \in B} (d(p, q))\), where \(d\) is a proper metric between pairs of points.

The distance between the two surfaces can be defined by means of the Hausdorff distance between them, which is generally complex to evaluate for freeform surfaces. A simpler approach would consider the parametric distance between the surfaces (as in Definition 2), which serves as an upper bound on the Hausdorff distance. Notice that Definition 2 is not parametrization-invariant. Hence, different parameterizations of \(F\) would yield different bounds.

We start by reviewing the main results of Filip et al. [40] which establish a bound on the deviation of a secant from a curve and on the deviation of a right-triangle from
the surface. The theorems are stated here without proof, but with the same numbers as in the original paper.

**Theorem 2.** in [40] Let $F : [a, b] \rightarrow \mathbb{R}^n$ be any $C^2$ curve and let $l(t) = F(a)(1 - t) + F(b)t$. Then, $\sup \|F(t) - l(t)\| \leq \frac{1}{8}(b - a)^2 \sup \|F''(t)\|$.

**Theorem 4.** in [40] Let $T \subset D$ be a right-triangle with vertices $(A, B, C)$ of the form $C = A + (l_1, 0)$ and $B = A + (0, l_2)$. Let $F : D \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$ be a $C^2$ surface and let $T_F(u, v)$ be the linearly parameterized triangle with $T_F(u, v) = \triangle F(A)F(B)F(C)$, and $d$ the distance from the point of maximal error, $P_0$, to $A$. Then,

$$\sup_{(s,t) \in T} \|F(s, t) - T_F(s, t)\| \leq \frac{1}{2}(d^2 \cos^2(\theta)M_1 + 2d^2 \cos(\theta) \sin(\theta)M_2 + 2d^2 \sin^2(\theta)M_3) \leq \frac{1}{2}(A B C l_1 l_2 T_1 P_0 Q R)$$.  

Figure 3.1: Figure (a) shows a right-triangle $\triangle ABC$ with edge lengths $l_1$ and $l_2$. $\triangle ABC$ is being subdivided into four right-triangles with edge lengths of $l_2^2$ and $l_1^2$. Figure (b) shows a $T_F(u, v)$ linearly approximating part of $F$.  

47
\[
\frac{1}{2}(l_1^2 M_1 + 2l_1 l_2 M_2 + l_2^2 M_3), \]
where

\[
M_1 = \sup_{(s,t) \in T} \left\| \frac{\partial^2 F(s,t)}{\partial s^2} \right\|,
\]

\[
M_2 = \sup_{(s,t) \in T} \left\| \frac{\partial^2 F(s,t)}{\partial s \partial t} \right\|,
\]

\[
M_3 = \sup_{(s,t) \in T} \left\| \frac{\partial^2 F(s,t)}{\partial t^2} \right\|,
\]

and \( \theta \) is the angle between \( AP_0 \) and \( AC \).

Figure 3.1 shows an illustration of the settings. The right-triangle \( T \), in Figure 3.1(a), lies in \( D \subset \mathbb{R}^2 \), while in Figure 3.1 (b), the surface \( F(u,v) \) and the approximating triangle (in red) are being shown. Notice that these bounds are parametrization-dependent.

### 3.2 Bounding the Deformation Error of Planar Geometry in FFDs

To compute the deformation error between a triangle inside a model that is embedded inside the parametric domain of a trivariate B-spline function, \( F(u,v,w) \), we need to consider Theorem 4 of [40]. \( T \in \mathbb{R}^3 \), the parametric domain of \( F \), is parameterized by its barycentric coordinates \( s \) and \( t \). Going one dimension up, we recall that \( A, B, \) and \( C \) are now points in \( D \subset \mathbb{R}^3 \), the parametric domain of the deformation function, \( F \).

Assume \( F \in C^2 \) and consider the composition of \( T(s,t) = (u(s,t), v(s,t), w(s,t)) = As + Bt + C(1 - s - t) \) and \( F(u,v,w), F(T(s,t)) \). In order to put bounds on the deformation error, the second partials of the composed surface, \( F(T(s,t)) \), should be
computed. Then,

\[ \frac{\partial^2 F(T(s, t))}{\partial s^2} = \frac{\partial^2 F}{\partial u^2} \left( \frac{\partial u}{\partial s} \right)^2 + \frac{\partial^2 F}{\partial v^2} \left( \frac{\partial v}{\partial s} \right)^2 + \frac{\partial^2 F}{\partial w^2} \left( \frac{\partial w}{\partial s} \right)^2 + 2 \left( \frac{\partial^2 F}{\partial u \partial v} \frac{\partial u}{\partial s} \frac{\partial v}{\partial s} + \frac{\partial^2 F}{\partial u \partial w} \frac{\partial u}{\partial s} \frac{\partial w}{\partial s} + \frac{\partial^2 F}{\partial v \partial w} \frac{\partial v}{\partial s} \frac{\partial w}{\partial s} \right) , \]

\[ \frac{\partial^2 F(T(s, t))}{\partial s \partial t} = \frac{\partial^2 F}{\partial u^2} \frac{\partial u}{\partial s} \frac{\partial u}{\partial t} + \frac{\partial^2 F}{\partial v^2} \frac{\partial v}{\partial s} \frac{\partial v}{\partial t} + \frac{\partial^2 F}{\partial w^2} \frac{\partial w}{\partial s} \frac{\partial w}{\partial t} + \frac{\partial^2 F}{\partial u \partial v} \left( \frac{\partial u}{\partial s} \frac{\partial v}{\partial s} + \frac{\partial u}{\partial t} + \frac{\partial v}{\partial t} \right) + \frac{\partial^2 F}{\partial u \partial w} \left( \frac{\partial u}{\partial s} \frac{\partial w}{\partial s} + \frac{\partial u}{\partial t} + \frac{\partial w}{\partial t} \right) \]

\[ \frac{\partial^2 F(T(s, t))}{\partial t^2} = \frac{\partial^2 F}{\partial u^2} \left( \frac{\partial u}{\partial t} \right)^2 + \frac{\partial^2 F}{\partial v^2} \left( \frac{\partial v}{\partial t} \right)^2 + \frac{\partial^2 F}{\partial w^2} \left( \frac{\partial w}{\partial t} \right)^2 + 2 \left( \frac{\partial^2 F}{\partial u \partial v} \frac{\partial u}{\partial t} \frac{\partial v}{\partial t} + \frac{\partial^2 F}{\partial u \partial w} \frac{\partial u}{\partial t} \frac{\partial w}{\partial t} + \frac{\partial^2 F}{\partial v \partial w} \frac{\partial v}{\partial t} \frac{\partial w}{\partial t} \right) , \]

(3.1)

since \( T(s, t) \) is linear in both \( s \) and \( t \), and the terms \( \frac{\partial^2 u}{\partial s \partial t} \), \( \frac{\partial^2 v}{\partial s \partial t} \), \( \frac{\partial^2 w}{\partial s \partial t} \), \( \frac{\partial^2 u}{\partial s \partial t} \), \( \frac{\partial^2 v}{\partial s \partial t} \), \( \frac{\partial^2 w}{\partial s \partial t} \), \( \frac{\partial^2 u}{\partial s \partial t} \), \( \frac{\partial^2 v}{\partial s \partial t} \), \( \frac{\partial^2 w}{\partial s \partial t} \), vanish during the derivation of Equations (3.1).

Equations (3.1) gives the second order partials that can be used in conjunction with Theorem 4 of [40] to compute the deformation error. Since \( u(s, t) \), \( v(s, t) \) and \( w(s, t) \) are linear in \( s \) and \( t \), their first derivatives are constants. Hence,

\[ \frac{\partial T}{\partial s} = A - C, \]
\[ \frac{\partial T}{\partial t} = B - C. \]

(3.2)

Denote by \( \{a, b, c\}_{u,v,w} \), the \( u,v \) and \( w \) coefficient of \( A,B \) and \( C \), respectively. Then,

\[ \frac{\partial u}{\partial s} = a_u - c_u, \quad \frac{\partial v}{\partial s} = a_v - c_v, \quad \frac{\partial w}{\partial s} = a_w - c_w; \]
\[ \frac{\partial u}{\partial t} = b_u - c_u, \quad \frac{\partial v}{\partial t} = b_v - c_v, \quad \frac{\partial w}{\partial t} = b_w - c_w, \]

(3.3)

and, Equations (3.1) become,
\[
\frac{\partial^2 F(T(s,t))}{\partial s^2} = \frac{\partial^2 F}{\partial u^2} (a_u - c_u)^2 + \frac{\partial^2 F}{\partial v^2} (a_v - c_v)^2 + \frac{\partial^2 F}{\partial w^2} (a_w - c_w)^2 \\
+ 2 \left( \frac{\partial^2 F}{\partial u \partial v} (a_u - c_u)(a_v - c_v) + \frac{\partial^2 F}{\partial u \partial w} (a_u - c_u)(a_w - c_w) \right) \\
+ \frac{\partial^2 F}{\partial v \partial w} (a_v - c_v)(a_w - c_w),
\]

\[
\frac{\partial^2 F(T(s,t))}{\partial s \partial t} = \frac{\partial^2 F}{\partial u^2} (a_u - c_u)(b_u - c_u) + \frac{\partial^2 F}{\partial v^2} (a_v - c_v)(b_v - c_v) \\
+ \frac{\partial^2 F}{\partial w^2} (a_w - c_w)(b_w - c_w) \\
+ \frac{\partial^2 F}{\partial u \partial v} ((a_u - c_u)(b_v - c_v) + (a_v - c_v)(b_u - c_u)) \\
+ \frac{\partial^2 F}{\partial u \partial w} ((a_u - c_u)(b_w - c_w) + (a_w - c_w)(b_u - c_u)) \\
+ \frac{\partial^2 F}{\partial v \partial w} ((a_v - c_v)(b_w - c_w) + (a_w - c_w)(b_v - c_v)),
\]

\[
\frac{\partial^2 F(T(s,t))}{\partial t^2} = \frac{\partial^2 F}{\partial u^2} (b_u - c_u)^2 + \frac{\partial^2 F}{\partial v^2} (b_v - c_v)^2 + \frac{\partial^2 F}{\partial w^2} (b_w - c_w)^2 \\
+ 2 \left( \frac{\partial^2 F}{\partial u \partial v} (b_u - c_u)(b_v - c_v) + \frac{\partial^2 F}{\partial u \partial w} (b_u - c_u)(b_w - c_w) \right) \\
+ \frac{\partial^2 F}{\partial v \partial w} (b_v - c_v)(b_w - c_w)).
\] (3.4)

Since $F$ is a trivariate B-spline function, $\frac{\partial^2 F}{\partial u^2}$, $\frac{\partial^2 F}{\partial v^2}$, $\frac{\partial^2 F}{\partial w^2}$, $\frac{\partial^2 F}{\partial u \partial v}$, $\frac{\partial^2 F}{\partial u \partial w}$ and $\frac{\partial^2 F}{\partial v \partial w}$ can be computed analytically by generalizing the formula for deriving B-spline curves to the trivariate case [37].

By plugging the results of Equation (3.4) into Theorem 4 of [40], we can compute the deformation error of triangles mapped through the FFD. For an input triangular mesh, we could consider the maximal value over all triangles that gives a global bound over the deformation error of the deformed mesh. A more practical approach would consider only the triangles where the deformation error is above $\epsilon_{F,T}$ and subdivides them.

50
3.2.1 Bounding the Deformation Error for General Triangular Meshes

The above result assumes that right-triangles are embedded inside the deformation function. For a right-triangle (see Figure 3.1(a)), the perpendicular line from $Q$ to $AC$ hits $R$, the median point of $AC$. The same argument holds for the median of $AB$. These lines subdivide a right-triangle, $T$, into four right-triangles, $\{T_i\}_{i=1}^4$ with edges that are half the length of the original edges of $T$. Denote by $P_0 \in T_1$, the point of maximal error, and assume without loss of generality that this point is internal to $T_1$. A similar argument holds if $\{P_0 \in T_i\}_{i=2}^4$. Define $V = A - P_0$ to be the vector that connects $P_0$ with vertex $A$ (see Figure 3.2) and let $d = \|V\|$; then $V = [d \cos(\theta), d \sin(\theta)]$, $\theta = \angle P_0AB$. In the case of right-triangles, we can provide a sharp bound of $d \cos(\theta) \leq \frac{l_1}{2}$ and $d \sin(\theta) \leq \frac{l_2}{2}$. This bound is used in Theorem 4 of [40] to get an upper bound on the error without knowing the position of $P_0 \in T$. 

Figure 3.2: Bounding the length of a vector $\overline{P_0A}$ in a general triangle $T = \triangle ABC$. 

51
To bound the deformation error in the general triangular case, we can apply similar arguments to those used in the right-triangular case. A slightly worse bound of \( d \cos(\theta) < l_1 \) and \( d \sin(\theta) < l_2 \) holds. In the case of a general triangle, the lengths of the edges \( \{T_i\}_{i=1}^4 \) are bounded by the lengths of edges of the original triangle. Hence, the deformation error is bounded by \( \sup_{(u,v) \in T} \| F(u, v) - T_F(u, v) \| \leq \frac{1}{2}(l_1^2 M_1 + 2l_1 l_2 M_2 + l_2^2 M_3) \), which is not sharp in general, yet only a factor of four from the right-triangle bound.

As a consequence of subdividing a general triangle \( T \) into four sub-triangles (see Figure 3.2), each sub-triangle has at least one known point where the deformation error is zero. A slightly different approach could consider the original triangle without any subdivision. Over \( T \), the error function \( \epsilon_{F,T}(s,t) \) has at least three known zeros, at the vertices. Using similar arguments to those of Theorem 4 of [40], given \( P_0 \), we need to bound \( \min_{A,B,C} \max_{Q \in T} \min_{Q \in T} \| AP_0 \|, \| BP_0 \|, \| CP_0 \| \) in order to bound the error. If we assume that \( P_0 \) is free to be at any point of \( T \), we consider \( P_0 \) that is farthest from any vertex of \( T \) to bound \( \epsilon_{F,T}(s,t) \). In other words, we seek \( P_0 \) such that \( P_0 = \arg \max_{Q \in T} \min_{A,B,C} \max_{Q \in T} \min_{Q \in T} \| AP_0 \|, \| BP_0 \|, \| CP_0 \| \).

**Lemma 1.** For an acute triangle, \( T = \Delta ABC \), 
\[ P_0 = \arg \max_{Q \in T} \min_{A,B,C} \max_{Q \in T} \min_{Q \in T} \| AP_0 \|, \| BP_0 \|, \| CP_0 \| \] is the circumcenter.

**Proof:** By contradiction. Assume there exists another point, \( R \), such that \( R = \arg \max_{Q \in T} \min_{A,B,C} \max_{Q \in T} \min_{Q \in T} \| AP_0 \|, \| BP_0 \|, \| CP_0 \| \) and that \( R \) is internal to one of the triangles \( \Delta P_0 AB \), \( \Delta P_0 AC \) or \( \Delta P_0 BC \). \( R \) is now at a shorter distance from at least one of the vertices of \( T \) than \( P_0 \). Then, without loss of generality assume that \( A \) is the closest vertex to \( P_0 \) and \( \| AR \| = \min \{ \| AR \|, \| BR \|, \| CR \| \} \), which implies \( \| AR \| < \| AP_0 \| \). Then \( R \) would not satisfy the maximum property.

**Lemma 2.** For an obtuse triangle, \( T = \Delta ABC \) with circumcircle \( C \) (see Figure 3.3), the length from \( P_0 = \arg \max_{Q \in T} \min_{A,B,C} \max_{Q \in T} \min_{Q \in T} \| AP_0 \|, \| BP_0 \|, \| CP_0 \| \) to the corresponding vertex closest of \( T \), is smaller than \( r \), the circumradius of \( C \).
Figure 3.3: A triangle $T$ being bounded by its circumcircle $C$ with circumradius $r$.

**Proof:** $P_0 \in T$ by definition. Assume, without loss of generality, that the distance $AP_0 \geq r$, the circumradius of $C$, and that $A$ is the closest vertex to $P_0$. Then, since $P_0$ is inside an obtuse triangle, its distance from at least one of the other vertices, $B$ or $C$, is smaller than $r$. Otherwise, $T$ would not be bounded by $C$. Hence, $AP_0$ would not qualify as the minimum, in the first place.

Lemma 2 actually tells us that it does not matter which triangle is selected and that acute triangles always perform worse than obtuse ones, in the sense that the maximal distance to one of the vertices of an acute triangle is larger than in the case of obtuse ones (assuming they are both bounded by $C$). By Lemma 1, the distance, $d$, from the point of maximal error, $P_0$, to the closest vertex in $T$ is bounded by the circumradius, $r$. Additionally, since $\cos^2(\theta) < 1$, $\sin^2(\theta) < 1$ and $\cos(\theta) \sin(\theta) < \frac{1}{2}$, $\forall \theta$, and by using the intermediate inequality in Theorem 4 of [40], the bound on the
deformation error becomes \( \sup_{(u,v) \in T} \|F(u,v) - T_F(u,v)\| \leq \frac{r^2}{2}(M_1 + \frac{1}{2}M_2 + M_3) \).

In order to use the above bound, we need to express \( r \) using the edges \( g_1 = \|AB\|, g_2 = \|AC\| \) and \( g_3 = \|BC\| \). To that end, a known equation from plane geometry \cite{97} can be used:

\[
r = \frac{g_1g_2g_3}{4\sqrt{s(g_1 + g_2 - s)(g_1 + g_3 - s)(g_2 + g_3 - s)}},
\]

where \( s = \frac{g_1 + g_2 + g_3}{2} \).

### 3.2.2 A Parametrization-invariant Deformation Error Bound

The bounds that are given in Section 3.2 are parametrization-dependent. For example, consider the surface \( F = xy, 0 \leq x, y, x + y \leq 1 \) that is approximated by the triangle \( T = \triangle ABC \) where \( A = (0,0), B = (0,1) \) and \( C = (1,0) \). Using Theorem 4 of \cite{40} we get \( M_1 = M_3 = 0 \) and \( M_2 = 1 \). Hence, the error is bounded by \( \frac{1}{8}(0 + 2 + 0) = \frac{1}{4} \). Now, let us reparameterize \( F \) by \( x = s^2, y = t^2 \). In this case, \( M_1 = M_3 = 2 \) and \( M_2 = 4 \) and the bound becomes \( \frac{1}{8}(2 + 2 \ast 2 + 2) = \frac{3}{4} \). Since the surface itself remains unchanged, it is clear that such a bound is not intrinsic to the surface.

Consider the surface case as in \cite{40}, and let us consider a reparametrization \( u(s,t) \) and \( v(s,t) \) of \( F(u,v) \) and use the chain rule to derive

\[
M_1 = \sup_{(u,v) \in T} \left\| \frac{\partial^2 F(u,v)}{\partial s^2} \right\|. \quad \text{We get,}
\]

\[
\frac{\partial F^2(u,v)}{\partial s^2} = \frac{\partial^2 F(u,v)}{\partial u^2} \left( \frac{\partial u}{\partial s} \right)^2 + 2 \frac{\partial^2 F(u,v)}{\partial u \partial v} \frac{\partial u}{\partial s} \frac{\partial v}{\partial s} + \frac{\partial^2 F(u,v)}{\partial v^2} \left( \frac{\partial v}{\partial s} \right)^2
\]

\[
+ \frac{\partial F(u,v)}{\partial u} \frac{\partial^2 u}{\partial s^2} + \frac{\partial F(u,v)}{\partial v} \frac{\partial^2 v}{\partial s^2}.
\]

Clearly, \( M_1 \) (and also \( M_2 \) and \( M_3 \)) could be affected from the reparametrization functions \( u \) and \( v \).

