FREE-FORM DEFORMATION FOR COMPUTER-AIDED ENGINEERING ANALYSIS AND DESIGN

By

Amy Lynn Ladner

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computational Engineering
in the Bagley College of Engineering

Mississippi State, Mississippi

August 2007

FREE-FORM DEFORMATION FOR COMPUTER-AIDED ENGINEERING ANALYSIS AND DESIGN

By

Amy Lynn Ladner

Approved:

Greg Burgreen
Associate Research Professor
of Computational Engineering
(Thesis Director)

Robert Moorhead
Billie Ball Endowed Professor of
Electrical and Computer Engineering
(Committee Member)

Seth Oppenheimer
Professor of Mathematics and
Statistics
(Committee Member)

Donna Reese
Associate Dean of Engineering and
Professor of Computer Science and
Engineering (Committee Member)

Mark Janus
Associate Professor Aerospace
Engineering and Graduate Coordinator
of Computational Engineering

W. Glenn Steele
Interim Dean of Engineering

Name: Amy Lynn Ladner

Date of Degree: August 11, 2007

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Greg Burgreen

Title of Study: FREE-FORM DEFORMATION FOR COMPUTER-AIDED ENGINEERING ANALYSIS AND DESIGN

Pages in Study: 51

Candidate for Degree of Master of Science

Toward support of the use of geometry in advanced simulation, a free-form deformation (FFD) tool was designed, developed, and tested using object oriented (OO) techniques. The motivation for creating this FFD tool in-house was to provide engineers and researchers a cost efficient, quick, and easy way to computationally manipulate models without having to start from scratch while readily seeing the resulting geometry. The FFD tool was built using the OO scripting language, Python, the OO GUI toolkit, Qt, and the graphics toolkit, OpenGL. The tool produced robust and intuitive results for two-dimensional shapes especially when multiple point manipulation was utilized. The use of "grouping" control points also provided the user the ability to maintain certain desired shape features such as straight lines and corners. This in-house FFD tool could be useful to engineers due to the ability to customize source code.

DEDICATION

I would like to dedicate this research to the following:

My parents, David and Gail Ladner, who inspired me to never give up

My fiancé, Dustin Pittman, for all of his love and support

My sister, Melissa Peterson, for always being there for me

My twin, Tammy Ladner, for being with me through it all

ACKNOWLEDGMENTS

I would like to take this time to express my sincere gratitude to the many people without whose support and selfless assistance this thesis would not have been possible. First, sincere thanks are due to Greg Burgreen, my advisor, for the time and effort he has devoted to helping me succeed even when times were tough. Second, thanks are due to my family and friends who never stopped supporting me even when I doubted myself. To my parents, thank you for encouraging me to continue my education. To my sisters, thank you for allowing me to complain when no one else wanted to listen. To my fiancé, Dustin, for believing in my success even when I did not. And last, but not least, I have to thank God for giving me the strength that I needed to get through the tough times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Numerical simulation is primarily used as an investigative tool to predict the physics associated with a given geometry. Unfortunately, this use of simulation has encouraged a shortsighted view of geometry as simply a means to obtain the discrete mesh necessary to perform analysis. More recently, multidisciplinary analysis and design optimization has required a reconsideration of the role of geometry in advanced simulation. For example, changes to geometry induced by physics-based simulation or by optimal shape modifications need to be accurately reflected in the geometric model as well as the discrete mesh representing that geometry. To date, there are few mechanisms and little automation to achieve such a dynamic union between geometry, mesh, and shape modifications.

Many of the geometries used in bioengineering applications have a complex topology and organic shape. Consequently, discrete geometry models are found more frequently in bioengineering compared to other traditional fields of engineering. Typical bioengineering shapes derive from a diverse range of sources including: MRI and CT medical imaging data sets, scattered point clouds of complex physiologic structures, and numerical simulations of growth mechanics of certain biologic processes such as platelet-mediated deposition and thrombogenesis. Due to the manner in which these data are obtained and digitally processed, smooth analytical shapes are rarely obtained, and discrete geometry models have become the normal mode of representation. When these discrete geometries are part of an advanced simulation that requires dynamic shape modifications, one finds that a crippling technological void exists.

In general, there are two main categories of geometry representation: CAD-based and mesh-based. With respect to dynamic shape modification, the capabilities of CAD-based approaches far exceed those of mesh-based models. Hence, one can find numerous research efforts that try to transform discrete representations into smooth analytical B-spline and NURBS (Non-Uniform Rational B-Splines) representations. While such methods have been successful, they are loaded with inordinate complexities and compromises. For example, automatic conversion of complex polygons or polyhedral meshes with sharp features (i.e., ridges, corners, darts, etc.) to B-spline patches remains a challenging problem. Also, enforcing higher-order continuity at extraordinary vertices is difficult or sometimes impossible and significantly increases the complexity of an analytical representation. In view of these limitations, what becomes evident is the need for a robust geometric framework that directly embraces discrete geometric modeling.

1.1 Objectives

The goal of this thesis is to develop an innovative, unified, object-oriented framework focused on the shape manipulation of discrete geometry models. The centerpiece of the proposed framework consists of free-form shape deformation techniques applied directly to discrete geometric models. This research implements free-form deformation (FFD) techniques to provide an object oriented shape modification kernel to computer-aided engineering analysts and designers. Specifically, Bézier basis functions serve as the foundation of the shape deformation equations.

1.2 Background

FFD was first introduced by Barr in 1984 [1]. He introduced global and local deformation techniques as new hierarchical transformations, such as bending, twisting, stretching, and tapering, for deforming an object. Because the operations are structured hierarchically, it is easy to create a more complex object from several simple ones. This method utilizes the surface normal vector and a transformation to achieve the normal vector of a smooth deformed surface.

Sederburg and Parry later developed a better model in 1986 [18]. Their technique can encapsulate and deform virtually any type of surface geometry. They make use of the trivariate Bernstein polynomials for transforming objects through the use of control points. They compare the FFD volume to a clear, flexible plastic that contains or embeds an object. Manipulation of the plastic in turn results in manipulation of the embedded object's shape.

Another approach to FFD was developed by Chang and Rockwood [2]. This method utilizes affine transformations, a Bézier curve, and a generalized de Casteljau algorithm. An object is deformed by repeatedly applying affine transformations in space and warps along the Bézier curve. Instead of using control points directly, as in [18], this method utilizes "handles" as indirect actors on the control polygon.