One could alleviate this problem by projecting the deformation error, \( \epsilon_{F,T} \) onto \( n(u,v) = \frac{\frac{\partial F}{\partial u} \times \frac{\partial F}{\partial v}}{\| \frac{\partial F}{\partial u} \times \frac{\partial F}{\partial v} \|} \), the unit normal field of \( F \). Notice that since \( \frac{\partial F(u,v)}{\partial u} \) and \( \frac{\partial F(u,v)}{\partial v} \)
are in the tangent plane of $F(u, v)$, \(\frac{\partial F(u, v)}{\partial u} \cdot n(u, v) = \frac{\partial F(u, v)}{\partial v} \cdot n(u, v) = 0\). Since the parametrization-dependence of the deformation error, \(\epsilon_{F,T}\), results from movements in the tangent plane of $F$, measuring \(\epsilon_{F,T}\) only in the direction of $n(u, v)$, would remove it. Denote

\[
\hat{M}_1 = \sup_{(u, v) \in T} \left\| \frac{\partial^2 F(u, v)}{\partial u^2} \cdot n(u, v) \right\|,
\]

\[
\hat{M}_2 = \sup_{(u, v) \in T} \left\| \frac{\partial^2 F(u, v)}{\partial u \partial v} \cdot n(u, v) \right\|,
\]

\[
\hat{M}_3 = \sup_{(u, v) \in T} \left\| \frac{\partial^2 F(u, v)}{\partial v^2} \cdot n(u, v) \right\|.
\]

Then, using the triangle’s inequality followed by Schwartz’s inequality on Theorem 4 of [40], the following inequality holds:

\[
\sup_{(u, v) \in T} \| (F(u, v) - T_F(u, v)) \cdot n(u, v) \| \leq \frac{1}{8} (l_1^2 \hat{M}_1 + 2l_1 l_2 \hat{M}_2 + l_2^2 \hat{M}_3), \tag{3.7}
\]

which is a bound on the projected deformation error and is derived by modifying Definition 2.

For brevity, we only derived \(\hat{M}_1\) with respect to $s$. \(\hat{M}_2\) and \(\hat{M}_3\) can be derived in a similar manner.

\[
\frac{\partial F^2(u, v)}{\partial s^2} \cdot n(u, v) = \frac{\partial^2 F(u, v)}{\partial u^2} \left( \frac{\partial u}{\partial s} \right)^2 \cdot n(u, v) + 2 \frac{\partial^2 F(u, v)}{\partial u \partial v} \frac{\partial u}{\partial s} \frac{\partial v}{\partial s} \cdot n(u, v) + \frac{\partial^2 F(u, v)}{\partial v^2} \left( \frac{\partial v}{\partial s} \right)^2 \cdot n(u, v)
\]

\[
= L \left( \frac{\partial u}{\partial s} \right)^2 + 2M \left( \frac{\partial u}{\partial s} \frac{\partial v}{\partial s} \right) + N \left( \frac{\partial v}{\partial s} \right)^2, \tag{3.8}
\]

where \(L\), \(M\) and \(N\) are the coefficients of the second fundamental form [38]. Note that Equation (3.8) is still not reparametrization-invariant and needs to be normalized by the coefficients of the first fundamental form.

55
3.2.3 Using Subdivision to Control the Deformation Error

In order to locally control the deformation error, the bounds from Section 3.2.1 or Section 3.2.2 can be used in conjunction with an adaptive subdivision procedure.

Let $F$ be a trivariate tensor product B-spline function and define $\delta$ as an error threshold. In this section, we would like to define a subdivision procedure over the original triangular mesh so that it would guarantee that $\epsilon_{F,T} < \delta$ for each triangle in the subdivided mesh.

**Algorithm 1. BoundMeshError**

1. For each triangle, $T$, in the input, push $T$ onto stack.

2. If the stack empty, stop. Otherwise, pop $T$ and compute $\epsilon_{F,T}$.

3. If $\epsilon_{F,T} > \delta$
   
   then apply procedure RefineTriangle($T$)

   else insert $T$ into the output mesh.


**Algorithm 2. RefineTriangle($T$)**

1. Find the median of each edge of $T$.

2. Construct four new triangles out of $T$ by breaking the edges of $T$ in the middle s.t. $T = \bigcup \{T_i\}_{i=1}^4$ and $\bigcap \{T_i\}_{i=1}^4 = \Phi$ (See Figure 3.4(a)).

3. Purge $T$. Push $\{T_i\}_{i=1}^4$ onto stack.

We need to show that the proposed subdivision process indeed reduces the deformation error for $T$ as a result of each subdivision step so that the algorithm terminates after a finite number of subdivision steps.

56
Figure 3.4: A refinement scheme for triangle $T = \triangle ABC$. In (a), $T$ is subdivided independently into four triangles at the medians of its edges. (b) shows the same triangle with four neighboring triangles that are subdivided only to preserve the connectivity of the refined mesh.

**Theorem 5.** The deformation error $\epsilon_{F,T} < \max\{\epsilon_{F,T_1}, \epsilon_{F,T_2}, \epsilon_{F,T_3}, \epsilon_{F,T_4}\}$ is reduced after every subdivision step by at least a factor of four.

**Proof:** The subdivision of $T$ inserts three additional vertices, $D$, $E$ and $F$, into the triangular mesh $M$ and forms a partition of $T$. Thus, the values of the second derivatives of $F$, $\frac{\partial^2 F}{\partial u^2}$, $\frac{\partial^2 F}{\partial v^2}$, $\frac{\partial^2 F}{\partial w^2}$, and $\frac{\partial^2 F}{\partial u \partial w}$ for each sub-triangle $\{T_i\}_{i=1}^4$, are bounded from above by the same values that are computed over $T$. Since the edges of $T$ are split in the middle after each subdivision step and by Equation (3.1), the error is reduced after every subdivision step, by a factor of four at least.

Notice that during the execution of Algorithm 1, T-junctions might be introduced into the refined mesh, and the mesh would become non-manifold. Since $F$ only maps the vertices of a mesh, cracks (black holes) would be formed in the deformed models.
To remedy the problem in the triangles, which do not satisfy the refinement criteria and share an edge with a previously divided triangle, an additional subdivision step should be added so that the mesh remains valid. Figure 3.4(b) illustrates this case where the central triangle is selected for refinement due to the error criteria whereas its three neighbors are subdivided only to preserve the 2-manifold property of the mesh.

3.3 Conclusions

This chapter puts a constructive bound on the deformation error that occurs as a result of deforming a polygonal model by an FFD function. Section 3.2.1 gives an explicit bound on the error in the case of general triangular meshes. The main problem with this bound is that it is not invariant under reparametrization. In Section 3.2.2, we tried to alleviate this parametrization dependency using projection arguments. The result is more intrinsic to the surface than the previous bounds. However, it still depends on the first fundamental form of the surface and the search of a completely intrinsic bound on the deformation error should continue.
Chapter 4

Real-time Free-form Deformation Using Programmable Hardware

4.1 Introduction and Background

Over the past few years, GPUs have become a powerful computational platform. Designed for vector-oriented computations and employing multiple computational pipelines, GPUs are specifically tailored for real-time rendering. Nevertheless, general-purpose computations can also take advantage of their high computational throughput. Specifically, expensive geometric modeling operations, which usually precede rendering, could take advantage of the expanded performance.

In this work, we propose a framework for deforming geometric objects in the GPU, using Free-Form Deformations (FFD) [85], at interactive frame rates (see Figure 4.1). FFD uses a trivariate deformation function, $F : D \subset \mathbb{R}^3 \Rightarrow \mathbb{R}^3$, usually in the form of trivariate Bézier or B-spline functions. The trivariate B-spline functions, which are used in this work, are defined as

$$F(u, v, w) = \sum_{i,j,k=0}^{l,m,n} P_{ijk} B_{i,\tau_u}^o(u) B_{j,\tau_v}^o(v) B_{k,\tau_w}^o(w),$$  \hspace{1cm} (4.1)
where $P_{ijk} \in \mathbb{R}^3$ form the 3D lattice of control points of the trivariate B-spline functions, and $B_{i,\tau_u}^o(u)$ is the $i^{th}$ B-spline basis function of order $o$ over knot sequence $\tau_u$. $o$ and $\tau_u$ will be omitted at times for clarity.

Moreover, in the ensuing discussion we consider the deformation of polygonal models where the exact deformation is approximated by deforming the vertices of the models, $V_i$, as $\tilde{V}_i = F(V_i)$.

The evaluation of trivariate B-spline functions is usually considered a rather time-consuming operation. Each evaluation of a vertex, $V_i$, requires the evaluation of $o^3$ B-spline basis functions. To improve the deformation rates of FFDs, hardware acceleration is one option that can be considered. Previous work [20] already outlined some general details of one such solution. In [20], a system for hardware accelerated EFFD [26], a variant of the original FFD, was described. The system is based on OpenGL’s evaluators [86], and suggested a hardware implementation that could support the EFFD evaluation. Another hardware-based approach for FFD evaluation [55] implemented it using Field Programmable Gate Arrays (FPGA). This solution is, however, less suitable for general purpose computer graphics applications.

![Diagram](image.png)

Figure 4.1: A trivariate function, $F(u, v, w) : D \subset \mathbb{R}^3 \Rightarrow \mathbb{R}^3$, is used to map point, $P_i \in D$ to a new point, $F(P_i) \in \mathbb{R}^3$. 

60
In this work, we present a hardware-based FFD (HFFD) evaluation scheme for trivariate B-spline functions. The proposed method is fully capable of real-time deformation of complex models at interactive frame rates using modern GPUs. The proposed scheme can be easily integrated into existing rendering applications, assuming that a proper GPU is available. Furthermore, the proposed scheme also incorporates all the computational aspects of the evaluation of FFDs inside the GPU.

The rest of this paper is organized as follows. In Section 4.2, we describe the GPU-based evaluation algorithm for trivariate B-spline functions and consider two evaluation alternatives on the GPU. Section 4.5 presents some animation and modeling results using HFFD. Finally, we conclude in Section 4.6 and consider some future directions.

4.2 Evaluation of Trivariate B-splines

In order to apply an HFFD operation, trivariate B-spline functions should be evaluated. As in curves and surfaces, Equation (4.1) could be represented in a matrix form. Due to the vector oriented architecture and specialized command set of the GPU, vector and matrix operations are much more efficiently processed in it than in the CPU. Hence, one can rearrange Equation (4.1) so that only vector-matrix multiplications and inner-products are used. In the tri-quadratic case, Equation (4.1) becomes

\[
F(u, v, w) = [B_0(u), B_1(u), B_2(u)]
\begin{bmatrix}
P_{002} & P_{102} & P_{202} \\
P_{001} & P_{101} & P_{201} & P_{212} \\
P_{010} & P_{110} & P_{210} & P_{222} \\
P_{200} & P_{210} & P_{220}
\end{bmatrix}
\begin{bmatrix}
B_0(v) \\
B_1(v) \\
B_2(v)
\end{bmatrix}
\]

(4.2)
Let

\[ M_k = \begin{bmatrix} P_{00k} & P_{10k} & P_{20k} \\ P_{01k} & P_{11k} & P_{21k} \\ P_{02k} & P_{12k} & P_{22k} \end{bmatrix}, \]

and

\[ \tilde{M}_k = [B_0(u)B_1(u)B_2(u)]M_k \begin{bmatrix} B_0(v) \\ B_1(v) \\ B_2(v) \end{bmatrix}. \]

Then,

\[ F(u, v, w) = \sum_{k=0}^{2} \tilde{M}_k B_k(w). \]  
(4.3)

The last summation can also be replaced by a single inner product computation on the GPU.

Many algorithms exist for evaluating B-spline basis functions with arbitrary knot sequences, and can be found in standard textbooks [24, 39]. In the case of B-spline basis functions with uniform knot sequences, fast and simple evaluation techniques do exist [23]. In contemporary GPUs, the uniform B-spline basis functions could be evaluated directly from their analytic, piecewise polynomial form. Using this method, a single-precision accuracy is attained.

Section 4.2.1 describes an implementation of an evaluation procedure for quadratic trivariate B-spline functions in the vertex processor, following Equation (4.3). Unfortunately, this approach is also limited due to some contemporary hardware restrictions, which will be discussed below. In Section 4.2.2 we suggest an improved scheme that uses a two-phase evaluation method that takes advantage of both the vertex and fragment processors.
4.2.1 Single-Phase FFD Evaluation

The most natural way to implement the HFFD is in the vertex processor, an approach that we denote the Single Phase (SP) approach. For every vertex $V_i$, the vertex processor could execute a vertex shader program that would evaluate Equation (4.3).

In an effort to reduce CPU usage and since GPUs are highly optimized for rapid texture queries, we propose to store all the control points of $F(u, v, w)$ in the GPU in the form of a texture map. This approach has become viable with the introduction of vertex textures, in shader model 3.0 [45]. To that end, all the control points of $F$ are stored as a small texture map, $T_{in} = [P_{ijk}]$, which encodes the position of $P_{ijk}$ in the RGB floating point color channels. Since floating-point volumetric textures are not currently accessible from within the vertex shader, the three-dimensional lattice of control points of the HFFD function is laid out in a sequence of two-dimensional floating point texture maps (see Figure 4.2). Then, the spatial coordinates of each vertex, $V_i \in D$, are used as texture coordinates in $T_{in}$, retrieving all the control points that are needed for the evaluation of $F(V_i)$.

The above evaluation scheme is specifically designed to take advantage of the efficient matrix/vector-based architecture of the GPU. Unfortunately, for tri-cubics and even tri-quadratics, it also requires more temporary memory than available on

![Figure 4.2: The three-dimensional lattice of control points of $F(u, v, w)$ is laid out, plane after plane, in a two-dimensional floating point texture map, $T_{in}$.](image)

63
the most advanced contemporary GPU’s. To alleviate this problem the read-only registers of the GPU were used. Since read-only registers can only be written from the CPU, some computations have to be pulled back. Accordingly, basis functions are computed once in the CPU and stored as part of the input values for the vertex shader. The above algorithm is illustrated in Figure 4.3.

An evident drawback of this approach is revealed when trying to animate an object, $\mathcal{O}$, inside the parametric domain of $F$. In such cases, the basis functions must be recomputed for each vertex, $V_i \in \mathcal{O} \subset D$, in every animation step. Consequently, the method puts a major burden on the CPU and the graphics bus. An improved two-phase approach, which performs the computations solely in the GPU, is suggested in Section 4.2.2.

### 4.2.2 Two-Phase FFD Evaluation

To better utilize the available hardware assets in the GPU, it is natural to ask whether the deformation algorithm can take advantage of the fragment processor too. If an
algorithm could be divided into two parts, the first part can be executed in the vertex processor and the second part on the fragment processor(s), exploiting the full depth of the computational pipeline of the GPU. Moreover, since the fragment processor offers much better texture queries support, bottlenecks due to the performance of the vertex texture should be eliminated.

The two-phase (TP) HFFD algorithm is shown in Figure 4.4. In the first phase, the deformation is computed by employing both the vertex shader and the pixel shader. In the second phase, the rendering is conducted by replacing $V_i$ by $F(V_i)$. To evaluate $F$, the B-spline basis functions are evaluated in the vertex processor. Toward this end, the vertex shader only has to know the position of each vertex in the parametric domain of $F$. Then, the fragment shader is called for each fragment, and the texture map, $T_{in}$, which contains $P_{ijk}$, the control points of $F$, is sampled. At this stage, the fragment shader possesses all the necessary information for evaluating Equation (4.3) and the deformed position of the vertex can be computed. The results of the deformation, $F(V_i)$, are rendered (essentially stored) in an off-screen buffer. In this work, we used pbuffers [101].

Nonetheless, there is one difficulty in the two-phase scheme. The vertex shader
is evaluated for each vertex $V_i \in \mathcal{O}$, forming a trivial one-to-one mapping between the data and the computational kernel that processes it. The difficulty is found in the need to ensure that for each invocation of the vertex processor, a single fragment shader will be called. To circumvent this difficulty, the object is first rendered as a 2D-grid of vertices onto a plane orthogonal to the viewing direction. Moreover, each vertex, $V_i$, is rendered inside the viewing frustum so that no vertices are clipped or Z-filtered. The result is a single frame that encodes the deformed positions, $F(V_i)$, into the color channels of the pbuffer. This concludes the first phase of the algorithm.

In the second phase of the algorithm, the 3D model is actually rendered. The pbuffer, which was constructed in the first phase, is reattached as a new texture and used as the input to this phase (see Figure 4.4). Recall that in the first phase, a 2D-grid representation of the model was generated. Each pixel in that image corresponds to a single vertex in the model and holds its deformed position. Hence, in the second phase, the output image is sampled from the vertex shader, resulting in the deformed position $F(V_i)$ that replaces the original vertices, $V_i$. Finally, the model is also transformed and lit to properly render the deformed model. In our tests, the SP approach proved to be 50% slower than the TP approach.

4.3 Normal Estimation

To properly render the deformed geometry, the direction of the normal for each deformed vertex $V_i$, $\vec{N}_i$, should also be computed. While $\vec{N}_j$ is orthogonal to $\mathcal{O}$, this is not necessarily the case for $F(\vec{N}_i)$ and $F(\mathcal{O})$, since $F$ is generally a non-conformal mapping. The correct normal $\vec{N}_i$ of $F(\mathcal{O})$ at $F(V_i)$ can be approximated by mapping the complementary, tangent space at $V_i$. Compute two new independent vectors $\vec{T}_1^i, \vec{T}_2^i \in \mathbb{R}^3$ that are orthogonal to $\vec{N}_i$, the original normal at $V_i$. $\vec{T}_1^i$ and $\vec{T}_2^i$ span the tangent space of $\mathcal{O}$ at $V_i$. $F(\vec{T}_1^i)$ and $F(\vec{T}_2^i)$ span the tangent space of the mapped geometry at $F(V_i)$, and can be approximated as $F(\vec{T}_1^j) \approx F(V_i + \vec{T}_1^j \epsilon) - F(V_i)$, $j = 1, 2$.
for some small $\epsilon \in \mathbb{R}^+$.

During the first phase of the deformation algorithm (see Section 4.2.2), two new points, $P^j_i = V_i + \epsilon T^j_i, j \in \{1, 2\}$, are rendered in addition to $V_i$. In the second phase, for each vertex $V_i$, the pbuffer is sampled three times, for $F(V_i), F(P^1_i)$ and $F(P^2_i)$. Then, the direction of the normal of $F(V_i)$ equals $\hat{N}_i = [F(V_i) - F(P^1_i)] \times [F(V_i) - F(P^2_i)]$. This approximation scheme for computing normals triples the amount of work in the two rendering passes. Note that this approach also reduces the effective size of the off-screen pbuffer, and hence, the maximal size of the available models.

Due to memory constraint of the SP algorithm, this method is only applicable to the TP method.

### 4.4 Efficient Evaluation of B-spline Basis Functions

For many applications, quadratic B-splines offer enough expressive power. However, there are cases where smoother deformation functions are required. In such cases, cubic B-spline functions might become desirable. When trying to use the formulation of Sections 4.2.1 and 4.2.2, the restrictions of the GPU become clear. Since there is a limited amount of read/write registers in contemporary GPUs, not all intermediate values can be stored in memory. A simple computation, similar to that in Section 4.2.1 for the cubic case, shows that more than seventy registers are required. In the rest of this section, a general scheme for using the hardware capabilities to evaluate B-spline basis functions of order $o$ in terms of B-spline basis functions of order $o-1$ is presented. Specifically, this approach could be used to evaluate cubic trivariate B-spline functions in terms of quadratic functions, as was shown in Section 4.2. Consider the B-spline curve, $C(t)$, of order $o$ over knot sequence $\tau$. Traversing only basis functions that are
non-zero for \( t_i \leq t < t_{i+1} \), one gets,

\[
C(t) = \sum_{m=i-o+1}^{i} P_m B_{m,\tau}^{o}(t)
\]

\[
= (1) \sum_{m=i-o+1}^{i} P_m \left[ \frac{t-t_m}{t_{m+o-1} - t_m} B_{m,\tau}^{o-1}(t) + \frac{t_{m+o} - t}{t_{m+o} - t_{m+1}} B_{m+1,\tau}^{o-1}(t) \right]
\]

\[
= (2) \sum_{m=i-o+2}^{i} \left[ P_m \frac{t-t_m}{t_{m+o-1} - t_m} + P_m \frac{t_{m+o-1} - t}{t_{m+o} - t_{m+1}} \right] B_{m,\tau}^{o-1}(t).
\] (4.4)

Where (1) is due to the recursive relation of DeBoor’s algorithm and (2) is due to the locality of B-spline basis functions. Then, Equation (4.4) becomes,

\[
C(t) = \sum_{m=i+2-o}^{i} [P_m(1-\alpha) + P_m\alpha] B_{m,\tau}^{o-1}(t), \quad \alpha = \frac{t-t_m}{t_{m+o-1} - t_m},
\]

\[
= \sum_{m=i+2-o}^{i} Q_i B_{m,\tau}^{o-1}(t).
\] (4.5)

This shows that the new control points \( Q_i \) are simply a linear combination of the original control points.

For B-spline basis functions with uniform knot sequences, Equation (4.5) can be further simplified. As in Section 4.2 and without loss of generality, assume the uniform knot sequences are over the integers, \( t_i = i \). Then \( \alpha = \frac{t-t_m}{t_{m+o-1} - t_m} = \frac{t-i}{o-1} \).

In the case of FFD, a trivariate version of Equation (4.5) should be used. An evaluation matrix, similar in form to that of Equation (4.3), is constructed. However, the entries in this matrix are efficiently computed by bilinear texture sampling from the control point texture, \( T_m \). The rest of the evaluation scheme remains the same, using Equation (4.3), over basis functions that are one degree smaller. Hence, evaluating an \([o \times o \times o]\) order trivariate would cost a multiplicative factor, \( \psi \), which is the cost ratio between a bilinear and a nearest-neighbor texture sampling, more than the evaluation of an \([o-1 \times o-1 \times o-1]\) order trivariate. Since, as of today, no GPU
exists that supports bilinear sampling of 32 bit floating-point textures, this method will become available only in future generation GPUs.

4.5 Results

The HFFD algorithm was implemented in Cg [66], and integrated into an existing OpenGL-based viewer that is part of the Irit [54] solid modeler. The presented images were all captured in run-time. In the examples below, we used uniform tri-quadratic B-spline functions for the deformation. In all cases the TP method was used. The timing statistics were all taken from a system equipped with an Nvidia 6800. All the models that are shown in the examples use simple triangle lists.

Figure 4.5 shows a snapshot from an animation sequence of four swimming whales. The total number of mapped vertices in the scene is 112848. We were able to achieve 7.5 frames per second when executing this animation. The animation is achieved by supplying the vertex shader with an animation offset in the direction of the animation for each vertex $V_i \in O$ in the static whale model. On the vertex shader, this offset was added to the stored position before evaluating $F$.

Figure 4.5: Four swimming whales are animated around a periodic deformation function. Each whale model contains 28212 vertices. The non-deformed whale model is shown on the top-left. A zoom-in on two of the whales is shown on the right image.
Figure 4.6: A snapshot from an animation sequence of seven walking ants. The shape of each ant is deformed to fit the shape of a Möbius band. Each ant model contains 10500 vertices. A zoom-in on one ant is shown below. The scene is modeled after a drawing by M.C. Escher.

In Figure 4.6, a line of seven ants is walking (having an additional articulated animation applied) around a Möbius band, in the style of M.C. Escher’s “Ants” drawings. The Möbius band serves as the base surface of a trivariate B-spline function. Each ant model has 10500 vertices and the total number of vertices that are mapped through $F$ is 73500. This example reaches 8.4 frames per second.

Figure 4.7 shows a composite of four snapshots of an animation sequence of a single model of a Porsche. The Porsche is Jello-warped as it drives along a path following a complex terrain. In this example, the model of a Porsche car is deformed inside a complex scene, to exemplify that HFFD can be integrated into existing OpenGL-based applications. The Porsche model contains about 66000 vertices. This example achieved about 5.5 frames per second.

When testing the above examples in the context of a software-based implementation of FFD, we could reach deformation rates of less than a one frame per second. These timing statistics are not totally strict since the software-based FFD that was used implements a more generic version of trivariate tensor product splines. Nevertheless, the large timing gap between the hardware implementation and software
Figure 4.7: On the left, a composite of four snapshots from an animation sequence of a moving Porsche. The cars are Jello-warped to drive along complex terrain. The right image shows a zoom-in on one of the cars. The Porsche model has 66000 vertices.

implementation give a clear indication that HFFD is much more efficient than its FFD counterpart.

4.6 Conclusions and Future Directions

In this work, a method for hardware-based FFD (HFFD) evaluation is presented. Using the proposed approach, complex models can be deformed at interactive frame rates.

This work exemplifies two basic difficulties in implementing general purpose algorithms in GPUs. First, GPUs possess only a limited amount of temporary registers and no secondary storage. As more advanced algorithms are being ported to GPUs, the restriction on the amount of read/write memory becomes ever more problematic. In such cases, an approach that would take advantage of both the vertex processor
and the fragment processor could alleviate this difficulty. By guaranteeing a one-to-one mapping between the execution of a vertex shader and the execution of a pixel shader, the computations can be broken up into two parts.

A second design issue that is demonstrated here is the problem of feedback. The GPU’s architecture is that of a stream processor, which moves data only downstream. If an algorithm requires feedback, special considerations must be taken. In this work we used a pbuffer, which requires expensive context-switches. Better feedback mechanisms, possibly by employing the newly finalized specification of EXT_framebuffer_object extension, may reduce that overhead too.
Chapter 5

Real-Time Geometric Deformation Displacement Maps Using Programmable Hardware

This chapter includes the paper "Real-Time Geometric Deformation Displacement Maps using Programmable Hardware". This paper was published in the journal The Visual Computer [83] and was presented in the 13th Pacific Conference on Graphics and Applications (PG).
5.1 Introduction

Real-time graphics has always sought to achieve the best possible rendering quality within pre-defined performance requirements. Due to performance restrictions, algorithms for adding surface details play an essential part in reaching this objective. Quite a few computer graphics schemes have been devised for adding details to smooth surfaces, among which texture mapping [15] and bump mapping [8] are the most common. When it comes to conveying the impression of rough surfaces, bump mapping is usually quite effective. However, since no geometry is manipulated, the smooth nature of bump-mapped objects is exposed easily, for example, when their shadows are inspected. To ameliorate this limitation, displacement mapping [25] was developed. Displacement mapping works as follows. Denote the original smooth surface, $S(u, v)$, as the base surface. A scalar displacement field $d(u, v)$ is added to the original geometry $S(u, v)$ in a specified direction, typically the direction of the unit normal of $S(u, v)$, $\vec{n}(u, v)$. This operation results in a geometrically displaced version of the original surface, $S_d(u, v) = S(u, v) + \vec{n}(u, v)d(u, v)$. More contemporary approaches [73, 91] tried to incorporate depth impression into the rendering of texture-mapped objects. These methods are usually successful at improving the visual quality of textured objects. However, they cannot handle the mapping of topologically-complex structures.

Displacement mapping indeed resolves the smooth silhouette and incorrect shadow problems of bump mapping. Moreover, with modern programmable graphics units (GPUs), one can support it in real-time applications. Several GPU-based approaches [51, 95, 96] for computing displacement mapping in real-time have been suggested recently. These approaches employ the pixel shader to compute displacement mapping in the direction of the viewer. They do not operate on vertices. [95] used a 5-dimensional visibility map to compute the intersection of each pixel with displacement map. Hence, its accuracy depends on the amount of sampling in a high order volume, limiting the resolution of features that can be sensed in real-time. In [51], polygonal patches
are rendered over the regions that might be affected by displaced base mesh. The intersection of this patch with the displacement map is computed at the pixel level. A recent work, [96], focused on the rendering of finely-detailed objects and eliminated some of the limitations of [51]. A generalized displacement map that records the shape of the detail geometry along a preselected set of viewing directions is pre-computed. As a result, non-bijective detail objects can be handled too. However, with the recent introduction of vertex-texture support in shader model 3.0, displacement mapping can now also be directly implemented at the vertex shader.

By its construction, the displacement map is homeomorphic to the original surface. As a result, the geometry of the added details is restricted. In addition, even though the base surface governs the resolution of the displacement, its resolution is typically much lower than that of the added details. This usually forces the base surface to be further refined in order to gain sufficient resolution. Controlling, and more so adaptively controlling, the refinement level of the base surface is not trivial, as it depends on the detail that is being added.

Several results [69, 33, 74] proposed tiling surfaces using geometric details as a method for adding details on surfaces. [69] suggested the use of volumetric texture tiles in the context of a ray-tracing rendering environment for tile surfaces with complex details. In [33], the Deformation-Displacement Mapping (DDM) algorithm was presented. While fully overcoming the homeomorphic limitation of displacement maps, DDM still generates huge amounts of geometry, typically in the order of millions of polygons. While not a prohibitive limitation in software-based rendering, such a large data size makes it difficult to use DDM in real-time applications. Like DDM, [74] parameterized a thin shell above a base surface so that complex details can be properly placed. However, [74] used volumetric textures for real-time modeling and rendering of topologically-complex features. Their results are comparable to ours but require larger preprocessing times and might be harder to integrate into existing
Exploiting the recent advances in graphics hardware, this work presents a computer graphics hardware variation of the DDM algorithm that can be supported in real-time applications. This solution allows a powerful generalization of the displacement mapping scheme; moreover, it is also interactive. The presented algorithm reverses the roles of geometry and texture maps, and represents the added details as pure geometric objects. Following [50], the base surfaces are represented as positions and normal fields, which are stored efficiently in the form of texture maps. Consequently, and as in DDM, several geometric detail locations could result from a mapping over the same base surface position – hence removing the one-to-one barrier. By taking advantage of the efficient texture sampling capabilities of modern GPUs, position and normal information of an arbitrary point on the surface can be computed efficiently. Furthermore, by storing the geometric details on the GPU, high resolution meshes do not have to be synthesized on the CPU. Additionally, considering details as core geometry has other important benefits. Details can be manipulated using any geometry manipulation tools; for example, attribute settings (coloring, translucency), animation and metamorphosis. Finally, the base surface can be tiled using any planar tiling scheme. A single tile can be used to cover the whole base surface in a regular tiling, or alternatively, multiple tiles could be used, forming a semi-regular tiling scheme.

The proposed method is capable of reproducing the results of traditional displacement mapping, and at the same time, supporting details that have an arbitrarily complex geometry. By reversing the role of the base surface and its geometric details, the two noted limitations of displacement maps, homeomorphism and complex resolution control, can be resolved.

The rest of this paper is organized as follows. In Section 5.2, we briefly survey the original DDM scheme. In Section 5.3, we show how to adapt the original DDM...
approach to contemporary computer graphics hardware environments, and present our hardware-based DDM (or HDDM) variant. Section 5.4 suggests several extensions to the HDDM algorithm. These are, mainly, the implementation of real-time HDDM over polygonal base surfaces and various options for animating texture geometry. In Section 5.5, we demonstrate the capabilities of this introduced real-time scheme by presenting screen snapshots of real-time renderings. Finally, we conclude in Section 5.6.

5.2 Deformation Displacement Mapping

Let $S(u, v)$ be a regular parametric surface, the base surface, and let $\vec{n}(u, v)$ be the unit normal field of $S(u, v)$. Assume $\vec{n}(u, v)$ points outside the object $S(u, v)$. Then, a trivariate function that parameterizes the surface $S(u, v)$ and the volume around it is defined with the aid of the unit normal field of $S$, $\vec{n}(u, v)$, as

$$T(u, v, w) = S(u, v) + \vec{n}(u, v)w.$$  \hspace{1cm} (5.1)

With the help of $T(u, v, w)$, any polygonal or rational geometry detail, $D$, can be mapped as a detail over $S(u, v)$ using the composition of $T(D)$. Utilizing the construction from [33], the trivariate function of Equation (5.1) parameterizes the volume above and below the base surface. $w$ in Equation (5.1) could serve as a displacement function, $w(u, v)$, which would yield the regular homeomorphic displacement mapping. Instead, this work (following [33]) assumes the embedding of a new arbitrary surface detail $D(r, t) = (u(r, t), v(r, t), w(r, t))$ in $T(u, v, w)$. Then, the composition of $T(D)$ maps $D$ into a deformed space that is above/below $S(u, v)$. This operation is, in essence, a variation over freeform deformations [85], only here the deformation function $T$ is fully prescribed by the base surface $S$.

Without loss of generality, assume $T(u, v, w)$ is defined over the unit cube, $u, v, w \in [0, 1]$. The exact result of a composition of a rational parametric surface detail $D(r, t)$
Figure 5.1: The flower tile, $D$, on the left is mapped through the trivariate function $T$, defined over $S$, resulting in a deformed and placed version of it, $T(D)$, on the right.

with a rational parametric deformation function $T$ is computable and is a rational parametric surface of a multiplicative order [33]. Nonetheless, the computer graphics world also encompasses numerous models that take the form of polygonal meshes. If $D$ is a polygonal model, the composition operation can be approximated by mapping the vertices of mesh $D$, through $T$ (see Figure 5.1). That is, each 3D vertex $V_i \in D$ is mapped to its new location $T(V_i)$. In the ensuing discussion, we will assume $D$ is a polygonal model.

The DDM scheme, as suggested in [33] in the context of off-line rendering, can, therefore, be expected to yield a huge amount of polygons. In the next section, we expand on these ideas and seek ways to reduce the memory overhead in order to support the DDM scheme in real-time, with the aid of modern programmable computer graphics hardware.
5.3 Hardware-based DDM

The hardware-based variation of the DDM algorithm (the HDDM) that is presented here exploits the programmability of the vertex processor, and is implemented as a single pass vertex shader. The key feature that makes HDDM possible is the ability to sample textures from within the vertex processor [45]. This capability is now available on GPUs that support shader profile 3.0, and as stated, is the cornerstone for executing the HDDM algorithm on the GPU.

The HDDM algorithm is composed of two phases, the preprocessing phase and the real-time, interactive phase. Figure 5.2 illustrates the different stages of the algorithm. In Section 5.3.1, the preprocessing phase is presented and in Section 5.3.2 the interactive phase is discussed.

Figure 5.2: A block diagram that illustrates the HDDM algorithm. The diagram portrays the preprocessing stages vs. the interaction steps and the computation in the CPU vs. the GPU, of smooth base surface $S$ and geometric detail $D$. 

79
5.3.1 The Preprocessing Phase

Given a vertex $V_i = (u_i, v_i, w_i)$ of a polygonal model $D$, the DDM algorithm requires the evaluation of the base surface $S(u_i, v_i)$ and its normal field, $\vec{n}(u_i, v_i)$ to derive the vertex mapped location, $T(V_i)$, following Equation (5.1). In [33], this process is conducted directly. That is, a surface $S$ is indeed evaluated at $(u_i, v_i)$, as is its normal field. Clearly, such an evaluation could have devastating effects on the expected real-time performance if applied on-line and an alternative solution must be found.

Recall that the base surface, $S$, contains information at much lower frequencies and is far smoother relative to the geometry detail, $D$. One can convert $S$ into an image-based representation, as in [50], and since $S$ is assumed to be smooth, only a relatively small number of samples is required to represent $S$ with adequate accuracy. Every location in the 2D image representation of $S(u, v)$ is evaluated into a position in $\mathbb{R}^3$ and every location in $\vec{n}(u, v)$ is evaluated into a unit normal field on $S^2$. These positions, $P(u, v) \rightarrow (x, y, z)$, and normal $N(u, v) \rightarrow (n_x, n_y, n_z)$ texture maps are sampled at uniform intervals in $u$ and $v$ over $S$ and stored in the GPU memory as two 32-bit (per color channel) floating-point texture maps, in the RGB channels, as part of the preprocessing phase (see Figure 5.3). This double-texture image representation of $S$ as the pair $(P, N)$ is equivalent to a uniform tessellation of $S$, assuming that bilinear interpolation of the four neighboring samples is used for fractional values. In this work, $(P, N)$ were constructed by evaluating $S$ and its normal field on the CPU. A different scheme for using the graphics hardware is described in Section 5.4.1.

To complete the preprocessing phase, the desired geometric texture tile $D$ is uploaded as well, in a canonical coordinate system, into the memory of the graphics card. In this work, we tested display lists and vertex buffers for storing $D$ on the GPU. Consequently, replacing $D$ with a more (or less) fine tessellation version of itself, or replacing it with a whole new tile becomes a very simple task. Moreover, this replacement does not harm the expected real-time behavior of the algorithm since it
merely requires switching to a different stored display list, or if the new tile is not already loaded, the re-rendering of a single such tile.

### 5.3.2 The Interactive Phase

Having completed the preprocessing phase, the algorithm enters the interactive phase. Assume we seek to tile the base surface $S$ with $n \times m$ texture tiles $D$. The task of the CPU is simply to traverse $n \times m$ times over all tiles, and provide the GPU with the offset amount of the tile in the parametric space of $S$. The amount of offset that should be applied to each tile depends on the number of tiles, their shape, and the application. In the case of a simple rectangular tile, the parametric domain of the base surface is divided by the number of desired tiles in each direction. The above offset scheme assumes a coverage of the base-surface with instances of $D$. However, such a coverage is not intrinsically required and different layout schemes could be easily used instead, forming gaps in the $u$, $v$ or $w$ directions of $T$. 

Figure 5.3: The position, $\mathcal{P}$, (in (a)) and normal, $\mathcal{N}$ (in (b)) texture maps of the body of the Utah teapot.
The offsets are used to place each instance of $D$ at its proper location in the parametric domain of $S(u, v)$, and hence, $T(u, v, w)$. The offset values, $[u_0, v_0]$ (see Figure 5.4), are transferred to the vertex shader that implements the HDDM algorithm. Such a construction reduces the load on the graphics bus, prevents the GPU from waiting for geometry feeds from the CPU, and reduces the load on the CPU.

On the GPU side, the vertex shader receives the following information from the CPU:

1. Handles of the $P$ and $N$ textures.
2. The $u_o$ and $v_o$ offsets of the currently rendered tile in the parametric domain of $S$.
3. The dimensions of the textures.
4. The position and the normal of the current vertex.

Recall that the position and normal are taken from the stored (canonic) tile, $D$. Given vertex $V_i = (x_i, y_i, z_i) \in D$, the $x_i$ and $y_i$ coordinates are treated as

![Figure 5.4: Tile $D_{ij}$ is placed over $S$ at offset amounts of $u_0$ in the $U$ direction and $v_0$ in the $V$ direction.](image)
offsets within the domain of the tile. The actual texture coordinates are computed as
\((u_f, v_f) = (x_i + u_o, y_i + v_o)\). However, due to the fact that currently available GPUs
do not support bilinear sampling of 32-bit floating textures, interpolation should be
implemented directly in the shader. Denote as \(uTexSize\) and \(vTexSize\), the size of
the position texture map, \(P\) over domain \([0,1]^2\). \(P\) is sampled at four neighboring
positions,

\[
(u_f, v_f)^{i,j} = \left\lfloor \left( u_f \ast uTexSize, v_f \ast vTexSize \right) + (i, j) \right\rfloor / (uTexSize, vTexSize), \quad (i, j) \in \{0,1\}.
\]

Then, a bilinear interpolation between the \((u_f, v_f)^{i,j}\) samples is computed using the
fractional coordinates of \(u_f\) and \(v_f\) in the rectangle, \((u_f, v_f)^{i,j}\). A similar procedure
is used to sample the normal texture map, \(N\). The final deformed position, \(T(V_i)\), is
computed as

\[
V_i^d = T(V_i) = P(u_f, v_f) + N(u_f, v_f)z_i. \tag{5.2}
\]

Note the similarity of this computation of \(V_i^d\) to the original DDM computation, as
presented in Equation (5.1). Here, several vertices of \(D\) can be mapped above a single
location \((x_i, y_i)\) of the base surface eliminating the one-to-one mapping limitation of
displacement maps.