Although these are three unique FFD techniques, all accomplish the same thing, namely, shape modification. The FFD shape modification technique presented in this thesis works by positioning a lattice of control points about a discrete model, establishing an analytical relationship between the lattice and the model, and indirectly modifying the shape of discrete model via control point manipulation. This procedure produces an interface to deform discrete meshes in an intuitively consistent fashion.

CHAPTER 2

CURRENT RESEARCH

In the area of FFD, research has historically been focused on animation. Although FFD is still largely being used by animators, engineers and medical researchers apply the same techniques introduced by [18] to manipulate relevant geometries with the aim of achieving better visual results.

2.1 Animation

Cartoon animators use FFD to add personality or movement to a character or object [4]. For example, FFD techniques could be used to make a character jump. The character would be squashed first, to make it appear as if it is squatting, and then stretched, to give the appearance of jumping as far as possible. As another example, an animated car crash could be modeled more realistically using FFD techniques to locally deform the vehicle at the point of impact instead of the entire vehicle. This idea has been applied to an application referred to as "ToonTown." The developer of ToonTown wanted to find a way to locally deform one part of a car instead of just demolishing the entire car [6]. Based on the idea of dropping an anvil on a particular part of the car, he used FFD techniques to animate an approximate type of effect the anvil would have on the car. Figure 2.1[6] shows the result of smashing the car in random places.

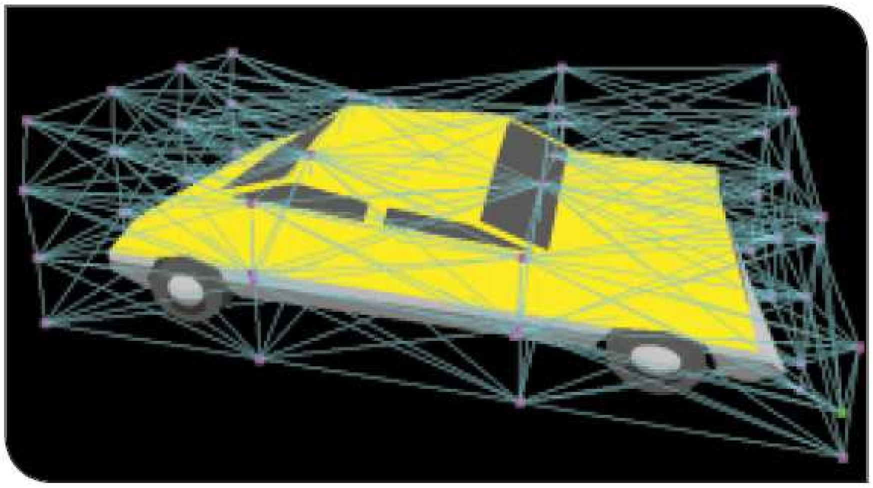

Figure 2.1

A taxicab after anvils have dropped on it [6]

2.2 Engineering

Engineers at the NASA Langley Research Center have applied FFD techniques in shape optimization procedures. The use of a nonuniform rational B-spline (NURBS) representation, which will be presented in Chapter 4, was found to preserve the smoothness of the initial geometry while still satisfying properties of geometric accuracy for aerodynamic shape as prescribed by National Advisory Committee for Aeronautics (NACA) [15]. Aerospace engineers sometimes prefer to use FFD techniques that utilize NURBS in order to reduce the number of design variables during the design optimization process. Reducing the number of design variables can result in a reduction in the time taken to optimize a model shape.

2.3 Medical Researchers

Medical researchers have applied FFD techniques to breast MRI [14]. FFD, based on B-splines, are used to describe local breast motion of volunteer patients. Scans were taken from several volunteers, and each scan required the volunteer to do something different (i.e. cough, move around, fully remove herself from the machine, etc). Using FFD in conjunction with other transformations enabled smoothing of the rigid results produced due to movement. This technique was able to produce improved images of the breast, and as a result, could ultimately have a significant impact on detecting breast cancer in early stages simply due to the improved clarity of the images [14]. Figure 2.2[14] shows a few of the images produced during this study. The image labeled (d) is a result of applying FFD techniques along with other transformations. The tumor is much easier to recognize using this technique compared to the other three techniques studied. Although the description in [14] was not specific in how the FFD techniques were employed, it was definitively stated that the use of FFD resulted in the best images.

2.4 Commercial Software

In addition to these research efforts, several software companies have developed shape manipulation programs to aid in design optimization. These companies rely on the same basic FFD techniques that current researchers are using, while each of them add their own unique features. Two of the more well known modeling programs are *Sculptor*^{TM} (Optimal Solutions Software, LLC, Idaho Falls, ID) and PowerCLAY (Exa Corporation, Burlington, MA).

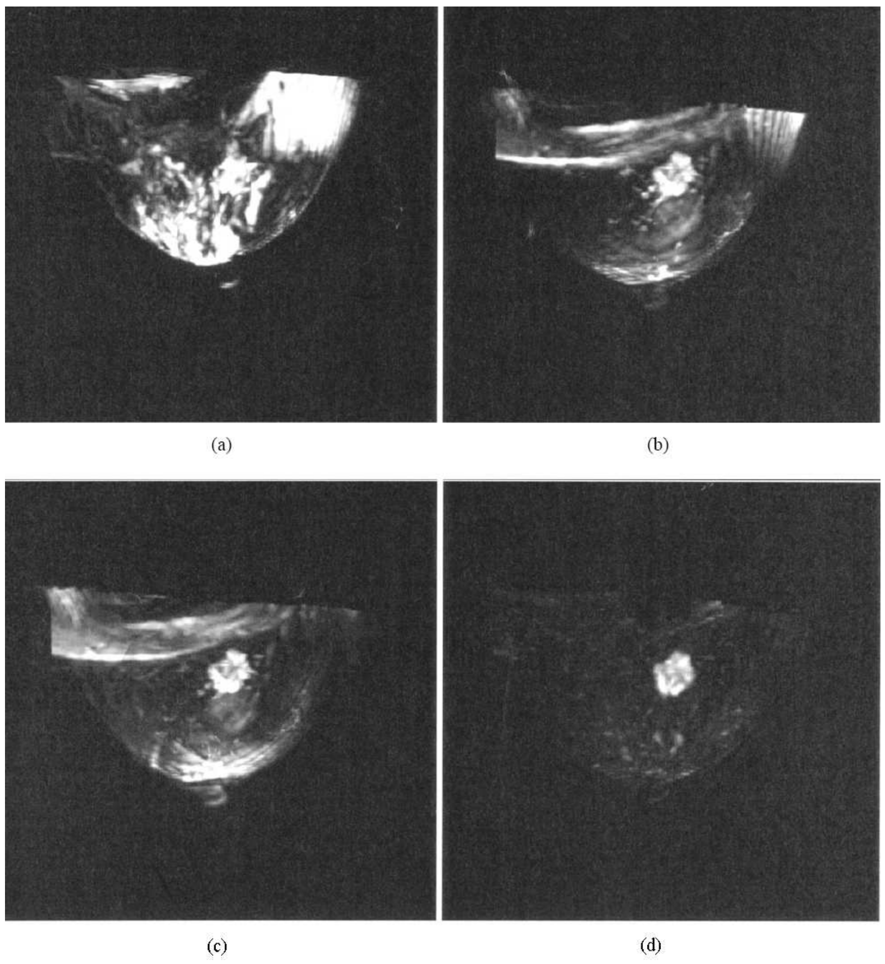

Figure 2.2

Breast MR Images [14]

2.4.1 *Sculptor^{TM}*

Sculptor^{TM} is a package that is useful in the area of computational fluid dynamics (CFD). It started out as an in-house analysis tool but it has since graduated into a powerful modification tool with an easy to use GUI for CFD engineers [16]. It was built on the principle of reducing the number of point adjustments required to produce good results. As a result, it relies on the use of control points to modify a mesh such as an airplane wing.

Not only did *Sculptor*^{TM} make things easier for engineers by reducing the number of points needed to manipulate a mesh, but it can also be coupled with automatic shape optimization techniques that include engineering constraints such as pressure drop [16]. It can modify any object whose shape is defined by points of a grid such is common in the field of numerical simulation or visualization. The new modified shape can then be reanalyzed using the same, or different, simulation code.

Figure 2.3 [16] shows the use of *Sculptor*^{TM} in modifying the shape of a pipe elbow. The points that connect the lines of the deformation volume are the control points and are the only means by which the user can manipulate the pipe elbow [16]. This type of control volume, in the use of shape modification, is central to programs like *Sculptor*^{TM}.

2.4.2 PowerCLAY

PowerClay is also a shape modification tool that gives users control of shape modification through the use of control points. It allows users to produce accurate CAD shapes without the use of a CAD system and is easier to learn than a full-scale CAD tool [9]. PowerCLAY utilizes two different methods that allow users to select the portion of the model

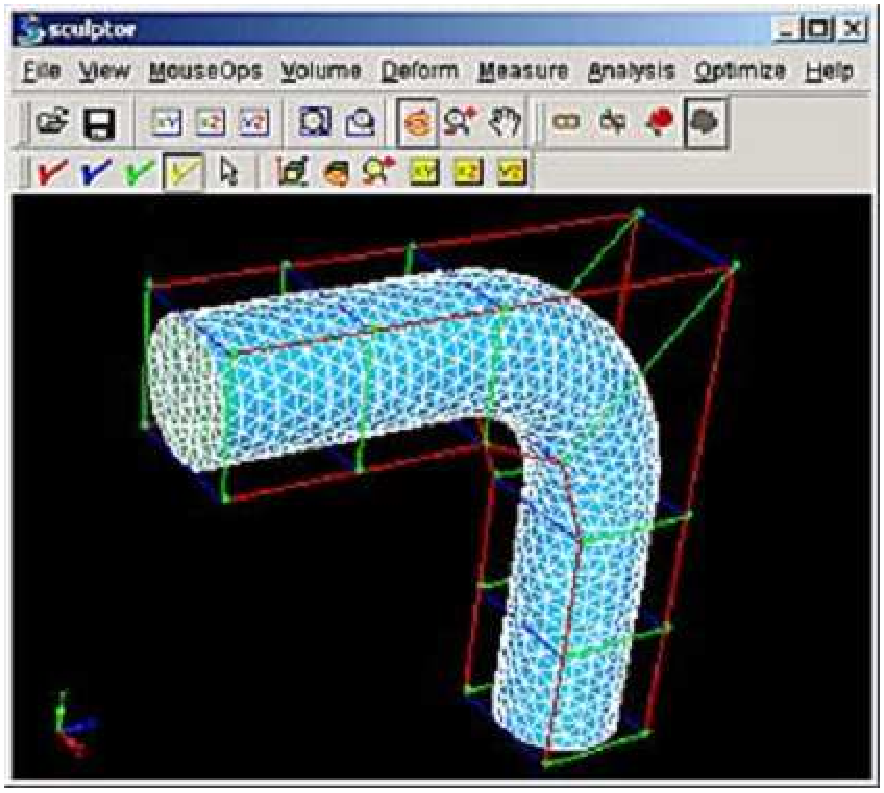
sculptor
File View MouseOps Volume Deform Measure Analysis Optimize Help

Figure 2.3

Shape Modification of a pipe elbow using *Sculptor*^{TM} [16]

to modify. The first method, and the only method that will be discussed here, makes use of a deformation lattice. The region to be modified is selected, and a lattice is drawn around that portion of the model with control points. Finally, a relationship between lattice and geometry is established so that manipulation of the control points in turn manipulates the shape.

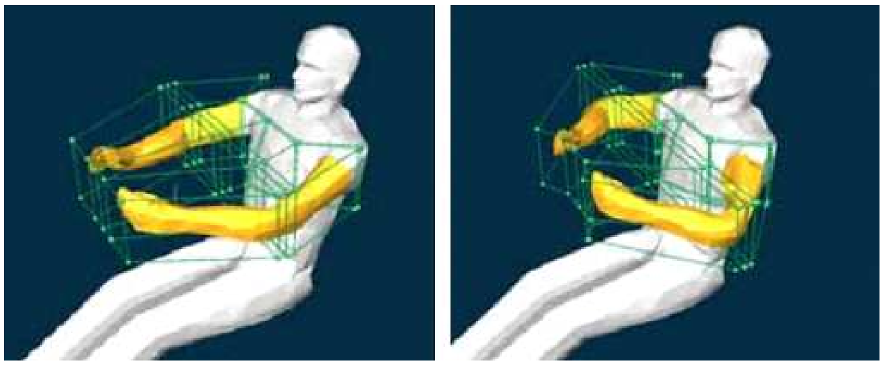

Figure 2.4

Image of PowerCLAY [9]

Figure 2.4 [9] shows the result of modifying the arms of a human model. Without PowerCLAY, making the arms appear to move closer to the body would be a very time consuming job as each and every point would have to be modified individually. Through the use of a lattice, modifying the arms on the model becomes a much easier process and produces effective and intuitive results.