To properly render the deformed geometry, the new direction of the normal of each
mapped vertex \(T(V_i)\) should also be computed. Denote the original normal of \(V_i\) by \(\vec{N}_i\).
Denote \(\tilde{V}_i = V_i + \vec{N}_i \epsilon\). A simple approximation of the normal direction at \(T(V_i)\) could
be \(T(\tilde{V}_i) - T(V_i)\), for some small \(\epsilon \in \mathbb{R}^+\). Nevertheless, \(T\) is not conformal, in general,
and so this direct computation is likely to yield a vector that is not orthogonal to the
tangent space at \(T(V_i)\). Alternatively, we can deduce the correct normal by mapping
the complementary, tangent, space of \(D\) at \(V_i\). Compute two new independent vectors
\(\vec{T}_i^1, \vec{T}_i^2 \in \mathbb{R}^3\) that are orthogonal to \(\vec{N}_i\). \(\vec{T}_i^1\) and \(\vec{T}_i^2\) span the tangent space of \(D\) at \(V_i\).
Hence, \(T(\vec{T}_i^1)\) and \(T(\vec{T}_i^2)\) span the tangent space of the mapped geometry at \(T(V_i)\),
and can be approximated as $T(\vec{T}_j^i) = T(\tilde{V}_j^i) - T(V_i)$, $j = 1, 2$ where $\tilde{V}_j^i = V_i + \vec{T}_j^i \epsilon$. Then, the normal direction at $T(V_i)$ is merely $T(\vec{T}_1^i) \times T(\vec{T}_2^i)$.

The vectors $\vec{T}_1^i$, $\vec{T}_2^i$ are pre-computed. From these two vectors, the two points $\tilde{V}_j^i$, on the tangent plane, are derived and stored as part of the data of $V_i$, during the preprocessing phase. Then, during the interactive phase, $T(\tilde{V}_j^i)$ is computed in the vertex shader. With the values of $T(\tilde{V}_j^i)$ in hand, the normal direction at $T(V_i)$ is approximated as $(T(\tilde{V}_1^i) - T(V_i)) \times (T(\tilde{V}_2^i) - T(V_i))$. Thus, the cost of the computing the normal direction using this method triples the numbers of vertices that are mapped through $T$.

5.4 Extending the Basic HDDM

The HDDM operation is closely related to the traditional texture mapping scheme. It is, therefore, interesting to ask whether HDDM can handle features supported by traditional parametric texture mapping. It turns out that some properties are easily extended to HDDM while others are more complicated.

One such an extension of HDDM is the ability to use base surfaces that are polygonal, an extension that is explained in Section 5.4.1. Another important texture property is the ability to transform the texture in the texture space. Such transformations are key components in the implementation of texture animations. This feature is presented in the context of HDDM in Section 5.4.2.

5.4.1 HDDM over Polygonal Models

Until now, the HDDM algorithm was discussed in the context of mapping arbitrary geometric tiles over a freeform parametric base surface. However, in many cases, only polygonal models are available. To make HDDM more viable in real-time computer graphics applications, polygonal base surface geometry should be supported.
as well. In [34], an extension of the DDM that supports polygonal base surfaces is presented, and here we propose a similar extension for HDDM. For its proper operation, the HDDM algorithm requires a reliable parametrization, a problem that has been intensively investigated in recent years. Parametrization is required for the creation of the position and normal texture maps, $\mathcal{P}$ and $\mathcal{N}$. In this work, the parametrization that we used is taken from [88]; other parametrization methods could be equally employed. For a complete survey on parametrization methods see [41]. Given such a parametrization, the main difficulty in constructing the position, $\mathcal{P}(u, v)$, and unit normal, $\mathcal{N}(u, v)$, texture maps is resolved.

Consider a polygonal base surface, $S$, with a one-to-one parametrization to a planar $(u, v)$ domain. This $(u, v)$ domain is used to yield the 2D planar positions to render the polygons of $S$. The polygons of the base surface are rendered on a region orthogonal to the viewing direction, which spans the whole screen, and parameterized by the $(u, v)$ parametric coordinates. As before, the position and unit normal of $V_i \in S$ are mapped into the RGB color channels. The bilinear (color) interpolation mechanism of the graphics hardware will ensure the proper position and/or normal values between the vertices. Figures 5.5 (a) and (b) show the position and unit normal texture maps, respectively, for the camel model that is shown in Figure 5.9.

During the interactive phase, tiles are placed over the whole parametric domain of $\mathcal{P}$ (and $\mathcal{N}$), and the CPU renders the tiles as before. As long as the mapping process of $T(D)$ (Equation (5.2)) samples regions in the position and normal maps that are covered by valid values, the algorithm works without any change (see Figure 5.5, $D_2^P$ and $D_2^N$). Unfortunately, since there are invalid regions in $\mathcal{P}$ (and $\mathcal{N}$), problems arise where tiles cross into invalid regions of the map. Furthermore, tiles could span both valid and invalid regions (see Figures 5.5, $D_1^P$ and $D_1^N$), and vertices that are mapped to invalid regions of the map would not be mapped properly. One can alleviate some of these problems by clamping vertices that are mapped into invalid regions of the
Figure 5.5: The position $P$ (in (a)) and normal $N$ (in (b)) texture maps of the polygonal model of a camel (see Figure 5.9).

texture to the boundary of the valid region. The question then becomes, how do we clamp such vertices and in what directions. In this work, we decided to clamp vertices toward the nearest valid location.

The problem of finding the nearest valid location is similar to that of computing a distance transform on a grid, or the computation of the closest point to a planar curve, in the continuous domain. To approximate the distance transform, a diffusion process is applied to both maps, $P(u, v)$ and $N(u, v)$. The validity of a pixel is encoded into the alpha channel. Valid pixels are ignored, whereas invalid pixels with valid neighboring pixels are assigned the RGB color of their valid neighbor, in an eight-neighborhood topology. Hence, by repeating this process until the whole texture map is filled, each invalid pixel would receive valid position and normal values that are closest to it. The process is terminated when all the pixels are assigned a valid value from the original map. The effect of the position and normal value diffusion is that tiles that are partially embedded in the original map are clamped to the nearest
Nonetheless, this solution is still only partial. In places where two propagating fronts of the distance field collide, the maps could be discontinuous, and tiles might be arbitrarily stretched. This problem occurs in places where the radius of curvature of the boundary curve of the planar layout (Figures 5.5 (a) and (b)) is below the diffusion distance. Smaller tiles may alleviate this problem that is inherent in the computation of distance maps.

One can also employ a simple optimization at the CPU level during this interaction. Only tiles that intersect valid regions in the texture maps are rendered. Tiles that are totally outside the valid region are purged. To support this optimization, a two-dimensional bitmap that captures the valid parametric domain (such as the ones in Figures 5.5 (a) and (b)) is constructed during the initialization phase. Given the tiling configuration, the map is examined for each tile’s position. Tiles that do not contribute to the valid coverage are never sent to the GPU for rendering.

5.4.2 Texture Transformation and Animation

One type of animation that is used in conjunction with texture mapping is the transformation of a texture map in texture space. Using such transformations, animation of textures can be employed to mimic water flow, cloud movement and even rotating wheels. In regular texture mapping, the full range of homogeneous transformations is supported by today’s graphic cards. In the case of HDDM, and since non-overlapping crack-free tiling is usually desirable, linear and even homogeneous transformations of the tile in the parametric domain of $T$ are easily supported. Note that in the case of HDDM, transformations could also be applied to the normal direction of $S$, which is the $w$ direction of the HDDM.

Following [33], a more complex approach would use HDDM in the context of metamorphosis of geometry, herein the tiles’ geometry. A scalar function of time,
\( \alpha(t) \), is used to control the interpolation between two geometric tiles. A simple metamorphosis between two polygonal tiles \( D^1 \) and \( D^2 \) could be implemented in the vertex shader itself by creating \( D^1 \) and \( D^2 \) with a matching topology and the same number of triangles. Then, given two compatible vertices, \( V_i^1 \in D^1 \) and \( V_i^2 \in D^2 \), and the current time \( t \), the vertex shader computes the linear interpolation between the position and normal directions of the two vertices as,

\[
V_i = V_i^1 (1 - \alpha(t)) + V_i^2 \alpha(t),
\]

\[
\hat{\vec{N}}_i = \frac{\vec{N}_i}{||\vec{N}_i||}, \quad \vec{N}_i = \hat{\vec{N}}_i^1 (1 - \alpha(t)) + \hat{\vec{N}}_i^2 \alpha(t).
\]

(5.3)

While typically \( \alpha(t) \in [0, 1] \), it is not required to be in this range, resulting in an extrapolation of the given geometry.

### 5.5 Results

The DDM algorithm was implemented as a single vertex shader written in Cg [66]. The algorithm was integrated into an existing OpenGL-based viewer. All the images presented in Section 5.5 are screen snapshots from this viewer.

The timing statistics were gathered from two systems. The first is equipped with an Nvidia 6800 AGP graphics card; the second machine has an Nvidia 6600 GT e-PCI graphics card. The only publicly available timing statistics that could be found [45] suggest that about 33 million vertex texture sampling operations can be done per second on a faster card than the above two. If we consider a frame rate of ten frames per second or above as interactive, and since the computation of mapping each vertex and its normal requires 24 texture queries, then a practical bound on the size of models that can be handled is around 150,000 vertices. Table 5.1 shows the timing statistics of the algorithm on the two testing machines with different models and tiles using the simple normal estimation method. Some timing statistics suggest better
numbers than in [45]. One can only speculate that the high sampling locality in the HDDM algorithm played an important role here.

The HDDM algorithm requires a preprocessing phase. In this phase the position texture map, $\mathcal{P}(u, v)$, and normal texture map, $\mathcal{N}(u, v)$, are constructed and stored on the video memory of the graphics card. In the case where the model is a freeform NURBs surface, the construction time of two $256 \times 256$ position and normal texture maps took around a second. In the case of polygonal models, a diffusion process was employed. This process was implemented on the CPU and took around a second to completely construct the two $256 \times 256$ texture maps. Note that the evaluation time of the diffusion process depends on the size of the region that is covered (i.e., valid) by the original map.

Figure 5.6 shows a thorn tile (bottom right) used as the input to the HDDM. This tile contains 672 vertices. The thorn is not a function above the domain, and hence, such a tile cannot be used in displacement mapping.

As already stated, the HDDM algorithm is not restricted to simple geometric tiles,

<table>
<thead>
<tr>
<th>Scene (Figure)</th>
<th>Texture Tile (Figure)</th>
<th>Polygons Per Tile</th>
<th>Number of Tiles</th>
<th>Total # of Vertices</th>
<th>Fr/Sec Nv6800</th>
<th>Fr/Sec Nv6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Face (5.6)</td>
<td>Thorn</td>
<td>224</td>
<td>640</td>
<td>430080</td>
<td>8.5</td>
<td>8.1</td>
</tr>
<tr>
<td>Teapot (5.7)</td>
<td>Buddha</td>
<td>41868</td>
<td>6</td>
<td>753624</td>
<td>7.5</td>
<td>6.9</td>
</tr>
<tr>
<td>Teapot (5.8)</td>
<td>Thorn $\rightarrow$ Flower</td>
<td>224</td>
<td>480</td>
<td>322560</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Camel (5.9)</td>
<td>Chains</td>
<td>544</td>
<td>1789</td>
<td>2919648</td>
<td>2</td>
<td>1.3</td>
</tr>
<tr>
<td>Horse (5.10)</td>
<td>Hair</td>
<td>384</td>
<td>2802</td>
<td>3227904</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Human Face (5.11)</td>
<td>Hexagonal</td>
<td>804</td>
<td>400</td>
<td>964800</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.1: Timing statistics for the HDDM algorithm.
Figure 5.6: HDDM of thorns over a face model. In this example, there are $20 \times 32$ tiles; each tile (bottom right) contains 672 vertices. The normal and position texture maps contain $256 \times 256$ pixels each.

and virtually any geometry could be employed here as texture tiles. In Figure 5.7, a statue of Buddha (on the right) is placed and deformed, using HDDM, over the surface of the Utah teapot. This example again demonstrates the extended flexibility of HDDM over simple displacement mapping. The relatively high performance of this example could be explained by the fact that the CPU invokes only a few rendering calls in this case. Hence, less synchronization is needed between the CPU and the GPU. Here, four snapshots out of a real-time animated movie are presented.

Figure 5.8 presents another set of four snapshots from a metamorphosis animation sequence. The thorn is linearly interpolated into a flower, in a continuous manner. The animation is solely evaluated on the GPU. Conceptually, nothing prohibits the use of any available animation and/or metamorphosis method to animate the texture tiles’ geometry.
Figure 5.7: Four snapshots from a real-time animation sequence of a Buddha tile (right side), moving the tile in the \( UV \) space of the base surface. The Buddha model is used as a texture tile over the body of the Utah teapot. Six tiles are placed around the teapot.

Figure 5.8: Metamorphosis of texture tiles in HDDM. Thorn and flower tiles (right side) are linearly interpolated to generate in-between tiles that are placed using HDDM. Four snapshots from a real-time animation sequence are shown. \( 16 \times 32 \) tiles were used here, while the position and normal maps are of size \( 128 \times 128 \).

HDDM is not restricted to rational parametric surface-based models. Figures 5.9 and 5.10 show polygonal base surfaces, with parameterizations, used in HDDM. Figure 5.9 shows the model of a camel, which is rendered using chain-like texture tiles.
Again, such tiles cannot be used in displacement mapping due to the homeomorphism limitation. In Figure 5.10, a model of a horse is rendered normally but HDDM is used to supplement it with hair.

Nothing prevents us from using non-rectangular tiles. In the case of a hexagon or polygonal base surfaces. A chain-like tile (bottom right) is used to tile the surface of this polygonal model of a camel. In this model, there were originally $70 \times 70$ chain tiles, each with 1632 vertices. After pruning tiles that are mapped to invalid regions, we are left with 1789 tiles. The position and normal texture maps are each $256 \times 256$ in size.

Figure 5.9: HDDM can also support polygonal base surfaces. A chain-like tile (bottom right) is used to tile the surface of this polygonal model of a camel. In this model, there were originally $70 \times 70$ chain tiles, each with 1632 vertices. After pruning tiles that are mapped to invalid regions, we are left with 1789 tiles. The position and normal texture maps are each $256 \times 256$ in size.

Figure 5.10: HDDM can be used to supplement the model with non-covering, hair-like tiles (left). In this polygonal base surface, $80 \times 80$ hair tiles were originally used but the actual number of tiles that are rendered due to pruning is reduced to 2802 tiles. Each hair tile contains 8 threads and a total of 1152 vertices. The position and normal texture maps are each $128 \times 128$ in size.
a triangle, a single tile can also be used to regularly tile the entire parametric plain. In Figure 5.11, an equilateral hexagon is used to tile the parametric domain. The problem of tiling the plane with tiles with general boundaries was recently investigated in [56]. HDDM can be used in this context as well. Given an arbitrary tiling scheme of the plane, regular or even semi-regular, as long as proper offsets for the tile(s) could be derived, these offsets can be plugged directly into the HDDM framework to completely tile the base surface’s parametric domain.

HDDM is not restricted to tiling with a single tile. Multiple tiles with different geometries can be used as well. In Figure 5.12(a), multiple brick-like tiles are used (see Figure 5.12(d)) to generate the impression of brick wall. Additionally, regular

![Figure 5.11](image)

Figure 5.11: A human face mask tiled with hexagonal tiles (bottom right). The position and normal texture maps are each $128 \times 128$ in size.
textures are mapped on the tiles, to better capture high frequency details of a bricks. This example also demonstrates that HDDM can produce results that are similar to simple displacement mapping. In Figure 5.12(b), a non-homeomorphic tile (see 5.12(c)) is also used, demonstrating how complex texture geometry can be handled by the HDDM with similar ease.

HDDM is not restricted to simple animation of tiles in the texture space. Complex articulated objects can be animated as well. In Figure 5.13, an animated model of a walking robot is placed on top of the body of the Utah teapot. In this example, multiple robots are placed and deformed using HDDM.

5.6 Discussion and Future Work

This work has shown the applicability of the DDM algorithm in the context of real-time applications on modern programmable graphics hardware, hardware that is freely available for home PCs. The HDDM algorithm requires minimal integration effort.

Figure 5.12: Multiple brick-like tiles (d) are randomly placed on a model of a wall (a), showing that HDDM can reconstruct the results of traditional displacement mapping. In (b), a non-homeomorphic tile (c) is also used. In this example, the tiles are also texture mapped.
Figure 5.13: A snapshot (left) of an animation sequence of walking robots over the body of the Utah teapot. The right image shows a zoom-in on the handle of the teapot. This example shows that HDDM can be utilized to map and animate articulated models over curved surfaces.

and can be added into existing applications with very little main memory or CPU overhead. The original software-based DDM synthesized highly complex geometry with hundreds of thousands, if not millions, of polygons. Herein, the memory overhead is one detail tile $D$, typically around a thousand polygons, and two, typically small, texture images representing the positions and normals of the base surface, in the GPU memory space.

DDM, and now its real-time companion, HDDM, offer a major step forward in texturing and/or detailing a smooth surface, beyond displacement mapping. The proposed HDDM approach stores the positional and normal information of the base surface as texture maps while the geometric detail is stored once as true geometry and rendered repeatedly. These reversed roles of geometry and texture images not only allow the synthesis of new geometry detail that is not homeomorphic to the base
surface, but are also the key to the reduced load on the graphics bus, main memory and CPU. Finally, these reversed roles also alleviate the difficulty from which regular displacement mapping suffers, having to coerce the unnecessary refinement of the base surface to high precision in order to achieve satisfactory displacement results.

As with every other algorithm, HDDM has its limitations. First, the lack of bilinear texture mapping of 32-bit floating point textures in current GPUs forces the algorithm to make multiple texture queries into the position and normal maps for every vertex in $D$. These queries are expensive and limit the real-time performance of the algorithm. One can predict that future GPUs may support bilinear sampling of 32-bit texture directly. Second, the algorithm is also somewhat restricted in the types of parameterizations that can be used. Specifically, there should be a one-to-one mapping between the parametric domain and the base surface, meaning that texture coordinates should be unique within the base surface. Adapting the algorithm to more flexible parameterizations of the base surface should be approached in the future.

Since the performance of the HDDM algorithm is largely governed by the cost of sampling the position and normal textures, some simple optimizations could have been used. First, tiles that are back-facing and far from the silhouette areas can be purged, if the base surface and the detail texture are both closed. This can be done by inspecting the inner-product, $\langle \vec{n}, \vec{V} \rangle$, of the normal of the base surface and the viewing direction. Another optimization that should be considered is integrating an occlusion culling mechanism, possibly as an early rendering phase of the tiles’ bounding boxes, to eliminate tiles that do not contribute to the rendered image.

A different direction that can be considered in order to reduce the cost of the HDDM is multi-resolution. A straightforward approach would store, in the graphics card’s memory, several versions of the same tile, each being a refinement of the previous one. During a first rendering pass, the bounding box of each placed tile is
rendered. Then, the number of rendered pixels could be used to compute a view-dependent measure on the proper level-of-detail for each tile.
Chapter 6

Discontinuous Free-Form Deformation

This chapter includes the paper "Discontinuous Freeform Deformation". The paper was presented in the 12th Pacific Conference on Graphics and Applications (PG) [82].
6.1 Introduction

The deformation of geometric models has become a fundamental modeling tool in computer graphics. Following the terminology of [68], *Model Deformation* denotes the application of a non-linear transformation $F(u, v, w) : \mathbb{R}^3 \Rightarrow \mathbb{R}^3$ to a geometric model. Applying the deformation to an existing model results in a new, altered version.

There are many available deformation techniques. Some modify selected parts of a model by applying local deformations while others operate on the whole model at once and are denoted as global deformation methods. Among the global methods, one of the most widely-accepted techniques is Free Form Deformation (FFD) [85]. Here, trivariate functions, usually in the form of tensor product Bézier or B-spline functions, are used to map a box into a contorted box in $\mathbb{R}^3$. Then, an application of this FFD function to a geometric model results in a deformed version of the model. FFD is usually applied to polygonal models but it is not restricted to them. Its application to free-form models is also possible [85]. In this work however, we only consider triangular meshes.

Most deformation approaches alter the geometry of the deformed object without affecting its topology. This can become restrictive when a designer wishes to incorporate holes or tears into an existing model. A deformation tool that can potentially modify the topology of a model as part of the deformation specification will offer diverse applications as a modeling tool.

This work proposes a new FFD variant, coined *Discontinuous FFD* (DFFD). DFFD will account for the needed discontinuities and deform the model properly while automatically allowing it to split and re-form at the proper locations.

The DFFD operation is demonstrated using two different applications. The first application splits a deformable object and then wraps it around an obstacle(s) in the scene, splitting the deformable object and then re-forming it behind the obstacle(s).
The second application demonstrates DFFD as a general framework for inserting cuts into the surfaces of geometric models, in real-time.

The rest of the work is organized as follows. In Section 6.2, we survey related work on deformation functions and geometric modeling of cuts. In Section 6.3, we present an overview of the proposed approach. Section 6.3.1 details the construction steps of a discontinuous deformation function. Section 6.3.2 defines two approaches to closing holes in a mapped model that result from the application of the DFFD operation. In Section 6.4, we examine the use of DFFDs as a generic cutting tool. Section 6.5 demonstrates some results that were achieved by using DFFDs and finally, in Section 6.6, we conclude and consider some future directions for the proposed method.

6.2 Related Work

Throughout the years, researchers have devised many competing deformation methodologies. Such methodologies can be classified according to the locality of their influence, the types of input that can be handled by the deformation method and whether the deformations are physically based or purely geometrical in nature. Local deformation methods such as [59, 90, 61] are usually found to provide designers with finer control over the shape of the deformed object than global methods. Global deformation methods are more suitable for specifying large scale deformation over their source model.

Among the global deformation techniques, Free Form Deformations (FFD) [85] is probably the most widely accepted general deformation tool. FFD was first suggested by Sederberg and Parry as a mapping, \( \mathbb{R}^3 \rightarrow \mathbb{R}^3 \), based on a trivariate tensor-product Bézier function. Later work by Griessmair and Purgathofer [49] suggested a B-spline representation for better localized control over the deformation process. This work was later generalized to NURBS-based FFDs by Lamousin and Waggenspack [58].
A significant shortcoming of FFD stems from the fact that it maps a box-shaped domain into a contorted-box in Euclidean space. When used to deform objects of complex geometry, the deformation function only remotely resembles the deformed object. As a result, it becomes even harder to control the final shape of the deformed object by manipulating the control points of the deformation function. To improve that shortcoming, the Extended FFD (EFFD) by Coquillart [26] combined multiple FFD volumes to construct a single deformation function that better resembles the shape of the deformed model. Later, MacCracken and Joy [64] suggested the use of arbitrary topology FFD based on subdivision volumes for free-form deformation. The method improves the localization of the deformation since the deformation mesh approximates the deformed model at the expense of higher computational complexity.

Most of the existing deformation techniques preserve the topology of the deformed object. This can be explained by the fact that common design paradigms are modeled after operations that do not change the topology of the object (bending, twisting, stretching, skewing and more). However, such deformations also prevent the introduction of holes or gaps into the models during the deformation process. Recently, Steyn and Gain [92] suggested a deformation scheme that tries to address this issue. [92] starts by extruding the three-dimensional (3D) object into a four-dimensional (4D) space followed by an application of a deformation operation on the 4D object. The extrusion is performed by making copies of the 3D object and stacking these copies as layers of a 4D object. Respective vertices of the model layers are connected, forming a 4D object comprising prismatic-shaped primitives that are later subdivided into tetrahedrons. Deformation control is achieved by using Direct Manipulation FFD [52] on the 4D object. The deformed 4D object is then intersected with a hyperplane in order to extract the deformed 3D object. The main shortcoming of this approach hinges on the fact that it requires the manipulation of objects in a 4D space. Such a manipulation may be counter-intuitive for designers who live and work in normal 3D
A completely different approach, which can be viewed as topological alteration of deformed models, was presented by Chen et al. [17]. [17] suggest the use of transfer functions as tools that prescribe spatial modifications on a volumetric dataset. Transfer functions are usually used to define a mapping between two different domains, such as heat and color. [17] extend this notion by suggesting transfer functions that map between locations in space, \( \mathbb{R}^3 \Rightarrow \mathbb{R}^3 \). Such transfer functions are denoted Spatial Transfer Functions (STF). STF are applied to volumetric models, resulting in non-structured volumetric models that may later be sampled on a structured grid of voxels and rendered using standard volume visualization techniques. By properly designing the STF, the user can introduce holes into an original solid volumetric model.

While the introduction of feature lines as part of the surface modeling process is of major importance in shape design, not many works are concentrated on the use of \( C^{-1} \) for design. Ellens and Cohen [36] suggested teared surfaces as a scheme for the incorporation of non-iso-parametric lines of \( C^{-1} \) discontinuity inside B-spline surfaces. The crux of their work was the localization of the influence of B-spline control points with the addition of a local overlapping layers of control points to accommodate a \( C^{-1} \) discontinuity. Points in the parametric domain are tagged as belonging to either side of the discontinuity curve. The evaluation of a teared surface is done by selecting the proper layer of control points.

Cutting through meshes is a common operation in the field of surgical simulation. Most systems employ a complex collision detection scheme and, at times, a haptic feedback mechanism. In the rest of this overview, we focus mostly on the geometric part of these efforts, namely the parts that deal with cutting through meshes.

Bielser et al. [6] described a procedure that cuts tetrahedral meshes. One problem with their approach, as well as with every other subdivision-based approach, is that the complexity of the mesh increases as the cutting operation progresses. To
To prevent this problem, Neinhuys et al. [71] proposed locally aligning the mesh to the curve that the virtual scalpel traces. This approach alleviates the increase in the size of the mesh but it also suffers from some drawbacks. First, the face alignment procedure can produce zero-volume tetrahedras that get in the way of the Finite Element Analysis (FEA) deformation engine. Second, if textures are mapped over the model, this process may cause undesirable stretches. Finally, if the scalpel is cutting through fine model details, those details tend to vanish since the mesh near them is altered. To combat the creation of degeneracies in the model, Neinhuys and Van der Stappen [72] combined mesh cutting with a Delaunay-based triangulation approach. Local edge-flip operations are used on the faces that are affected by the cutting operation. The result of employing edge-flips is the elimination of triangles with large circumference. [72] consider surface meshes only.

6.3 The Discontinuous Deformation Algorithm

The DFFD algorithm operates in two phases. The first phase constructs a discontinuous deformation function. The selection of B-spline functions as the FFD tool lets us exploit a rich set of modeling techniques that are readily available for surface modeling. For the task at hand, knot insertion [22] will be used to introduce a potential iso-parametric $C^{-1}$ discontinuity into a tensor product B-spline FFD. The proposed algorithm automatically manipulates the control points of the deformation function to allow the deformation of the geometric model around the potential discontinuity and the creation of gaps or tears in the model. The two sides of the iso-parametric $C^{-1}$ discontinuity retain their former spatial continuity in every place but the regions of influence of the modified control points. This easily ensures the continuity of the deformed model everywhere but for the openings.

A deformation operation is a mapping $\mathbb{R}^3 \Rightarrow \mathbb{R}^3$. In most cases the deformation is applied to polygonal models that discretely approximate a free-form object. The
vertices of the model are deformed while the edges linearly connect these deformed vertices. Thus, edges and faces that cross the discontinuity are oblivious to the potential presence of that discontinuity and, consequently, do not reflect the needed topological change. To overcome this problem, the proposed algorithm incorporates an automatic model-split operation that splits the edges and faces that cross all discontinuities in the DFFD. In Section 6.3.1, we discuss the construction of the discontinuous deformation function and the automatic split operation.

**Definition 5.** A hole is introduced into a valid two-manifold geometric model resulting in an increase of the genus of the model while retaining its two-manifold validity. An opening in a geometric model introduces a boundary-cut that transforms it from a valid two-manifold geometric model into a non-manifold geometric model.

One undesirable result of our model-split operation is the introduction of openings into the model. Openings can be undesirable for applications that require closed, two-manifold, valid models. To seam such openings, two automatic approaches are presented. The first approach applies Boolean operations to the model before the deformation is applied in order to identify and fill the opening. Since Boolean operations are not always tractable, a second method that stitches openings is presented. This second method requires an additional manipulation of the deformation function and may yield a noticeable stretching effect in the deformed model. Section 6.3.2 elaborates on these two options for seaming an opening in a model.
6.3.1 Construction of the Discontinuous Deformation Function

As described above, a trivariate tensor product B-spline function serves here as the deformation function:

$$F(u, v, w) = \sum_{i=0}^{l} \sum_{j=0}^{m} \sum_{k=0}^{n} P_{ijk} B^o_i(u) B^o_j(v) B^o_k(w),$$

(6.1)

$$(u, v, w) \in [U_{\min}, U_{\max}] \times [V_{\min}, V_{\max}] \times [W_{\min}, W_{\max}],$$

where $P_{ijk}$ are the control points and $B^o_i(u)$ are the univariate basis functions of order $o$, in all three directions. In this application, we initially use uniform knot sequences in all directions, which complete the initial definition of a B-spline function. This function is continuous with a continuity that is governed by $o$, the order of the basis functions.

The insertion of $o$ knots into a knot sequence of curve $C(t)$ of order $o$ at $t_0$ would result in the interpolation of two different control points at $C(t_0)$, which, for clarity, are referred to as $C(t^-_0)$ and $C(t^+_0)$, accommodating a potential $C^{-1}$ discontinuity. The same knot insertion algorithm is directly applicable to any of the three parametric directions of the trivariate tensor product B-spline function $F$ in Equation (6.1). The result of inserting $o$ knots at a parameter $u_{\min} < u_0 < u_{\max}$ introduces two identical and adjacent control mesh planes along a potential $C^{-1}$ discontinuity. In the ensuing discussion, we assume that the knots are inserted in the $u$ parametric direction of $F(u,v,w)$.

**Definition 6.** A potential $C^{-1}$ discontinuity due to the insertion of $o$ knots into the parametric domain of a trivariate B-spline function $F$ of order $o$ at $u = u_0$ defines two potential discontinuity planes. These planes are denoted as $F^- = F(u^-_0, v, w)$ and $F^+ = F(u^+_0, v, w)$, respectively.

The newly introduced discontinuity planes of control points, which are inserted into the deformation function, and the control points in their immediate vicinity, will
provide the extra degree of flexibility that is needed in order to wrap the space around an obstacle.

In order to define the DFFD function, the algorithm starts by constructing a trivariate B-spline function that spans somewhat beyond all the obstacles in the scene. The obstacles are static objects around which we wish to split and wrap a moving deformable object. The algorithm uses the axis-aligned bounding box, \( B = [x_{\text{min}}, x_{\text{max}}], [y_{\text{min}}, y_{\text{max}}], [z_{\text{min}}, z_{\text{max}}] \), of these obstacles to deduce the dimensions of the scene. The parametric domain of the DFFD function is then prescribed to follow \( B \). In order to fully determine the deformation function, \( F \), the user supplies the order, \( o \), of the basis functions and the number of control points in each direction, \( l, m \) and \( n \). Uniformly-spaced knot sequences in each parametric direction complete the specification of \( F \). Then, the control points of the deformation function, \( P_{ijk} \), are placed uniformly inside \( B \).

The process of building up the DFFD function continues by inserting potential \( C^{-1} \) discontinuities into the initial trivariate function, \( F \). Let the center of mass of the \( i^{th} \) obstacle, \( O_i \), in the \( u \) parametric direction be \( u_{0i}^o \) (see Figure 6.1). \( o \) knots are inserted at \( u_{0i}^o \), resulting in a potential \( C^{-1} \) discontinuity at that parameter. Control points that are associated with the \( u = u_{0i}^o \) parameter, or are in close proximity, are relocated in order to avoid obstacle \( O_i \). In other words, we seek to modify \( F(u,v,w) \) so that its range will have an empty intersection with the obstacles, \( F(u,v,w) \cap O_i = \emptyset, \forall i \).

In order to achieve this object avoidance, \( O_i \) is intersected with a series of \( w \)-parallel planes \( W_k : w = w_k, 0 \leq k < n \). For each \( W_k \), the set of intersection points, \( W_k \cap O_i \), is then used to construct two boundary curves, \( C^+(v) \) and \( C^-(v) \), which wrap \( O_i \) at plane \( W_k \) (see Figure 6.1). Each pair of control points, \( P_{ijk} \), from \( F^- \) and \( F^+ \) along the \( C^{-1} \) discontinuity, are projected onto \( C^-(v) \) and \( C^+(v) \), respectively, resulting in a gap in the range of \( F \) that approximates the shape of \( O_i \).
The projection procedure, though simple to implement and comprehend, also possesses some limitations. A control point, \( P_{ijk} \), on the discontinuity plane, could be projected over adjacent control points, resulting in self-intersections in the deformation function. Self-intersections in the deformation function are undesirable and may lead to self-intersections in the deformed model (see [43]). Denote by \( P_{ijk}^u \) the component of the control point \( P_{ijk} \) and by \( V_{ijk}^{u\pm} \) the computed translation vector for control point \( P_{ijk}^\pm \in F^\pm \) along the \( u \) axis. The influence of \( V_{ijk}^{u\pm} \) is propagated to neighboring vertices such that \( P_{qjk}^\pm = P_{ijk}^\pm + \kappa(P_{ijk}, P_{ijk})V_{ijk}^{u\pm} \), where \( \kappa(P_{qjk}, P_{ijk}) \) is a finite kernel filter that decays from one to zero as \( P_{qjk} \) moves away from \( P_{ijk} \). In this work, we used a truncated Gaussian function and a quadratic B-spline function for \( \kappa(\cdot, \cdot) \). Proper specification of \( \kappa(\cdot, \cdot) \) is needed to prevent the introduction of self-intersections into the deformation function, on the one hand, and excessively wide deformations, on the other. In this application the kernel parameters, such as the variance and cut-off level in the case of the Gaussian kernel and the shape of the B-spline filter, are controlled by the user.

As described in [43], the presence of zeros in the Jacobian of \( F, J(F) = < \frac{\partial F}{\partial u} \times \frac{\partial F}{\partial v}, \frac{\partial F}{\partial w} > \), signals when the application of a kernel function, \( \kappa(\cdot, \cdot) \), introduces self intersections into \( F \). Let \( F \) be a trivariate tensor product function with control points \( P_{ijk} \) that are uniformly spread over a 3D box. Then,

**Theorem 6.** Given \( F \), a trivariate tensor product B-spline function with uniformly distributed control points along a 3D box \( P_{ijk} \), such that \( P_{i+1,j,k}^u > P_{i,j,k}^u \) and \( \| J(F) \| > 0 \). Assume we move control points along the \( u \) axis such that \( P_{i+1,j,k}^u > P_{i,j,k}^u \) is preserved \( \forall i, j, k \) after the application of \( \kappa(\cdot, \cdot) \) to \( F \). Then \( \| J(F) \| > 0 \) still holds.

**Proof.** \( \frac{\partial F}{\partial u} \) continues to point to the \(+u\) direction and \( \frac{\partial F}{\partial v} \) retains its \(+v\) component. Hence, the projection of \( \frac{\partial F}{\partial u} \times \frac{\partial F}{\partial v} \) on the \( w \) axis never vanishes. Since all control points move orthogonally to the \( w \) direction, \( \frac{\partial F}{\partial w} \) retains its \(+w\) coefficient and the inner product \( < \frac{\partial F}{\partial u} \times \frac{\partial F}{\partial v}, \frac{\partial F}{\partial w} > \) remains positive. \( \square \)
Therefore, it is sufficient to test for the occurrence of fold-overs of control points in the \( u \) direction to identify the introduction of zeros into the Jacobian of \( F \).

This projection procedure limits the types of obstacles that can be properly embedded. Let \( P_{ijk} \) be a control point along the inserted \( C^{-1} \) discontinuity. Then, the projection operation of that point on the curve \( C^\pm(v) \) should be unique, which allows only obstacles that are monotonic relative to the \( v \) direction. In addition, one should recall that \( C^\pm(v) \) only approximates the shape of the obstacle. For objects with complex geometry such an approximation might be too crude, revealing gaps between the obstacle and the deformed object, or causing the deformed object to penetrate the obstacle. This problem may be alleviated by inserting more degrees of freedom into the deformation function.

Thus far, we have described the insertion of a single discontinuity into the DFFD function.

![Figure 6.1: The cross-section of obstacle \( O_i \) in plane \( W_k \), in the shape of a silhouette of a human face, guides the construction of the DFFD function. The two boundary curves, \( C^+(u) \) and \( C^-(u) \), on the cutting plane \( W_k \) are shown in red.](image)

109
function. Conceptually, the insertion of multiple discontinuities is similar; see Figures 6.10 and 6.11. However, since each control point in the control volume has a finite region of influence, discontinuities that are too closely packed in the $U$ direction will interfere with each other. To eliminate such interference, there should be enough degrees of freedom between each pair of obstacles, $O_i$ and $O_j$. The minimal number of required control points in the direction of the manipulation depends on the order of the deformation function, $o$, and on the support of the kernel function, $\kappa(\cdot, \cdot)$.

### 6.3.2 Splitting and Stitching the Model

The constructed deformation function, $F$, as presented in Section 6.3.1, when used to naively deform a polygonal model, $M$, does not achieve its goal. While the vertices of $M$ are properly mapped, the edges and faces of $M$ are not evaluated through the deformation function. Edges that connect vertices on opposite sides of the discontinuity would simply cross it; faces that share these edges would present the same problem. To prevent the phenomena from occurring, $M$ must be split at each discontinuity, $u = u_i, \forall i$, before the deformation takes place. The splitting operation is performed in the parametric domain of $F$, which is also the object space of $M$. Since the object resides in the parametric domain of $F$, the split operation is done relative to a flat iso-parametric surface. Denote by $E_{j,k} = V_j, V_k$ the edge connecting vertices $V_j$ and $V_k$, and let $v^u_j$ and $v^u_k$ be the $u$ coordinates of $V_j$ and $V_k$, respectively. Without loss of generality, assume $v^u_j < v^u_k$ and recall that $u_i$ is the splitting parameter. Then, if $v^u_j < u_i < v^u_k$, $E_{j,k}$ is a crossing edge that must be split. The result of the polygon splitting phase is a modified model, $\bar{M}$, which contains no edges that cross the discontinuity. As a result of the potential $C^{-1}$ discontinuity in $F$, as described in Section 6.3.1, points with $u \geq u_i$ would be mapped to one side of the discontinuity, while points with $u < u_i$ would be mapped to the opposite side. Because the above edge-splitting procedure introduces new points with $u = u_i$, the algorithm
moves these points to the proper side by adding (subtracting) a small amount, setting $u_i^\pm = u_i \pm \epsilon$.

Mapping $\bar{M}$ through $F$ will result in openings in the mapped model, $F(\bar{M})$, along the discontinuities of $F$ (see Figure 6.2(a)). In other words, as a result of the DFFD operation, a valid two-manifold model, $M$, will become a non-manifold one, with openings with boundaries. This behavior may not be desirable in many cases and ways to close these openings must be sought. We propose two approaches to close these openings. The first one employs Boolean operations [65, 54]. Let $M_F = F(M)$ be the mapped model and let $\mathcal{P}^+$ be an infinite cutting plane defined by the split parameter $u_i$ and capturing the half space $u \geq u_i$. Then, $M^+ = F(M \ast \mathcal{P}^+)$ and $M^- = F(M - \mathcal{P}^+)$ define the two split parts of $M_F$, where $\ast$ and $-$ represent the intersection and subtraction Boolean operations, respectively. If $M$ is a valid two-manifold, these operations are well defined. $M^+$ and $M^-$ are two closed objects that are fully contained in domains $[u_{\text{min}}, u_i)$ and $[u_i, u_{\text{max}})$, respectively. Hence, $F(M^+)$ and $F(M^-)$ are two closed objects at the opposite sides of the discontinuity, with added caps at the intersection. See Figure 6.2(b) for an example of an application of this method to openings in a model.

Figure 6.2: A non-trivial shape, the silhouette of a human face, (see also Figure 6.1) is molded into the center of a simple mechanical part using a DFFD operation. In (a), the edges of the opening are left open, while in (b), Boolean operations are used to cap the introduced opening.
A potential difficulty in the general use of Boolean operations is that many polygonal models are not proper two-manifolds, for which Boolean operations are not applicable. Moreover, the caps that Boolean operations create for seaming the openings are only $C^0$-continuous. For a $C^1$ smooth object, $M$, such low continuity might be insufficient. Thus, a second option that offers better continuity, while assuming nothing about the mapped object $M$ but demands an additional manipulation of the trivariate deformation function, is suggested.