These recent advances in research and software development inspired the production of an in-house FFD tool. In the next few chapters, the development environment, mathematics behind FFD, components of FFD, and code development will be discussed. Several examples will be introduced and discussed and finally ideas for future work on the current tool will be mentioned.

CHAPTER 3

OBJECT-ORIENTED DEVELOPMENT ENVIRONMENT

After the decision was made to build the FFD tool from scratch, the development environment had to be chosen. In general, there are two types of development environments: procedural oriented (PO) and object oriented (OO). PO programming is an approach of coding a program such that it is built around functions and statements that manipulate data. OO programing (OOP) is a methodology that allows the programmer to combine data and functionality into an "object" [21], where an object refers to an entity that exhibits specified characteristics and behaviors. It contains data and procedures along with an interface that describes how it can interact with other objects. An object is visually represented in Figure 3.1 [7]. The FFD tool needed certain desirable features such as multi-platform capability and reusability. It also had to be intuitive and simple to use and had to produce expected results. Because of these features, OO design (OOD) and OOP were obvious choices for the development environment. As a result, the overall framework and programming language also need to be OO friendly. Utilizing OO techniques to satisfy the desired features was also one of the goals of this thesis.

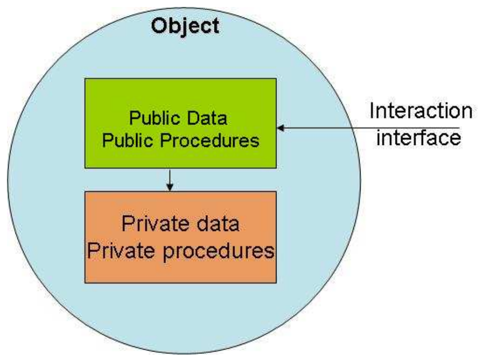

Figure 3.1

An abstract object

3.1 Design

OOD forces programmers to think in terms of objects rather than procedures. Reference [7] defines OOD as "a programming language that has five conceptual tools to aid the programmer". These five conceptual tools [20][7], and their definitions are:

1. Encapsulation: the ability to build self-contained pieces of software,
2. Information hiding: the ability to protect certain data members so they cannot be changed by the user,
3. Inheritance: the ability of an object to "inherit" all the traits of a different object,
4. Interface: the ability of the object to interact with an outside entity,
5. Polymorphism: the ability to define different functions or classes with the same name.

Programs that follow OOD can be more readable than non-OO programs, because it follows the natural way of thinking. Another benefit of OOD is that debugging the code becomes easier. This is a direct result of programming with objects after the project has been designed. Once one object has been debugged, then it often does not need to be considered when debugging the rest of the program.

3.2 Programming

OOP is a design process that allows inheritance and polymorphism but is not restricted to any one programming language. It gives the programmer a better way to organize the program due to the use of classes and modules within these classes. OOP forces the programmer to think of programs in abstract terms [20]. For example, a house can be abstracted as several "house" objects classified into the different architectural style: Beach, Colonial, Country, Log, etc. Each would also have several room objects: family room, bedroom, bathroom, kitchen, and so on, which are characteristic of that particular style.

Thinking of objects in this natural way fits the thesis problem. It is natural to think of the FFD tool in object oriented terms. The interface is one object, the manipulation space is another object, the FFD lattice is yet another object, and so on. These objects must be able to work together to achieve a fully functional FFD tool.

Recalling our desired features from the previous section, OOP is the most logical decision for the development environment. Ultimately, the goal of OOP is to produce natural, reliable, reusable, mantainable, and extendable code in a timely manner [20]. OOP allows the FFD tool to be built while successfully attaining all of the desired features and reducing the time spent during development as long as the OOD is followed exactly. It also allows the program to be both maintainable and extendable due to the inherent nature of coding with objects.

In general, classes and objects are the building blocks of OOP where a class represents a new data type and an object is an instance of the class. Each of these classes have attributes and methods associated with them. Attributes refer to the characteristics of a class, and methods refer to behavior of a class [21]. For example, a “house” object may have an attribute called “color” or “number of rooms,” and a method called “open door.”

3.3 GUI Frameworks

In keeping with OOD and OOP, the graphical user interface (GUI) framework (aka GUI library) needs to be OO friendly. A general GUI framework is simply a set of pre-programmed classes and functions with several built-in objects such as buttons and menus that are essential for an effective user interface. The GUI framework for the FFD tool should make the coding process much easier and more reliable while also being comprehensive but not overwhelming. Although there are many GUI libraries currently available today, but not all satisfy the requirements of the FFD tool. Some are not cross-platform compatible (such as Motif), while others were developed for a specific programming language (such as Tk).

The GUI library chosen for the purpose of this thesis and which satisfies all of the required features is Qt (Trolltech, ASA, Oslo, Norway). According to [11], Qt is a development framework that enables a user to develop high-performance, cross-platform applications through the use of specific tools and features. Qt has several built-in classes and objects, most of which inherit a base class. Even a cursory survey of classes in the Qt documentation clearly reveals that Qt was built using OOD and OOP principles.

3.4 Languages

In keeping with the OOD, the language chosen also needed to be OO friendly. Recall some of the key features of the FFD tool: it must be easy to code, it must be simple to use, and it must have multi-platform capability. Although there are many programming languages available that are considered to be OO languages, some have more benefits than others. For example, although C++ and Java are OO languages, the required syntax for each of these languages takes an experienced programmer. One missed semicolon can result in a great deal of grief for an inexperienced programmer! Python, on the other hand, is considered to be one of the simplest OO programming languages to learn because of the simplicity of its syntax. As an example, a for loop for C++ and Python are compared below:

```
for(int i=0; i<10; i++)
        std::cout << i << ‘‘ ’’ << std::endl;
```

```
for i in range(10):
        print i, ‘‘ ’’
```

The C++ snippet is the top piece of code, and the Python snippet is the bottom. Although the loop is a simple loop, the simplicity of Python’s syntax is still evident.