In order to close the openings that are formed in a model as a result of the DFFD operation, the deformation function can also be further manipulated. Modify $F$ such that $F(u_i, v, w) = F(u_i, v, w_s), \forall w$, where $w_s$ serves as the $w$-stitching parametric level. As a result, all control points of $F$ interpolating the discontinuity plane at $u = u_i$ would be stitched together at a single parametric level $w = w_s$. $w_s$ could be derived in several manners, and currently, the algorithm selects $w_s$ to be the average $w$-value of all the vertices, $V_i$, with $v^u_i = u_i$, in $\bar{M}$. To complete the stitching operation, the control points of $F$, $P_{ijk} = (u_i, v_j, w_k)$ on the two discontinuity planes are moved to $P_{ijk} = (u_i, v_j, w_s), \forall j, k$. Figure 6.3 presents the result of stitching shut a rounded hole in a DFFD trivariate function. A properly stitched DFFD function,

![Figure 6.3](image)

Figure 6.3: In (a), a modified trivariate function, $F$, which has a circular hole in it, is stitched together, closing the opening. The stitching operation preserves a closed mapped object, $F(\bar{M})$. In (b), a side-view of $F$ is shown.
$F$, has the property that every application of $F$ to $\bar{M}$ will preserve the two-manifold property in $F(\bar{M})$, if $\bar{M}$ is a two-manifold geometric object. No new openings are introduced into $F(\bar{M})$ and the openings along the discontinuity $u = u_i$ are stitched together into a closed seam.

$C^\pm(v)$ are $C^0$ continuous in two locations, at the minimal and maximal $v$ values of $O_i$, $O_i^{\min(v)}$ and $O_i^{\max(v)}$ (see Figure 6.1). Projecting control points $P_{ijk}$ onto $C^\pm(v)$ is insufficient to capture this low continuity in $C^\pm(v)$. Hence, we insert $o - 1$ knots in the $v$ parametric direction at parameters $v_i^{\text{max}}$ and $v_i^{\text{min}}$ corresponding to the two $C^0$ locations $O_i^{\min(v)}$ and $O_i^{\max(v)}$, respectively. Figure 6.4 shows the difference with and without an interpolation constraint in the $v$ direction of the DFFD function at $O_i^{\min(v)}$ and $O_i^{\max(v)}$.

### 6.4 Local DFFD

Section 6.3 described the construction and manipulation of the DFFD algorithm from the point of view of a global deformation algorithm. Alternatively, the DFFD

![Figure 6.4: A model of a sphere is squashed in the vertical direction and wrapped around a cylindrical obstacle in front. In (a) no interpolation constraint is imposed on the control points in the $v$ direction, which results in an opening in the middle of the model. (b) shows the effect of inserting $o - 1$ knots in the $v$ direction at $O_i^{\max(v)}$, thereby closing the gap.](image)

Figure 6.4: A model of a sphere is squashed in the vertical direction and wrapped around a cylindrical obstacle in front. In (a) no interpolation constraint is imposed on the control points in the $v$ direction, which results in an opening in the middle of the model. (b) shows the effect of inserting $o - 1$ knots in the $v$ direction at $O_i^{\max(v)}$, thereby closing the gap.
algorithm can also be treated as a local mesh-splitting algorithm that cuts the surface of polygonal meshes while properly handling the shape and deepness of the cut and preserving its two-manifold property. The algorithm models the shape and deepness of the cut produced by the shape of the cutting tool. Moreover, by properly designing the DFFD function, the algorithm can simulate the time-response of the deformable object in the vicinity of the cut.

The local DFFD cutting operation comprises three steps. First, the end-user defines the region that should be influenced by the cutting operation and constructs a DFFD function. Then, the algorithm identifies the geometry that is embedded in the spatial support of the DFFD function. Assuming a polygonal geometry, we need to locally parameterize the vertices of these polygons relative to the range of the DFFD function. Finally, the DFFD should be evaluated by computing the deformation of the target surface polygons, splitting polygons that cross the discontinuities of the DFFD function and adding the polygons that define the deepness of the cut.

To define the DFFD function, we locate the generator curve on the surfaces of the geometry to be cut. A series of mouse-recorded hit points on the surface is used to

![Figure 6.5: A cross-section of a local DFFD function placed over a cylindrical surface.](image)

The direction of the scalpel $G(v)$ is oriented into the page. (a) shows the DFFD function prior to the insertion of discontinuities, while (b) shows the same function after the introduction of the discontinuity cut; see also Figure 6.6.
Figure 6.6: In (a), a general view of a local DFFD function is shown. The two red iso-parametric surfaces depict the cut. (b) presents the iso-surface $F(u, 0.5, w)$.

construct a B-spline curve that approximates the generator curve, $G(v)$, in a least-squares sense. From $G(v)$ we also derive the tangent curve, $T(v) = \frac{dG(v)}{dv}$. Moreover, from the surface normals at the hit points we construct a normal curve, $N(v)$ (see Figure 6.5). Denote by $G_i$ a sampled position along $G(v)$, and by $T_i$ and $N_i$ the normalized unit vectors sampled along $T(v)$ and $N(v)$, respectively; let $S_i = T_i \times N_i$. Then, the control points $P_{ijk}$ of the DFFD trivariate function $F$ are defined as

$$P_{ijk} = G_j + \left( \left\lfloor \frac{l}{2} \right\rfloor - i \right) \alpha S_i + \left( \left\lfloor \frac{n}{2} \right\rfloor - k \right) \beta N_i,$$

(6.2)

where $l$, $m$ and $n$ follow the notation of Equation (6.1), and $\alpha$ and $\beta$ are two user-defined offset-amount scaling factors. Equation (6.2) positions the trivariate deformation function such that the generator curve, $G(v)$, is approximately equidistant from the outer envelope of the DFFD function, $F$, at $F(\frac{1}{2}, v, \frac{1}{2})$, assuming $[0, 1]$ domain in $u$ and $w$. In a similar way to the construction of $F$ in Section 6.3.1, $\omega$ knots are inserted in the middle of the $u$ parametric axis of $F$, resulting in a potential $C^{-1}$ discontinuity.
Since the DFFD construction scheme generates curved-shaped deformation functions, a different algorithm is used to model the shape of the discontinuity. To form the cut, the control points that reside near and on the discontinuity, $P_{ijk}^\pm$ for $i = \lfloor \frac{l}{2} \rfloor$, are moved along the $u$ parametric axis. The amount of movement is controlled by the kernel function, $\kappa(P_{ijk}^\pm, P_{\lfloor \frac{l}{2} \rfloor jk}^\pm)$; see, for example, Figure 6.6.

The next step of the cutting algorithm is the detection of polygons in the cut geometry that are contained in the range of the deformation function. Here, we select all the polygons that were traversed by the generator curve as well as polygons with close proximity whose vertices are contained in the spatial domain of $F$. Vertices that are contained in the domain of the deformation function are selected and parameterized by their local coordinates inside the volume of $F$. Let $V_i$ be a vertex on the discontinuity. We parameterize $V_i$ such that $u = \frac{1}{2}$, $v = \arg \min_v \|G(v) - V_i\|$, and $w = \frac{1}{2}$. Edges and faces that cross the potential discontinuity are split and the deformation function is evaluated (as in Section 6.3.1).

As was described in Section 6.3.2, we should maintain the two-manifold validity of the model in the presence of cutting operations. Here, we can use the assumption that our cutting operation is local, which means that the cutting tool always cuts a single patch of the input geometry. This assumption lets us construct a new geometry for the deepness of the cut, in the shape of the cutting tool. An evaluation of the original and new geometries through the DFFD function completes the construction of the deepness of the cut, and also guarantees the validity of the surface. The introduction of new geometry to close the opening can be further viewed as another option to seaming the opening that is formed during the DFFD operation. We presented this method in another paper [87] that focused on the applications of DFFD as a cutting framework for surgical simulations.
6.5 Results

We present some results of applying DFFD operations to various models. Figure 6.7(a) shows the application of the embedding algorithm to the Stanford bunny model with a cylindrical shaped tool entering from above. The opening in the model that results is clearly visible. In Figure 6.7(b), the stitching algorithm was applied to the DFFD function and then the deformation was applied to the model, resulting in a valid two-manifold object.

Figure 6.8 demonstrates the use of local DFFD operations for real-time cutting operations. The shape of the DFFD is designed to produce an incision in a model of a face. Each incision on the face corresponds to an application of a local DFFD operation. The deepness of the cut is modeled and textured to replicate the influence of a cut to the model.

Figure 6.7: A cylindrical tool slicing through the Stanford bunny model. In (a), the model was left open so that its inside is exposed. In (b), the DFFD’s stitching method is used to close the opening.
Figure 6.8: The local DFFD method was used to introduce several cuts into the cheek and nose. The inner part of the cut is shown, with flesh-like texture.

Cases exist where the stretching introduced by the stitching method are significant. If the mapped model, $M$, is a two-manifold, Boolean operations can be used to generate caps over the holes that are introduced into the model by the DFFD operation. Figure 6.9 shows several snapshots of an animation sequence of a moving bulldozer. A redwood tree splits open to admit the moving tractor; the split is shown as a gradually widening gap. Boolean operations were used to define the caps in the interior of the tree, which are textured with a wood-like texture. In Figure 6.10, an animation sequence of a walking robot is shown. Multiple discontinuities are introduced into the DFFD function, one for each bar. The robot simultaneously passes through several vertical parallel bars. When the robot reaches the bars, it splits apart, only to re-form beyond the bars. In this case, neither the DFFD stitching technique nor the Boolean operation method was used. As a result, the robot’s interior is exposed during the split.

Figure 6.11 demonstrates that DFFD is independent of the orientation of the model. The robot model is placed in three different orientations relative to three semi-transparent parallel bars.
Figure 6.9: Snapshots from an animation sequence of a bulldozer moving through a forming hole in a tree. Boolean operations were used to compute the interior of the hole, which is also wood-textured.

Figure 6.10: Snapshots from an animation sequence of a robot walking through vertical bars. The model was left open to reveal its inner parts. In this example, the bars that were originally used to generate the opening were replaced by slimmer bars to better expose the splitting effect.

The DFFD operation is fairly cheap in terms of execution time. A DFFD function with several thousand control points can be constructed at interactive rates, regardless of the complexity of any deformed model. The evaluation time of any model is linear in the number of vertices. For example, a single deformation of the model of the robot in Figure 6.10, which contains about 32000 vertices, took about one second on a Pentium™4 PC with 512 MB of RAM. For local DFFD (Section 6.4), both the construction rates of the DFFD function and the deformation time pose no limit on the real-time characteristics of the system.
6.6 Discussion and Conclusions

We have presented an algorithm for modeling cuts and openings in general meshes. The method builds upon a variant of FFD that supports the introduction of $C^{-1}$ discontinuities into a deformation function. The algorithm employs standard knot insertions, followed by a manipulation of the control volume of the deformation function. To achieve a deformation that adheres to the discontinuities in the deformation function, a model-split algorithm is also defined. The deformed model is first split in the parametric domain of $F$. Since the inputs to the algorithm are surface-based meshes, we also present the means to close or stitch the openings that are introduced into the model as a result of applying the DFFD operation.

The DFFD operation has been demonstrated using two applications. The first automatically wraps deformable objects around obstacles that are placed in a scene. Several animation sequences that demonstrate the applicability of the proposed method are shown. The second defines a local DFFD function that cuts the surface in real-time. The embedded parts are parameterized relative to the DFFD volume and then
split and re-formed, resulting in an arbitrarily shaped cut. We foresee that surgical simulators could benefit from such cutting operations. We expect that a careful tissue-based analysis of the cutting models, such as FEA, could be encoded into the cutting DFFD to a user-defined precision. Then, a purely geometric deformation phase would cut into the material in real-time, modifying the shape of the cut to resemble the expected shape. Applying DFFD to local areas of a model has two direct benefits. First, since only part of the model undergoes a deformation, the algorithm can achieve interactive manipulation rates. Second, the local version of the algorithm lets the designer specify arbitrarily-shaped incisions, alleviating the limitation of iso-parametric cuts.

DFFD operates by inserting iso-parametric discontinuities into the deformation function. Nothing prevents us from composing multiple DFFD operations on a single model, each in a different direction. An anticipated problem with such a composition will be in places where discontinuity planes intersect. One simple solution would be to compute the effect of a DFFD with a single iso-parametric discontinuity over an input geometry and then feed the result into a second, rotated in space, DFFD function.

The presented algorithm also has some limitations. First, currently it can only handle iso-parametric cuts. This shortcoming is a direct result of using knot insertion to incorporate a potential $C^{-1}$ discontinuity. Enabling arbitrarily shaped discontinuity contours possibly as trimmed volumetric DFFD functions, would broaden the class of available cuts. Toward this end, intuitive ways should be developed to model the complex shape of the trimming surfaces. One approach that might present a feasible research direction is an extension to direct manipulation tools [89] to the problem of trimming surfaces design.

A second limitation of the algorithm results in the use of a projection procedure
to construct the DFFD function. Such a projection procedure can only handle obstacles that are monotone relative to the axis of movement, \( v \), which implies that each projected point might intersect \( C^\pm(v) \) exactly once. In order to broaden the class of objects that can be used for design of the DFFD function, other construction methods should be explored.

As presented, the algorithm cuts through surface-based meshes or models. In order to improve the usability of the algorithm for surgical simulations, we need to support cuts through volumetric meshes. Since our deformation model is volumetric by nature, we foresee no conceptual problem here. However, surgical simulations require the deformation engines to work in real-time.

Similarly, in order to make DFFDs into a more general design tool, the support for free-form rational geometries should be considered. Supporting free-form surface-based models could be added by supporting trimmed surface in the deformable models.
Chapter 7

Silhouette Extraction from Volume Data

This chapter includes the paper "Adaptive extraction and visualization of silhouette curves from volumetric datasets". The paper was published in the journal The Visual Computer [80], it was also presented in the 4th Israel-Korea Bi-National Conference on Computer Graphics and Applications.


7.1 Introduction

The field of volumetric rendering has been the focus of a lot of research activity in the last decade. The widespread use of Computerized Tomography (CT) and Magnetic Resonance Imaging (MRI) systems in today’s hospitals has contributed to an exponential growth in the amount of volumetric data that needs to be analyzed correctly, efficiently and promptly. With the growing popularity of these systems, physicians, trained to analyze large sets of two-dimensional X-ray images, have to develop new skills to cope with modern data representations.

In parallel to the work in volumetric rendering during the past ten years, research in the field of Non-Photorealistic Rendering (NPR) has also advanced vigorously. The term NPR covers a plethora of methods and techniques. Some serve to create paint-like effects while others are more focused on creating illustrative results. Illustrations are common in educational and engineering books, mechanical manuals and architectural blueprints where they efficiently convey the geometric structure of objects while filtering out the photorealistic impression. The resulting images are often more easily understood than photorealistic images. In order to generate an automatic illustration from a model or another photorealistic image, the regions of the model that convey the geometry of the object to the viewer in the most clear way must be identified. Watching the way professional illustrators work, it can be seen that silhouettes, boundary lines and cusps are almost always used to achieve this.

Definition 7. A surface point $P$ of object $O$ is a silhouette point with respect to a viewing direction $V$ if $< V, N_p >= 0$, where $N_p$ is the normal vector of $O$ at $P$.

The two disciplines, volumetric rendering and NPR, are quite remote from one another. In volumetric rendering, the focus has always been on generating the most realistic results, whereas NPR tries to achieve either an aesthetic result, in the case of artistic NPR, or an informative result if a technical illustration is the goal. This
paper tries to use the NPR technique of silhouette rendering within the context of volumetric rendering. The generated silhouette can be used to enhance images generated by direct rendering methods or as part of an illustration package, to create a technical illustration of the volumetric data. Overlaying the silhouettes on top of a volumetric image might improve its readability. This overlaying idea is not new, as, for example, it was presented in [76].

In this paper, we present a novel approach that calculates silhouettes from volumetric data using implicit trivariate tensor product B-spline functions that model the data. A subdivision method is used to robustly and efficiently calculate the silhouettes that have a superior look compared to voxel based silhouette extraction schemes such as [63]. This paper is focused mainly on silhouette extraction but the presented method can be easily adapted to extract other features of interest, such as boundaries, cusps, parabolic lines or lines of curvature, from the volumetric data.

In Section 7.2, we review some previous results in the field of NPR and recognize the importance that silhouette curves play in this field. We also refer to previous work that developed some of the tools that are extensively used in this paper. In Section 7.3, we present the algorithm and data structures used in the calculation of the volumetric silhouettes. In Section 7.4, we present some of the results generated using the presented algorithm, and finally we conclude in Section 7.5.

7.2 Previous Work

Basic definitions and algorithms for direct volumetric rendering, as defined in [77] and [60], try to create images that reveal the inner structure of the volumetric data. [77] defines a visualization pipeline for direct volumetric rendering using a transfer function, while [60] models the behavior of light traveling through translucent materials with varying opacity levels. When looking at images generated by these methods the geometry of the inner structures is often hard to decipher.
Work on automatic illustration generation has focused mostly on the creation of illustrations from the boundary representation of objects. Such work can be roughly divided into work that are interested mainly in creating the most aesthetic products [47] and others that are more focused on the creation of an interactive illustration system [48, 67]. Gooch et al. [47] concentrated on a proper lighting model for creating technical illustrations. Markosian et al. [67] presented an approximated, probabilistic, approach for the extraction of silhouettes, while Gooch et al. [48] looks at creating a complete set of techniques for generating illustrations. Another possible distinction can be made between efforts that are focused on silhouette extraction for polygonal models [5, 11, 79] and work that deals with silhouette extraction from free-form models [99, 32]. Winkenbach and Salesin [99] adapted techniques for pen and ink illustration to free-form surfaces. Variations in stroke shape and different texture selection play a major role in the generation of the surface illustration. A planar map of the free-form surface model is used for the extraction of silhouettes. Benichou and Elber [5] demonstrates the usage of a Gauss map as a data structure for speeding up the query time for a silhouette edge in a polygonal model. In [79], a low resolution model is enhanced by clipping a high resolution silhouette onto it, thereby improving the overall quality of the generated image. The silhouette clipping is used to hide the jagged edges that are most visible on the silhouette of a polygonal model.

Illustrating surfaces in volumetric data was suggested by Interrante [53]. Differential features of the iso-surfaces, such as minimal and maximal curvature and principal directions, are calculated from the piece-wise constant volumetric dataset. These features were later used as the input for a three-dimensional Line Integral Convolution (LIC) algorithm. Different textures were used to influence the final illustration of the iso-surfaces. Silhouette extraction is not dealt with in the context of this work but the same approach could be used to extract and visualize the silhouette from the volumetric datasets. The term *volume illustration* was coined by Ebert and Rheingans in

126
to describe the use of NPR visualization techniques in the context of volumetric visualization. The main focus of [76] is on incorporating NPR techniques in the direct rendering pipeline, in order to achieve an illustration of the volume. [76] uses, among other techniques, silhouettes in the generation of the final volume illustration. It does not, however, describe any specific extraction algorithm. Treavet et al. [94] propose a system for artistic modeling and rendering of volumetric datasets. The volumetric datasets are represented as a rectangular lattice of samples and a set of tri-linear interpolation functions, and proposes a set of methods for incorporating artistic effects into the rendering pipeline. Although silhouette rendering is not explicitly mentioned, normal calculation, a major part in any silhouette extraction scheme, is done using a discreet approximation. Recent work, [63], advocates the use of stippling techniques for the generation of volumetric illustration at interactive rates. [63] renders areas of interest such as silhouettes or boundaries using a variable point density per positively classified voxel. The silhouettes presented in [63] are disconnected and inexact, due to the direct use of piecewise constant voxel-based data, a problem we completely circumvent herein. [28] used a linear regression method to approximate the gradient field of a volumetric dataset and generate the contour lines with a Maximum Intensity Projection (MIP) rendering scheme. [28], however, works with a fixed gradient resolution and thus may prove to be less accurate than the proposed algorithm.

### 7.3 Extracting the Silhouette Curves

Before presenting the silhouette extraction algorithm, a short explanation on data representation is provided in Section 7.3.1. We then present the silhouette extraction algorithm itself. The silhouette extraction algorithm of volumetric datasets works in three phases. First, a preprocessing phase which is presented in Section 7.3.2, second, a silhouette extraction phase, described in Section 7.3.3, and a third, silhouette rendering phase, described in Section 7.3.4.
7.3.1 Data representation

Datasets that are generated by any of the modalities (CT, MRI, SEM) can be treated as a three-dimensional structured grid with, typically, scalar values at the points of the grid. The scalars can, for example, be interpreted as the physical density levels of the inspected material. These scalars are used as control points of a trivariate tensor product B-spline function, $D : \mathbb{R}^3 \rightarrow \mathbb{R}$ s.t.

$$D(u, v, w) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{l-1} P_{ijk} B_i(u) B_j(v) B_k(w),$$

(7.1)

where $B_i(u)$, $B_j(v)$, and $B_k(w)$ are the B-spline basis functions and $P_{ijk}$ are scalar values at locations $(i, j, k)$ of a grid of size $n \times m \times l$. Further, the iso-surface at level $D_0$ is then equal to $D(u, v, w) = D_0$. This definition follows the terminology of [75]. Although it makes little difference for the presented algorithm, in the ensuing discussion we assume that the B-spline basis functions all have uniform knot sequences and are all of the same order, $d$. Clearly, $D$ in Equation (7.1) is continuous (and hence differentiable) with continuity that depends on $d$, the order of the basis functions.

7.3.2 Preprocessing

The motivation for a preprocessing stage comes from the fact that usually only a small fraction of the volumetric image contributes to the silhouette curves of some isosurface and a given viewing direction. Hence, a preprocessing stage that enable us to early-reject irrelevant parts of the dataset would accelerate the silhouette calculation.

Real life datasets often present a strong spatial coherence in the data. Also, when inspecting the gradient field of the data, a directional locality in the vector field is also apparent. This is because most real life datasets present certain regularity characteristics. The presented preprocessing stage will take exploit of the locality in data and gradient orientation by breaking the dataset into small subvolumes, and arranging these subvolumes in a fast, indexed, data structure.
Consider the original trivariate tensor product B-spline function $D$. Let 

$$G = \nabla D = \left( \frac{\partial D}{\partial u}, \frac{\partial D}{\partial v}, \frac{\partial D}{\partial w} \right),$$

be the gradient vector field of $D$. Hereafter, we assume that the gradient never vanishes. That is, $\|G\| > 0$. Let $\min(D)$ and $\max(D)$ be the minimum and maximum values that $D$ can assume. Let $\alpha$ and $\delta$ be two small, positive and real values. Then,

**Definition 8.** A sufficiently small trivariate Bézier function, $D(u, v, w)$ of orders $d_0$ in $u$, $d_1$ in $v$, and $d_2$ in $w$ with $\nabla D$ directions bounded by a bounding cone’s apex angle $\alpha$ and a dynamic range bounded by $\max(D) - \min(D) < \delta$ is a singleton.

A B-spline trivariate function could be subdivided into a set of singletons via knot insertion [24] at all its interior knots in $u$, $v$, and $w$. The two functions of $D$ and $G$ are subdivided into singletons via knot insertions as part of the preprocessing of the volume. The control points of (the singleton of) $G$ are in a vector space that completely bounds the possible gradients that the function $D$ can assume in this domain. Compute a bounding cone of each singleton of $G$ and denote it by $C_i$. For an in depth description of calculating bounding cones see [3].

Therefore, each singleton has the defined triplet of $T_i = < D_i, G_i, C_i >$ where $D_i$ is the $i$’th original data set’s singleton, $G_i$ is the corresponding gradient vector field, and $C_i$ is the bounding cone of $G_i$. These triplets are then arranged in a two-dimensional lookup table, $T$, of size $(x_T \times y_T)$. Let $T_{l,m} \in T$ be the $[l, m]$’th bin of table $T$. The $l$’th, first dimension of $T$, corresponds to the dynamic range of $D$, or the levels of the iso-surfaces. The dynamic range of the data is divided into a predefined $x_T$ domains. $\min(D_i)$ and $\max(D_i)$ could be easily established by examining the control points of $D_i$. Then, triplet $T_i$ is a candidate to be inserted into bin $T_{l,m}$ if domain $[\min(D_i), \max(D_i)]$ overlaps with the dynamic range of the $l$’th row of $T$.

The second, $m$’th, dimension of table $T$ considers the directions of the gradients, $G$. Consider a set of $y_T$ cones that forms a coverage for the unit sphere, $S^2$. See
Figure 7.1 for an example and [32] for more on such a coverage of $S^2$. The union of all the $y_T$ cones spans all possible direction on $S^2$. Then, triplet $T_i$ is a candidate to be inserted into bin $T_{l,m}$ if cone $C_i$ overlaps with the $m$’th cone of the covering cones’ set of $S^2$. Two cones are considered overlapping if their intersection is not empty.

Figure 7.1: Visibility cones’ coverage: The unit sphere with a small subset of the covering cones.

Triplet $T_i$ could be inserted into more than one bin of table $T$. Yet, by marking processed triplets, the singleton with triplet $T_i$ will be handled once at most.

7.3.3 Silhouette Extraction

Let $V$ be a prescribed viewing direction and $D_0$ the current iso-level. Then, let $S$ be the desired silhouette set of dataset $D$ from $V$. Further, let $G_p$ be the gradient at point $P \in D$. Following Definition 7, a point $P$ is a silhouette point, $P \in S$, if the following two constraints hold

\[
<V, G_p> = 0, \quad D(P) - D_0 = 0.
\]  

Equation (7.2) simultaneously requires that a point be on a silhouette curve and at the proper iso-level.

In order to efficiently resolve Equation (7.2) we exploit table $T$. Given a viewing direction $V$ and an iso-level $D_0$, only validated bins of $T$ with a directional cone that
contains $V$ and an iso-level that is in $[\min(D_i), \max(D_i)]$ are further examined. Each singleton inside a validated bin generates at least one silhouette point $(u_0, v_0, w_0)$ at the middle of the singleton’s domain. Apply, a multi-dimensional Newton Raphson (NR) step to $(u_0, v_0, w_0)$, resolving Equations (7.2) to a desired accuracy. Alternatively, several points can be placed in each singleton as initial seeds for the NR step, or the singleton could be further subdivided until a desired accuracy is reached.

Using trivariate B-spline functions to model the volumetric data brings forth two important advantages over the piecewise constant, voxelized, data representation. Since the gradient field of the trivariate B-spline function is continuous, when using quadratic or higher order basis functions, the resulting silhouette are smooth and continuous, see Section 7.4. Second, the extraction algorithm offers its user with an adaptive behavior. The adaptivity is achieved by setting two operational parameters to the geometric constraint solver. A subdivision parameter which controls the extent to which each singleton is further subdivided which also implicitly controls the number of seed values for the NR iteration. The second parameter controls the accuracy of the NR iteration itself. This parameter directly controls the number of iterations before a convergence. Moreover, the ability of the solver to subdivide the data at any needed parameter and not only at grid points, as in traditional piece-wise constant volumetric data representation, improves its adaptivity. For a proper discussion on the usage of a subdivision scheme and the multi-dimensional NR approach for solving geometric problems, see [35].

7.3.4 Rendering the Silhouette Curves

The silhouette extraction phase generates a dense set of (seed) silhouette points, $S$. Let $G_p^v$ be the projection of $G_p$ onto the image plane that is orthogonal to the unit viewing direction vector, $V$,

$$G_p^v = G_p - V < G_p, V > .$$
Similarly, let $P^v$ be the projection of $P$ onto the image plane.

For each silhouette point, a small edge through $P^v$ is drawn, in the direction orthogonal to $G_p$. In other words, we approximate the silhouette curve in the neighborhood of a singleton by a small edge in the silhouette direction through $P^v$. The length of the edge is in the order of the size of the singleton.

Clearly, if the singleton is further subdivided, the edges drawn for each silhouette point become smaller, offering an adaptive scheme to control the quality of the stroked silhouettes.

The above rendering scheme uses information about the position of the silhouette, while disregarding the magnitude of the gradient in each silhouette point. The magnitude of the gradient can serve as a cue to the characteristics of the material through which the silhouette is passing. Three options are considered for visualizing the magnitude of the gradient on the silhouette. The first option is to color-code the rendered silhouette edges according to the hue values in an HSV color system. Silhouette points with high gradient magnitude are rendered in shades of blue while silhouette points with a lower gradient magnitude are rendered in shades of red (see Figure 7.10). A second experimented scheme draws a small glyph whose shape encodes the magnitude of the gradient in each silhouette point. Different glyph shapes are stored in a lookup table and selected according to the magnitude values. A third option renders, for each silhouette point, an edge with a variable width and a variable gray shade. Silhouette points that have large gradient magnitude are drawn as fine darkly shaded edges while points with low gradient magnitude are rendered as thick lightly shaded edges (see Figure 7.12). In Section 7.4, we present sample images which show the result of using different silhouette rendering schemes with the extraction algorithm.
7.4 Results

We now give some qualitative as well as quantitative results for the proposed algorithm. The average running time is calculated by running the algorithm in multiple viewing directions and averaging the resulting times. The presented implementation is based on a multivariate geometric constraint solver that was used for the calculation of silhouette points, solving Equations (7.2), and is part of the IRIT [54] modeling system. The visualization tasks were conducted with the aid of the VTK [84], Visualization Tool Kit. The GUI system is based on FLTK [42], Fast Light Tool Kit. All the images presented in this work were produced on a 800MHz Pentium 3 PC with 512 MB of RAM.

Tables 7.1 and 7.2 provide some timing results of data-sets that vary in size and complexity. These results are for tri-quadratic and tri-cubic volumes. The adaptive nature of the algorithm makes it hard to compare it to fixed resolution extraction methods. When compared to state of the art silhouette extraction results the presented algorithm achieves silhouettes that are smoother and continuous. When looking at some published timing results, for example in [28, 63], and factoring out the differences in computing environment and datasets, we believe that the added complexity of using continuous data representation costs about an order of magnitude in running time. However, the timing results for the algorithm are taken at the best available extraction quality. Selecting lower levels of quality may lead to comparable running times at the expense of degrading the overall quality.

Figure 7.2 shows a human jaw bone. The camera is situated above the jaw at approximately the center of the skull and is looking in the direction of the front teeth. Figure 7.3 shows the same image with the silhouette overlaid on top of the iso-surface that is extracted from the volumetric data using a marching cubes algorithm. Figure 7.4 shows the neck bones. The camera in this example is looking at the spine from the side. Here, multiple silhouettes at gradually changing grey levels are drawn.
Table 7.1: Timing of silhouette calculation for various data sets. The size of the data is the number of control points in the trivariate function.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>B-spline orders</th>
<th>Voxels ×1000</th>
<th>Silhouette points</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iron protein</td>
<td>3</td>
<td>315</td>
<td>2217</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>315</td>
<td>2367</td>
<td>11</td>
</tr>
<tr>
<td>neck bones</td>
<td>3</td>
<td>63</td>
<td>20878</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>63</td>
<td>29454</td>
<td>133</td>
</tr>
<tr>
<td>jaw bone</td>
<td>3</td>
<td>57</td>
<td>8709</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>57</td>
<td>8649</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 7.2: Memory consumption and preprocessing time for various datasets. Cubic B-spline basis functions are used for these measurements.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Memory(MB)</th>
<th>Preprocess(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iron protein</td>
<td>315</td>
<td>1450</td>
</tr>
<tr>
<td>neck bones</td>
<td>130</td>
<td>1100</td>
</tr>
<tr>
<td>jaw bone</td>
<td>115</td>
<td>796</td>
</tr>
</tbody>
</table>

at close iso-levels, generating a shading effect that conveys depth. Figure 7.5 shows the foot bones. The silhouettes of the skin and bone iso-levels are extracted. The result is overlaid on top of the iso-surface of the bones that was extracted using a marching cubes algorithm.

Figure 7.6 shows an electron density map of an iron protein molecule. Here, five iso-levels are rendered to create depth perception. The large gradient locality in the dataset results in fewer solutions in the solution set and a faster rendering time. Figure 7.7 shows the same iron protein rendered using a ray casting volumetric

134
rendering algorithm with a silhouette overlaid on the image.

Figure 7.8 and 7.9 introduce a distance based coloring scheme. The scheme is usable by ChromaDepth™ stereo-glasses. Silhouette that are close to the viewer are shaded in red while silhouette that are farther away are shaded in dark blue to support the stereo-scopic impression. The background color in this coloring scheme needs to be black or dark blue. The use of stereo-glasses enables such images to convey the three-dimensional shape of the silhouette. For more information about ChromaDepth™ technology see [19]. Figure 7.8 shows a silicium crystal, while Figure 7.9 shows the iso-level of the skin and bone of the foot.

Figure 7.10 shows an example of color-coding the magnitude of the gradient at each silhouette point. The red shades correspond to a lower magnitude of gradient while green and blue shades correspond to a higher magnitude of gradient. The result is overlaid on top of an iso-surface that was extracted using a marching cubes algorithm. Figure 7.11 visualizes the silhouette by attaching a small fan-like glyph to each silhouette point. The fans vary in their opening angle, fans with smaller opening angle correspond to higher gradient magnitudes and vice-versa. The coloring scheme here is the same as in Figure 7.10.

Figure 7.12 visualizes the varying magnitude of the gradient at each silhouette point by changing the color and width of each rendered vector. Areas with large gradient are rendered as thin dark lines while areas of small gradient values are shaded in a paler and thicker vectors. Figure 7.12 shows the silhouette of three different iso-levels of the electron density map of an iron protein molecule. Each iso-level is colored with different hue value.

7.5 Conclusions and Future Work

In this paper, we have presented an algorithm for the extraction of silhouette curves from volumetric datasets. The presented method can be used by illustration packages,
where it can generate high quality smooth silhouettes that are of great value in the generation of a technical illustration. The presented method can also be incorporated into volume rendering packages where it can improve the visual quality of direct rendered images. Overlaying the silhouettes on top of a volumetric image can help to better convey the geometric structure of inner objects that are exposed during a direct rendering process. The algorithm also contains a convenient quality control
Figure 7.4: Neck bones’ silhouettes.

Figure 7.5: Shaded foot bones with silhouettes of the bones and skin overlaid.

feature that enable users to trade silhouette quality for system interactivity. This quality control enables the algorithm to be incorporated into interactive rendering systems, generating coarse silhouette at interactive rates and improving the quality of the silhouette if the user needs better quality images.
Figure 7.6: Silhouette curves of an iron protein molecule.

In order to benefit from the algorithm in interactive environments, several future research paths are presented. First, our implementation uses a general geometric constraint solver that is part of the IRIT [54] modeling system. This solver is not optimized for speed or for this specific application. We expect that an optimized, specific solver might significantly reduce the computation time. Second, in order to retrieve candidate triplets from table $T$. A hierarchy of enclosing cones could be constructed to further reduce the seek-time for candidate triples that might contribute to the silhouette set, given the viewing direction $V$. Third optimization one can employ, which might be proved highly beneficial, is parallel processing. The current algorithm can be parallelized by simply dividing the processed data between multiple machines since there is no data inter-dependency. Forth idea, that was not explored in this paper, is to take advantage of the results of one computation phase to improve the running time of subsequent phases. This idea could be extended in two directions. The first, is to add a multi-resolution ability to the algorithm. This could be done by
Figure 7.7: Direct ray casting rendering of the iron protein molecule, with silhouettes overlaid in blue.

Figure 7.8: Electron density map of a silicium crystal. The image is rendered with a color coding that is suited for the ChromaDepth™ stereo-glasses.

storing the subdivision results of a low resolution silhouette extraction and use these results as the initial step when later extracting high resolution silhouettes. Since higher resolution silhouettes demands the same subdivision steps, this option could
be integrated easily into the current implementation. Second, the results of silhouette computation for a viewing direction $V$ could be used to improve the running time of silhouette extraction for a viewing direction $V + \epsilon$ for small $\epsilon$. Such small changes in the viewing angle are common to interactive applications. The silhouette points of one step could serve as an initial guess for the next step, while new silhouette loops should be sought as well, when the topology of the silhouette edges changes.

The result of the silhouette extraction algorithm are currently rendered as a set of small edges that approximate the silhouette. However, NPR offers a rich set of stroke styles that could be rendered instead the simply rendering edges in order to improve the visual quality of the generated silhouette. While the presented algorithm tries only to deal with silhouette extraction, the same algorithm can be easily adapted to extract other features of interest from the dataset, such as isoclines, line of curvature and lines of curvature and render these as well.
Figure 7.10: Multicolored silhouette of the knee joint. Low gradient values are colored in shades of red while high gradient values are shaded in blue.

Figure 7.11: The silhouette of an electron density map is rendered with a variable geometry option. The color coding is similar to that in Figure 7.10.
Figure 7.12: Silhouette of the electron density map of an iron protein molecule. Three different iso-levels are rendered. Silhouette width and color-saturation levels are made functions of the magnitude of the gradient.
This thesis has focused on modeling using trivariate tensor product Bézier and B-spline functions. In the context of volume modeling, trivariate functions can fill two different roles. First, trivariate functions can be used to manipulate surface-based models. Second, they can be used to directly represent and manipulate volumetric data sets. Most of this thesis focused on the use of trivariate functions in the first framework. Our central observation is that although trivariate B-spline functions are highly flexible, both as a modeling tool and as a volumetric representation, they are also under-employed in CAGD. This can be explained by the fact that they are quite complex to visualize and manipulate. To that end, we decided to explore trivariate B-spline functions, discover novel applications for them and extend existing ones.

Currently, trivariate tensor product B-spline functions are most often used for defining global deformation functions in FFD. When analyzing trivariate B-spline functions, three main obstacles come into mind. First, the modeling of a trivariate B-spline functions necessitates the manipulation of a three-dimensional lattice of control points. Such a manipulation is typically counter-intuitive and cumbersome. Second, trivariate B-spline functions are “too” flexible. By this, we mean that, for example, when these functions are used for FFD, they do not have intrinsic means for
preserving model properties, such as internal volume, total surface area of embedded
gometry and non-self-intersection. To preserve such invariants, as well as others,
the deformation is usually coupled with an optimization process. The optimization
process proceeds the evaluation step and prevents non-admissible settings of control
points from occurring. Yet, a third limitation of trivariate B-spline functions follows
from the relatively high cost of their evaluation. Assume that a trivariate tensor
product B-spline function, \( F \), is of tri-degree \( d \), in each axis. Evaluating \( F \) would
take \( O(d) \) longer than evaluating a surface of a similar bi-degree. This performance
limitation restricts the applicability of FFD, inside interactive settings, to relatively
small models. In this thesis, we addressed only the first and third limitations of
trivariate tensor product B-spline functions.

To address the first limitation, automatic and semi-automatic design tools should
be used to eliminate the need for direct manipulation of control points. One such tool
was presented in Chapter 2, where FFD was used for the placement of deformable
objects. This is a difficult modeling task that usually requires physically-based, hence
expensive, modeling approaches. Chapter 2 presented a low-cost, geometric solution
to this problem, since the application of FFD on the vertices of a polygonal mesh
would introduce deformation errors into the deformed model. In Chapter 3, we de-
rived several bounds on this error. These bounds are intended to control a refinement
algorithm that would limit the deformation error that is introduced during the defor-
mation process.

In order to accelerate the evaluation of trivariate tensor-product B-spline functions
alleviating the third limitation noted above, Chapter 4 suggested the use of modern
GPUs. Using our evaluation scheme and modern GPUs, we were able to deform
large models of more than 120,000 vertices, in real-time. The proposed approach is
usable in any application that requires fast evaluation of trivariate functions, such as
deformation-based registration [78, 102]. In Chapter 5, we used GPUs for an additional volumetric modeling task: computing (H)DDM [33] in real-time. Both these research projects were excursions into a relatively new field of research, which uses the GPU to accelerate geometric modeling tasks. We foresee that other computationally complex modeling tasks could benefit from employing a similar approach.