Python is a dynamically typed language that was developed with the OO framework in mind [10][13]. As a direct result of not having to worry about object data types, it is usually more natural to create a functional OO program in Python than any other language [13]. Python is also cross-platform, running on all major operating systems and even on Nokia mobile phones. It is also free and open source which makes it easier for people to obtain, both for personal and commercial use. There are also Qt and OpenGL bindings readily available for Python. As a result, Python was chosen as the programming language for the FFD tool. Although it was built with OO in mind, Python does not allow information hiding because of the lack of 'private' data; however, the other four requirements for a programming language to be OO are satisfied.

CHAPTER 4

MATHEMATICS AND ALGORITHMS OF FFD

In order to fully understand FFD, the mathematics behind the construction of the FFD lattice needs to be investigated. A 2D FFD lattice can be thought of as a lattice of n x m control points that surround the input mesh. These n x m control points serve as the interface between the user and the input mesh. Within this lattice, the mesh is affected by the manipulation of the control points using a specialized Non-Uniform Rational B-Spline (NURBS) entity known as a Bézier curve.

4.1 NURBS Curve

NURBS have become a widely accepted standard for the representation of geometric information in computer processing. Over the past several years, the success of NURBS is due largely to the fact that the algorithms are very fast and numerically stable. They also provide a common mathematical basis for representing both analytic shapes, such as circles and spheres and free-form shapes, such as an airplane or human body [12][8].

Mathematically, a pth-degree NURBS curve is defined by its degree, control points, knots, and evaluation rule [12]. Here, *degree* is a positive whole number that refers to the type of curve, i.e. a linear curve is degree 1 where as a cubic curve is degree 3. Furthermore, a NURBS curve of degree p is said to be a of order $p + 1$. A pth-degree

NURBS curve always has at least $p + 1$ *control points* which are used to manipulate the NURBS curve associated with them. Each of these control points have a weight associated with them, and when every control point has a weight of exactly 1, the curve is considered "rational." *Knots* are a list of numbers, specifically $(degree + n + 1)$ numbers, where n is the number of control points. The list is usually called a knot vector and each element is a knot value [8]. Each knot value has a multiplicity associated with it where the multiplicty is the number of times that particular knot value appears in the list. The multiplicity of a knot vector must be less than or equal to the degree of the curve. For example, for a general NURBS curve of degree 3 with 11 control points, the knot vector given by:

$$U = 0, 0, 0, 1, 2, 2, 2, 3, 7, 7, 9, 9, 9 \tag{4.1}$$

satisfies the requirements to be list of knots. None of the knot values appears more than 3 times (i.e. none of the knot values have a multiplicity $>$ degree). The multiplicity of the knots always has an effect on the appearance of the curve. For example, a NURBS curve is less smooth when there are multiple values of a knot in the middle of the knot vector. The *evaluation rule* is simply the formula that returns a point on a NURBS curve [12]. The formula for a general NURBS curve is given by Equation 4.2.

$$\mathbf{C}(u) = \sum_{i=0}^{n} R_{i,p}(u)\mathbf{P}_i \text{ for } a \leq u \leq b \tag{4.2}$$

where **C** is the physical point on the curve, $\mathbf{P}_i$ are the control points and $R_{i,p}$ are the rational basis funtions and are given by Equation 4.3

$$R_{i,p}(u) = \frac{N_{i,p}(u)w_i}{\sum_{j=0}^{n} N_{j,p}(u)w_j} \tag{4.3}$$

where w_i are the weights associated with the control points, and $N_{i,p}(u)$ are the pth-degree B-spline basis functions defined on the nonperiodic knot vector

$$U = \{\underbrace{a, ..., a}_{p+1}, u_{p+1}, ..., u_{m-p-1}, \underbrace{b, ..., b}_{p+1}\} \tag{4.4}$$

When dealing with NURBS curves, certain geometric properties are observed. Assuming that $a = 0$, $b = 1$, and $w_i > 0$ for all i, we have an important property: $\mathbf{C}(0) = \mathbf{P}_0$ and $\mathbf{C}(1) = \mathbf{P}_n$ which states that the first and last control points are the first and last points on the curve. Another important property is that the curve is changed through manipulation of the control points only. There are several other properties satisfied by NURBS curves, but they will not be detailed here [8]. See Reference [8] for a more detailed description.

4.2 Bézier Curve

In mathematics, all types of conic curves can be represented by rational functions, which are defined as the ratio of two polynomials. Rational Bézier curves are no different, and are dfined as NURBS curves with no interior knots [8] defined by Equations 4.5 and 4.7. Note here that $\mathbf{P}_i$ represents the ith control point and $B_{i,n}$ represents the ith Bernstein polynomial of degree n.

$$\mathbf{C}(u) = \sum_{i=0}^{n} R_{i,n}(u)\mathbf{P}_i \text{ for } 0 \leq u \leq 1 \tag{4.5}$$

where

$$R_{i,n}(u) = \frac{B_{i,n}(u)w_i}{\sum_{j=0}^{n} B_{j,n}(u)w_j} \tag{4.6}$$

$$B_{i,n}(u) = \begin{pmatrix} n \\ i \end{pmatrix} P_i(1-u)^{n-i}u^i \tag{4.7}$$

The rational basis functions for this curve, given by $R_{i,n}(u)$ satisfies several properties, one of which is that the polynomial Bézier curves are a special case of rational Bézier curves [8]. Polynomial Bézier curves, given by Equation 4.8, have most of the properties of rational Bézier curves, except that they are limited in the types of curves that can be represented well (i.e. no conic curves can be represented). However, because the computation of points are efficient, the numerical processing is quick and reliable, and the functions have small space requirements, the benefits can potentially outweigh the shortcomings.

$$\mathbf{B}(u) = \sum_{i=0}^{n} \mathbf{P}_{ij} B_{i,n}(u) \text{ for } u \in [0,1] \tag{4.8}$$

When evaluated, this equation will return a point, **B**, that is on the Bézier curve of degree n. This equation serves as the basis for the FFD lattice. Figure 4.1 shows a simple cubic Bézier curve and its control points. The dashed line connecting the control points is the control polygon which is a connected sequence of points used to control the shape of the curve. Notice that every point on the curve lies within the control polygon. This is not just coincidence. One of the useful properties of Bézier curves is that the curve will always remain within the convex hull of the control points regardless of how they are manipulated. A convex hull is simply the smallest convex set that includes all points in the data set, or on the curve this case. Figure 4.2 [3] shows an example of a convex hull, specifically the rubberband idea of the convex hull. The idea is to surround the outermost points in the set with a rubberband to obtain the smallest convex set of the data.