FFD, as most other deformation techniques, does not alter the topology of the deformed model. In Chapter 6, we extended the FFD paradigm, defining Discontinuous FFD (DFFD), so that it could handle topology changes to the deformed models. Using DFFD, we were able to cut and wrap deformed discontinuous models around obstacles inside a virtual scene. The theoretical work on DFFD was followed by an important application presented in the context of surgical simulations. In this context, DFFD was used as a generic mesh-cutting tool for real-time incision handling during surgical simulations [87]. This incision approach is promising due to its low computational complexity cost, which makes it suitable for real-time applications. Moreover, since the cutting tool that was used is defined in terms of trivariate functions, this approach could be easily extended to handle the cutting of volumetric meshes as well.

This thesis focused mostly on using trivariate B-spline functions as a modeling tool for surface-based models. Nevertheless, trivariate functions can also be used for representing volumetric datasets. In Chapter 7, scalar B-spline trivariate functions are used in conjunction with a geometric constraint solver to extract high-quality silhouettes from the volumetric data.

Trivariate B-spline functions are very effective in representing smooth phenomena but they are less suitable for representing datasets that contain arbitrarily-oriented boundaries and sharp edges. Such boundaries occur frequently in real-life volumetric datasets. For example, this is the case in a CT scan near the transition between a bone and a muscle. Although, tools such as DFFD or trimmed-volumes can be
used, arbitrarily-shaped discontinuities are not easily captured by using B-spline basis functions. During the course of this research, we explored the suitability of trivariate simplex splines, also known as DMS splines [29], as an alternative volumetric representation. One advantage of DMS-splines is the fact that their parametric domain is defined over a triangular, or tetrahedral, mesh, and hence it provides a more flexible representation than the tensor product of trivariate B-spline functions. Using DMS-splines, the specification of discontinuities of more flexible shapes is possible.

We implemented the supporting software that enabled us to explore the usefulness of this data representation. In Figure 8.1, a DMS surface, which underwent a series of modeling operations, is shown. A checkerboard texture is mapped on the surfaces to visualize the parametrization of the surface. In our current implementation we are able to interactively manipulate polygonal surfaces of several thousand

Figure 8.1: DMS-spline surface. The triangular mesh on the bottom part of the image, in blue, illustrates the parametric domain of the surface. The black oval on the right depicts four control points, shown as small red tetrahedra.
Figure 8.2: FFD using DMS-spline volume and a trivariate cubic simplex spline basis functions. In (a), a DMS spline volume is used to deform a model of a goblet. The control points of the DMS spline volume are visualized using small red tetrahedra. In (b), a trivariate cubic simplex spline is shown. Several iso-surfaces are rendered to visualize the function.

Additionally, we implemented an FFD-style deformation algorithm using volumetric DMS-splines. Using this deformation scheme, we were able to achieve interactive frame rates for models of around two thousand vertices. In Figure 8.2 (a), the volumetric DMS-spline is used for to deform a wine glass. In Figure 8.2 (b), a single cubic volumetric simplex spline, the building block of a DMS-spline volume, is shown. To illustrate the shape of this basis function, several iso-surfaces are rendered semi-transparently. Unfortunately, during this implementation effort, we found out that the current evaluation algorithms of DMS-splines are not robust enough for real-world modeling. Furthermore, due to the fact that the only existing evaluation schemes are based on the recursive blossoms paradigm, the evaluation scheme is also
not fast enough to handle sufficiently-large models. Nevertheless, this alternative representation does possess some promising properties and their usefulness for volumetric modeling should be further explored.

The quest for volumetric representations that are better than voxels is far from over. Trivariate tensor product B-spline functions can serve to partially fill this gap, but they are more suitable for representing smooth phenomena that do not possess arbitrarily-oriented boundaries and sharp edges. A future direction, of potentially high merit in this line of research, is to seek an alternative volumetric representation that would provide the required flexibility. Such an ideal trivariate representation would have sufficient flexibility to efficiently capture smooth material transitions and, at the same time, be able to represent sharp boundaries. DFFD can be used for this task but it can only represent efficiently discontinuities along iso-surfaces. Trimmed volumes may also suffice, but they are complex to model and manipulate. An alternative research direction might possibly seek a solution for the problem of representing volumetric data that contain arbitrarily-oriented edges, by considering an extension of non-tensor product wavelets into a 3D space. Some recent approaches such as ridgelets [12] and curvelets [13] were suggested in the context of image processing, and can be used as a starting point for this research. An extension of these ideas to 3D was also recently suggested in [104]. In the future, we would like to explore the applicability of such tools for the problem of a volumetric modeling. We foresee that these tools and their extensions would prove highly valuable.
Bibliography


פונקציות שלושה משחבים
במיוחד גיאומטריה, דינומט רפואית
וגריפת מומחה

דיבור על מחבר

לשם מלייב אלקק של דרימשט ליקקלת ח Toastr

dוקר הילgetService

שלום שירן

הוסע להט אתטנitmap - מכון טכנולוגי לישראל

אידר תשמ"ז וידף, מאי, 2006
המחק נועד להשתתף בדנהの一 פרופסור גרשון אלבר בסקולות לעוד ימח口コミ

אני מודע ל العاصונות על התמיכות והכפרות והריכוז הבינלאומי
תוכן ענין

1. מבוא
1.1. ז règle מודלים נmetic
1.2. מיקום של מדדים קשיים בתורת השפה והגיון והוקסם
1.3. הסט של שגיאות הנגיה של מודליים פונקציונליים
1.4. דרישות תקשורת והקשקשים - מתמטיקה והпередה שלה
1.5. רדך מודלים על ידי הח moduleIdה הכלית בגרירת התוכנה
1.6. פונקציונליות ר💖וח או ר mascot
1.7. חלום quánית ממידור פנימי
2. הפרטים של מדדים קשים בתורת השפה והוקסם והקשקשים
2.1. מבוא
2.2. תכונות קומרט
2.3. אנגימה והקשקשים
2.3.1. פונקציונליות תחומי
2.3.2. מודל המ maxHeight והקשקשים
2.4. מספר החודש
2.4.1. חדש נוכי של פונקציונליות תחומי
2.4.2. אנגימה חלשים
2.4.3. אנגימה
2.5. דוגמא עבורה תעוכת
2.6. מסקנות ועמודatat תעוכת
3. הפרטים לשיאת מכפרי האזפורטה שיניהוז ודך פונקציונל
3.1. מבוא
3.2. הפרטים לשיאת העניין עבורה גיאומטריה שיתוד
3.2.1. הפרטים לשיאת העניין עבורה גיאומטריה של
3.2.2. הפרטים לשיאת העניין עבורה גיאומטריה של
3.2.3. הפרטים לשיאת העניין עבורה גיאומטריה של
3.3. סיכום
59. שיטות ליניאריות ומשושים בבלוק - אופן ביצוע תופעת וקופסה

61. קורמכת וחברת ק_accumת
62. שיטה מקרמכת בבלוק
63. למידה מקרמכת בבלוק
64. למידה מקרמכת בבלוק
65. קורמכת וחברת ק
66. קורמכת וחברת ק
67. תרגול וידיעות
68. מסקנות וידיעות
69. תרגול וידיעות
70. תרגול וידיעות
71. תרגול וידיעות
72. תרגול וידיעות
73. תרגול וידיעות
74. תרגול וידיעות
75. תרגול וידיעות
76. תרגול וידיעות
77. תרגול וידיעות
78. תרגול וידיעות
79. תרגול וידיעות
80. תרגול וידיעות
81. תרגול וידיעות
82. תרגול וידיעות
83. תרגול וידיעות
84. תרגול וידיעות
85. תרגול וידיעות
86. תרגול וידיעות
87. תרגול וידיעות
88. תרגול וידיעות
89. תרגול וידיעות
90. תרגול וידיעות
91. תרגול וידיעות
92. תרגול וידיעות
93. תרגול וידיעות
94. תרגול וידיעות
95. תרגול וידיעות
96. תרגול וידיעות
97. תרגול וידיעות
98. תרגול וידיעות
99. תרגול וידיעות
100. תרגול וידיעות
101. תרגול וידיעות
102. תרגול וידיעות
103. תרגול וידיעות
104. תרגול וידיעות
105. תרגול וידיעות
106. תרגול וידיעות
107. תרגול וידיעות
108. תרגול וידיעות
109. תרגול וידיעות
110. תרגול וידיעות
111. תרגול וידיעות
112. תרגול וידיעות
113. תרגול וידיעות
114. תרגול וידיעות
115. תרגול וידיעות
116. תרגול וידיעות
117. תרגול וידיעות
118. תרגול וידיעות
119. תרגול וידיעות
120. תרגול וידיעות
121. תרגול וידיעות
122. תרגול וידיעות
123. תרגול וידיעות
124. תרגול וידיעות
125  שבעשות קדמויות  7.2
127  והולך פטעמי שלולית  7.3
128  זיהוי רמיה  7.3.1
128  דגמה פקודה  7.3.2
130  שלוח שלולית  7.3.3
131  תפקוד תמונת שלולית  7.3.4
133  תאמות  7.4
135  מסקנות ועבדה תענידית  7.5
143  מסקנות ועבדה תענידית 8
רשימת אורחים

פונקציות B-spline וקטורית בשילוש מ(xy)

1.1

שנしたもの של אינפיניטיסים

1.2

שימו את הפוכותファンקרת של היגו של בום אמט

1.3

שימי בファンקרת DDM של המנה

1.4

שימי ב- FF דファンקרת למנה

1.5

יפור תמציתון של הל數據 של היגו

1.6

Escher של Reptiles

2.1

FFD של הבクラшка

2.2

בנוצר של קורמות היגו של מונים של השבך

2.3

Extrusion

2.4

שהואות של ביניית אוטומוטיית פונקציות

2.5

FFD של הקורמות של הבクラקשה של אופטיזציה

2.6

FFD של Fermi

2.7

טבלת תיוק עד הפרק יצור

2.8

FFD של ניואק

2.9

 ölolkת מודל קורלית של הפונקציה שלחט פית

2.10

אינפיניטיסים של שילול מצלת קורפה

2.11

מקורות של קורפה על גג של כוס

2.12

SSID מוניות

2.13

מש וריאליhör dilemma

2.14

לווע פונקציה של CGGC

2.15

כן פונקציה של CGGC

2.16

אינפיניטיסים של דיפיור

2.17

סמכות של הלוק הובער של שיר ווית

3.1

.ForeColor yohad בחר משולש לשור ווית

3.2

סמכות של ת_Thread משולש שו

3.3

סמכות של ת_Thread משולש שו

3.4

FFD של נופריים

4.1

FFD של נופריים

4.2

FFD של נופריים

4.3
65 הסכנת וי-שלביות עיבוד HFFD 4.4
69 אכיפת היחלים בומת אתFrozen 4.5
70 אכיפת היחלים בומת אתFrozen 4.6
71 אכיפת היחלים בומת אתFrozen 4.7
78 DDM חידות הבסיס נאפות מריצוע פפת ופנורמייםumperLIN 5.1
79 HDDM חידות הבסיס נאפות מריצוע(perLIN 5.2
81 מפת כיוון וצורות עיבוד HFFD 5.3
82 אולטרהפורץ של מיקום וחוזת בוסס בדידה HFFD 5.4
86 מפת כיוון וצורות עיבוד מצל משלי Feld 5.5
90 מפקח קורייט על מצל משלי Feld 5.6
91 מפקח מצל משלי בשדות בחיתות בוסס אופטיקה Feld 5.7
91 סנסוור דינמי משלי Feld 5.8
92 HDDM משלי Feld 5.9
92 HDDM משלי Feld 5.10
93 ריצוף סנסו בחיתות בוסס משולש Feld 5.11
94 Displacement mapping HDDM במשולש Feld 5.12
95 במקiosis Murant המצל משלי Feld 5.13
109 זזנית על מצל משלי Feld 6.1
111 זזנית על מצל משלי Feld 6.2
112 פונקציה ממקימה Feld 6.3
113 מודל של בדיד עקופת זזנית Feld 6.4
114 זזנית על מקימה Feld 6.5
115 פונקציה ממקימה Feld 6.6
117 מודל של אולטרהפורץ על ידי בדיד עקופת Feld 6.7
118 מקימה משמשת לזריקה של מצל משלי Feld 6.8
119 אינפנצה של זִוֶר שִׁבְרֶה עִּבְרָה צֶּרֶבֶב Feld 6.9
119 אינפנצה של זִוֶר שִׁבְרֶה עִּבְרָה צֶּרֶבֶב Feld 6.10
120 מודל של זִוֶר שִׁבְרֶה עִּבְרָה צֶּרֶבֶב Feld 6.11
130 בוסי דינומי לֶאִיید Feld 7.1
136 צֶּרֶבֶב פִּנְדֵּם הַדָּלָת Feld 7.2
136 צֶּרֶבֶב פִּנְדֵּם הַדָּלָת Feld 7.3
137 צֶּרֶבֶב פִּנְדֵּם הַדָּלָת Feld 7.4
רשימה סכמטית

88 תומך אלגוריתמים HDDM
132 צופן ורשיב של אלגוריתמים תורמים לכלית
132 פרויקט וקורי והכנת מספר אלגוריתמים תורמים לכלית
っくり

במקרה זה, נערך מחקר על תכונות התוכנה הממויות של פיתוח הולוגרים וกกון, שהовать להם בהבנה והערכה של התוכנה הממויות בימינו. הד HTMLElement יועד לquisar תכונות של התוכנה הממויות בימינו, ולה𬃊 בהבנה והערכה של התוכנה הממויות בימינו.

במקרה זה, נערך מחקר על תכונות התוכנה הממויות של פיתוח הולוגרים וגדל בplementation של פיתוח התוכנה הממויות בימינו. הד HTMLElement יועד לquisar תכונות של התוכנה הממויות בימינו, ולהฟัง בהבנה והערכה של התוכנה הממויות בימינו.

במקרה זה, נערך מחקר על תכונות התוכנה הממויות של פיתוח הולוגרים וגדל בplementation של פיתוח התוכנה הממויות בימינו. הד HTMLElement יועד לן תכונות של התוכנה הממויות בימינו, ולהψ בהבנה והערכה של התוכנה הממויות בימינו.

במקרה זה, נערך מחקר על תכונות התוכנה הממויות של פיתוח הולוגרים וגדל בplementation של פיתוח התוכנה הממויות בימינו. הד HTMLElement יועד לן תכונות של התוכנה הממויות בימינו, ולהψ בהבנה והערכה של התוכנה הממויות בימינו.

במקרה זה, נערך מחקר על תכונות התוכנה הממויות של פיתוח הולוגרים וגדל בplementation של פיתוח התוכנה הממויות בימינו. הד HTMLElement יועד לן תכונות של התוכנה הממויות בימינו, ולהψ בהבנה והערכה של התוכנה הממויות בימינו.
הימית מספר רבך הקליפה לינוע מורדלים וקמפיין. חורף ועופת חצלאל, היא בועת ווקסלים
ובאת הצהובות לתלות-מעון של פיקסלים, מ الساد 맞יצים פתיחת
העבודה ביצועים נוספים וממקמים, סרטים, להיפוך תנועת ומקחי המיתון הנ更低.
לצאתו, והשימש ביצועים ליצירות יזון ממדל. את מדריך הנבואה מ יצירת עשר פיסולות להבננות וליצבות
שלי מורדלים וקמפיין. המפגשה והתיקון של יזון ביצועים וקסיJimmy ליצבות שיפורים
בהئت מתוחכם בברית אבסטרקציה מבית, בין מתכון ביצוע על המיתון.
ורשה המיתון להבנה את תונון של כל את ברשת חתו-מדית. ליצבות, סרוקת
בינה ואנCertificates לשונות והפכות כפ_splits שעורב לע המיתון.
ועבר את המיתון לארץ הפניקרים. מכסה נססה נצפת בניקודה של המיתון
הנ��ם אבל בודא-כל גול צוין. בוחנה麥 מכי בכרה להגנה חזותمبرכת
ומיתון, המיתון נטף ברובות שכת של 16 סירות פלטה. הצריך של מסרן שנייה על תואם
ועבר והראים רהב גוים למודלים הספרים לつつים מספרים בר בוכרי הנבואה של הומובט
באשר ורגיות לבעז פסילה לע המיתון הנבואה בכסב עפרות עראmeyeווטית, המיתון
ורבח של המיתון הוכת אעקב הגיבורים ולא מי מזו duroe.
על מפת הלחת בשמאלא פוחת בצמך והשעון יזון והפוכים למודלים מפורטים.
והל מציאוניות מתמשכות בצמך הדריים לחתוך וסבוני ייר של המיתון.
היפוגה הדירייר
מונצל מבשל מAtlas דרום לק Octtree אישメディア ליצירות והפוכים במציאות
מעיונים בטן. סגנונות הלוחית והאריות משמשות במע-fashion ודרייבולדIAN ליצירות
למביתר וההר_BOTH. כלאו מתוחנה לשון Kספאות ביצועה ליברונות
ופמיירום. במקום, התוכנה ראשית פיסולת ביצועי אל יזון משני אל אבח הנה
B-spline
בתחי אור מתסכים פיסולת לכלל ליצירות风光. ליצורים זה, הארגון על הדיסני בר שנטה
ששבצת בטリアル המידיאלי והגונתרים, המדוריכים את הזרות של כלים ייזום וכפ划分
B-spline במודלים פורים. בשטח הоздיאלי היברוני, הועדה מציאותו ומרכזים כמנים
ולא שישה המיתון ליצירות מודלים. במשה שובות ההתחדשה של פסקולות ממערכות
ללא פסילות פקלאדעל פורטסיון לזר. בהזא מפתוחה בכלים אחר מכוסי
B-spline

\[ F(u, v, w) = \sum_{i,j,k} P_{ijk} B_i(u) B_j(v) B_k(w) \]

where \( B_i(u) \) are the B-spline basis functions. The coefficients \( P_{ijk} \) are

\[ P_{ijk} \]

The B-spline basis functions are defined for uniform and non-uniform knot vectors. The B-spline basis functions are non-negative and have compact support. The degree of the B-spline basis functions is determined by the number of control points. For a quadratic B-spline, the degree is 2. The B-spline basis functions are used to represent the surface in a piecewise polynomial form.

In this thesis, we focus on the B-spline basis functions and their properties. The B-spline basis functions are used to represent the surface in a piecewise polynomial form. The B-spline basis functions are non-negative and have compact support. The degree of the B-spline basis functions is determined by the number of control points. For a quadratic B-spline, the degree is 2. The B-spline basis functions are used to represent the surface in a piecewise polynomial form.

The B-spline basis functions are defined for both uniform and non-uniform knot vectors. The B-spline basis functions are non-negative and have compact support. The degree of the B-spline basis functions is determined by the number of control points. For a quadratic B-spline, the degree is 2. The B-spline basis functions are used to represent the surface in a piecewise polynomial form.

In this thesis, we focus on the B-spline basis functions and their properties. The B-spline basis functions are used to represent the surface in a piecewise polynomial form. The B-spline basis functions are non-negative and have compact support. The degree of the B-spline basis functions is determined by the number of control points. For a quadratic B-spline, the degree is 2. The B-spline basis functions are used to represent the surface in a piecewise polynomial form.
The wireframes of objects cloak the legacy by modifying their colors and opacity. However, they are still too complex and require a lot of processing power. A preferable method is to use B-spline (Non-photorealistic rendering) to generate wireframes. This method is more efficient and can be used for a wide range of applications. The user can customize the appearance of the wireframes by adjusting the parameters and applying different styles. This technique allows for the creation of visually appealing wireframes that can be used in various fields such as architecture, engineering, and design.
מששhexdigestים. שימו לב לפקודות הנמויותhä ב.getValueAtויול ובהן ילךזרת מודל נופי סימול.

בפרט בפקודות B-spline המשולשת מרחבים י ArrayBuffer המושפע על צורת השלול הלטיה של המודל. עבור B-spline המשולשת מרחבים הייתם י ArrayBuffer התוכנות של הלטיה ביניהן יعار. בפונקציה Ember המודל בנד מודול B-spline הזרם תלול למודל ומדא הזרם תלול למודל.

נמויות ומקראות, גם, אם היו תיאור של מודל נופי שאינו חוק חוכל שמת אל ר nhị

בכלו בחוד בלול.
פונקציות שלושה משתנים
במیدול גיאומטרי, דימות רפואית
גראפיות ממוחשבות

שגריר שניי