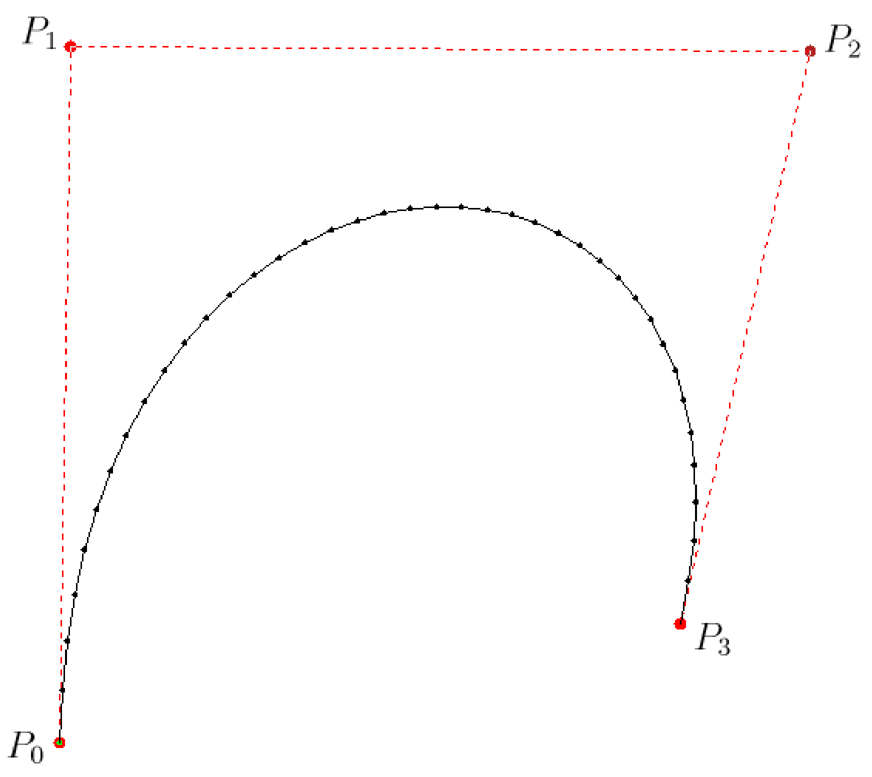

Figure 4.1

A Cubic Bézier Curve and its Control Points

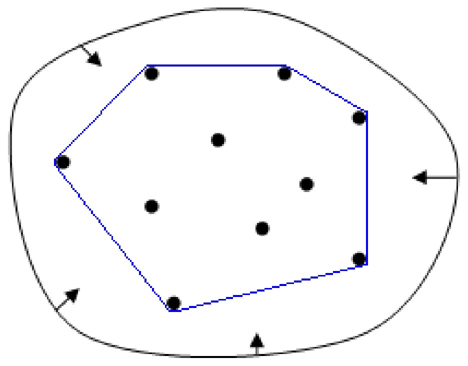

Figure 4.2

Example of a Convex Hull

4.2.1 de Casteljau's Algorithm

There are several methods to evaluate a Bézier curve [8]. One such algorithm is the deCasteljau algorithm, and the pseudocode is given by Algorithm 1[8][19]. This algorithm was the first algorithm available for drawing parametric curves. It is a so-called corner cutting algorithm that ultimately returns a single point on a Bézier curve of nth-degree. The point on a curve is calculated by repeated linear interpolation between two points, coded by Line 6 in Algorithm 1, and results in a triangular table of points as shown in Table 4.1. An important thing to remember when coding this algorithm is to save the

Algorithm 1 DE CASTELJAU(float P[], float u, int n)

```
1: for $i = 0$ to $n$ do
2:     $Q[i] \Leftarrow P[i]$ // save input
3: end for
4: for $k = 0$ to $n$ do
5:     for $i = 0$ to $n - k$ do
6:         $Q[i] \Leftarrow (1 - u)Q[i] + uQ[i + 1]$
7:     end for
8: end for
9: return $Q[0]$
```

input. If the original input is not saved, the original control points will be over written and therefore lost. This algorithm, compared to others similar to it, minimizes round-off error therefore producing better results [8]. The error is minimized due to the use of repeated linear interpolation. Figure 4.3 graphically illustrates the corner cutting process to obtain the points shown in Table 4.1.

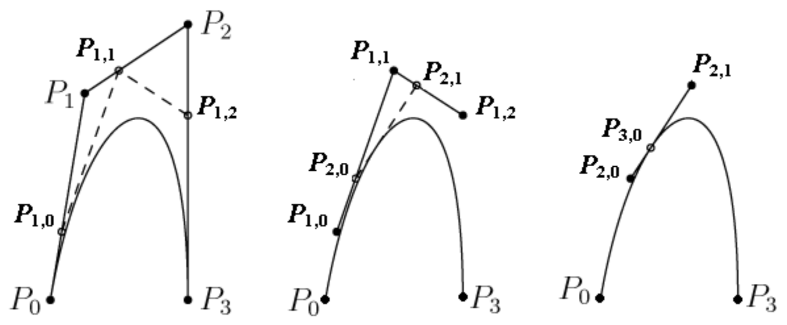

Figure 4.3

Finding points on a Bézier curve using de Casteljau's Algorithm

Table 4.1

Points generated by DE CASTELJAU's Algorithm

$\mathbf{P}_0$							
	$\mathbf{P}_{1,0}$						
$\mathbf{P}_1$		$\mathbf{P}_{2,0}$					
	$\mathbf{P}_{1,1}$						
$\mathbf{P}_2$							
$\vdots$	$\vdots$	$\vdots$		$\mathbf{P}_{n-1,0}$			
$\vdots$	$\vdots$	$\vdots$	$\cdots$		$\mathbf{P}_{n,0}$	$=$	$\mathbf{C}(u_0)$
$\vdots$	$\vdots$	$\vdots$		$\mathbf{P}_{n-1,1}$			
$\mathbf{P}_{n-2}$							
	$\mathbf{P}_{1,n-2}$						
$\mathbf{P}_{n-1}$		$\mathbf{P}_{2,n-2}$					
	$\mathbf{P}_{1,n-1}$						
$\mathbf{P}_n$							

4.3 Bézier Surface

A Bézier surface is a generalization of a Bézier curve and is defined by the following equation:

$$\mathbf{B}(t,u) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{P}_{i,j} B_{i,n}(t) B_{j,m}(u) \text{ for } t,u \in [0,1] \tag{4.9}$$

where all of the nomenclature from a Bézier curve still applies to the surface. One important property of the Bézier surface is that it is contained completely within the convex hull of the control points. This property follows directly from the fact that a Bézier surface is simply an extension of a Bézier curve. Algorithm 1 can also be generalized to two dimensions and used to calculate a point on a Bézier surface.

4.4 Relationship to FFD

All of this math is the core of FFD. The formula for a Bézier surface is used to calculate the new location of a point of a deformed object inside an FFD lattice. The entire FFD is defined in terms of a tensor bivariate Bernstein polynomial given by

$$\mathbf{X}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,n}(u) B_{j,m}(v) \mathbf{P}_{ij} \tag{4.10}$$

where $B_{i,n}$ and $B_{j,m}$ are given by Equation 4.7. Equation 4.10 is known as the deformation function, is exactly equivalent to Equation 4.9, and represents the relationship between input data, **X**, and the user manipulated control points, **P**.

CHAPTER 5

COMPONENTS OF FFD

To build an FFD, several components are needed. A key ingredient to an FFD is an input mesh. From this mesh, a bounding box is created and a list of control points are generated based on the number of subdivisions in the n and m direction. After the list of control points have been generated, a transformation of the input mesh from physical coordinates to computational coordinates is required. After a control point is manipulated, the mesh data is transformed back into physical coordinates and is redrawn to reflect the changes. These components will be discussed in more detail in the following sections of this Chapter.

5.1 Input Mesh

The input data is simply a list of discrete points that define the discrete shape of an object. The first line in the file contains the number of points that will be found in the file. Each remaining line contains the point location, in column format, with x and y coordinates separated with a space. The first eleven lines of a sample input file are shown in Table 5.1.

Table 5.1

Portion of Sample Input Mesh

38	
0.300000	0.500000
0.301186	0.512565
0.305163	0.523734
0.311934	0.533507
0.321496	0.541884
0.333850	0.548864
0.348997	0.554449
0.366936	0.558637
0.387668	0.561429
0.411191	0.562825
⋮	⋮

5.2 Bounding Box

From the input mesh, a bounding box is defined such that all points in the input mesh lie in or on the bounding box. These points are later used to establish the lower-left corner and upper-right corner points of the FFD lattice. To do this, these two points are first manipulated so that all points of the input mesh will lie totally within the FFD lattice. This will be discussed in the next section.

5.3 FFD Lattice

The FFD lattice consists of three different constituitve components. First, the mesh needs to be represented or embedded within the lattice framework and nomenclature. As a result, a function is needed to transform the physical coordinates of the input mesh to the computational coordinates of the FFD lattice. The computational coordinates of the lattice are on the interval $[(0,0),(1,1)]$. Second, control point manipulation from the user is a major component of FFD. Without manipulation of the lattice, manipulation of the embedded mesh cannot occur. Last, after a control point (or several points) have been manipulated, the mesh points as represented in computational space need to be transformed back into physical space in order to be rendered properly. These three components will be discussed further in the following three sections.

5.3.1 Transformation of Mesh to Computational Coordinates

This portion of the FFD sets up the local coordinate system [18] such that any physical point, (x,y), will uniquely map to computational coordinates, (u,v), satisfying

$$\mathbf{X} = \mathbf{X}_0 + u\mathbf{U} + v\mathbf{V} \tag{5.1}$$

where the computational coordinates can be found using linear interpolation as

$$u = \frac{x - X_{min}}{X_{max} - X_{min}} \quad \text{and} \quad v = \frac{y - Y_{min}}{Y_{max} - Y_{min}} \tag{5.2}$$

where $0 \leq u \leq 1$ and $0 \leq v \leq 1$. This can be shown in Figure 5.1 [17]. It is important to note that physical points, (X_{min}, Y_{min}) and (X_{max}, Y_{max}) map to the computational coordinates (0,0) and (1,1), respectively.

Once the computational coordinates of the mesh (u, v) are calculated, they are held fixed during the manipulation of control points.

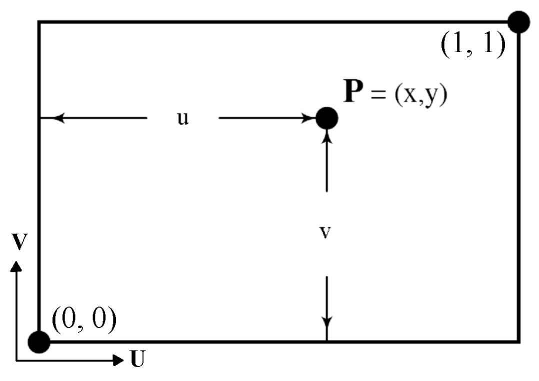

Figure 5.1

FFD local coordinates

5.3.2 Control Points

A grid of $n + 1$ x $m + 1$ control points are then imposed on the mesh such that their initial locations are defined by the following:

$$\mathbf{P}_{ij} = \mathbf{X}_0 + \frac{i}{n}\mathbf{U} + \frac{j}{m}\mathbf{V} \tag{5.3}$$

These control points are then manipulated by the user, and the updated location of the control point(s) is used to transform the computational coordinates of the mesh back to physical coordinates.

5.3.3 Transformation of Mesh back to Physical Coordinates

Recall Equation 4.10, the deformation equation:

$$\mathbf{X}(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,n}(u)B_{j,m}(v)\mathbf{P}_{ij} \tag{5.4}$$

Using the computational coordinates of the mesh which are determined using Equation 5.2, new physical coordinates are obtained when the control points are manipulated. The new physical coordinates of the mesh can be found using Equation 4.10 where (u, v) are the computational coordinates of the mesh and $\mathbf{P}_{i,j}$ are the control points, each of which may or may not have been moved by the user.

5.4 Output Mesh

Once the control points have been manipulated and the input data has been transformed back to physical coordinates, the user then needs the ability to view the output mesh. In addition, the mesh can be written to an output file that can be saved for later use.

CHAPTER 6

DEVELOPMENT OF THE CODE

The development of the code was the next logical step in the thesis process. Code for each of the major components was needed to implement the FFD tool. Built-in functions from OpenGL and Qt, as well as built-in definitions in Python proved to be very useful throughout the coding process. Also, an existing OpenGLWidget class was modified to handle control point manipulation. The development of the major components of the FFD tool are discussed in the following sections.

6.1 Picking Functionality

As mentioned in Chapter 5, one of the major components of an FFD is control point manipulation. In order to manipulate the control point, the user must be able to select (or pick) it first. This 'picking' functionality was handled using the OpenGL selection buffer. To utilize the selection buffer, a buffer and buffer size are specified. Selection mode is then entered and the objects are "rendered" to the selection buffer. A call to glRenderMode(GL_RENDER) exits selection mode and returns the records that were "hit" while in selection mode. Each record contains the minimum and maximum depth values and the object id that is uniquely associated with each control point. The minimum and maximum depth values are not used in a 2D implementation of an FFD. However, for

3D implementations, this information would be useful in determining which object was on top. The hit object id’s are stored in a list which is utilized to identify which control point(s) will be manipulated by the user.

6.2 Rendering the Mesh

The input file containing the input data must be in the format of Table 5.1 from Chapter 5. This data is read and stored accordingly in an object defined as a PointCloud object. This object has attributes for the number of nodes in the file and the position of each node. It also has functions for finding the centerpoint of the data, which is used to center the object in the window for viewing, and a bounding box, which is used when building the FFD lattice. The data stored is drawn to the screen using the standard point drawing functions in OpenGL via the OpenGLWidget object. The mesh is drawn as discrete data points and rendered using a ModelViewer object.

Table 6.1

Raw Data for a Simple 2D Vase

6	
1	7
2	6
2	2
6	2
6	6
7	7

As a simple example, consider a simple 2D vase object that can be defined by the six points in Table 6.1. This mesh data, once stored in a PointCloud object, is referred to as the FFD shape. This shape is then embedded in an FFD lattice which is discussed in the next section.

6.3 Developing the FFD Lattice

The FFD lattice was developed as an object, FFD_2d, in the BasicObjects module. This object has attributes to store the control points, the transformed mesh data, and whether or not a control point is selected. It also has functions to transform the mesh data and render the full lattice and mesh to the screen.

When an FFD lattice is first initialized using the method FFD_2d(shape, ni, nj, percent), the new FFD object contains all information required to manipulate and draw the lattice and shape. Recall that the input mesh data is stored as a PointCloud object and becomes the shape of the FFD. As a result, manipulation of the shape is equivalent to manipulation of the mesh. Upon initialization, the bounding box is determined from the shape argument, which is the input mesh data stored in a PointCloud object. The bounding box is the smallest box that can be drawn around the object, and it will typically have some of the object data on its edges. The percent argument is used to expand the bounding box so that all object points are located completely within the bounding box. A typical value for percent is 0.01 or 10 percent. This specific value expands the current bounding box by 10 percent of its diagonal.

After the "expanded" bounding box is determined, the list of control points is then populated. These points are stored in a Python built-in data structure, called a dictionary which is a mapping object indexed by a *key*. Since a dictionary cannot have an ordered pair as its *key* value, each ordered pair (i, j) has a corresponding integer instead, namely $i + (ni + 1) * j$. Figure 6.1 shows the positioning of the control points in the lattice. Note that $ni = 3$ and represents the number of subdivisions in the i direction.

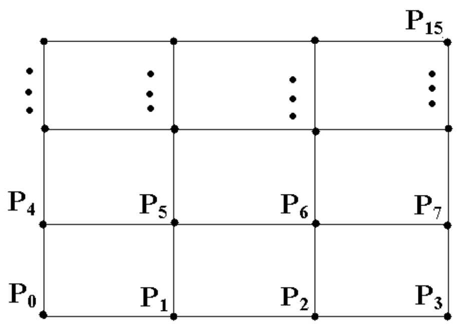

Figure 6.1

Control Point Location

Notice that $\mathbf{P}_0 = \mathbf{P}_{0,0}$, $\mathbf{P}_1 = \mathbf{P}_{1,0}$, and so on. The single integer subscripts for each point shown in the figure are the key values of the Python dictionary. The x and y coordinates, as well as a boolean attribute for selection, is stored for each control point. The x and y coordinates are found using the ni and nj arguments. Table 6.2 is an example of a list of control points with one control point being selected and the others being marked as unselected. Once the list of control points has been found, the shape data is then transformed using Equation 5.2. The lattice and mesh are then rendered to the screen. This process takes care of the initialization of the FFD lattice object. The simple 2D vase, whose data was given in Table 6.1, and its control lattice is shown in Figure 6.2.

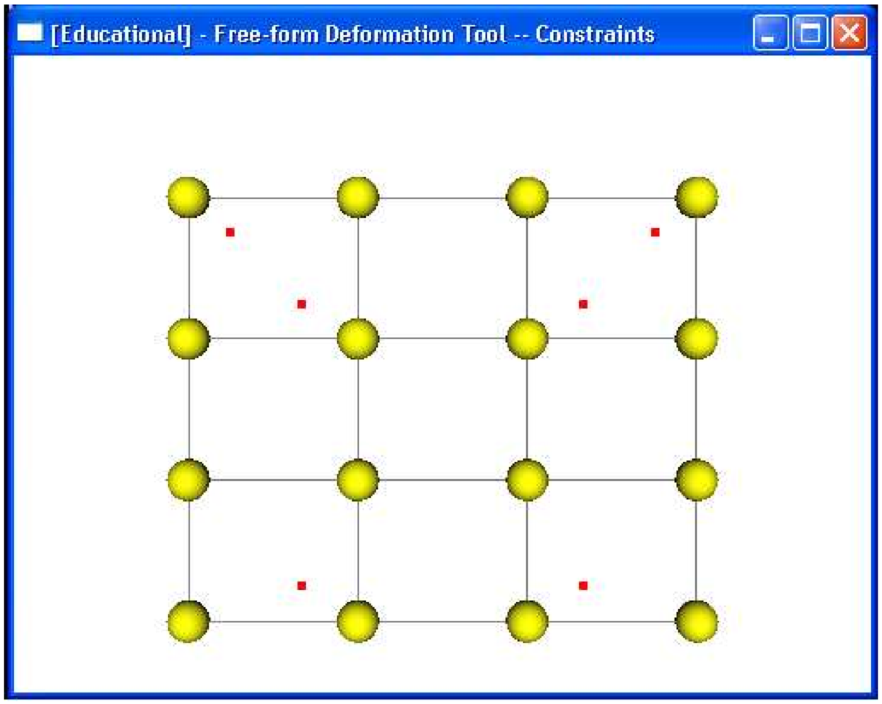

Figure 6.2

Rendering a Simple 2D Vase

6.4 Control Point Manipulation

The manipulation of control points is handled using OpenGL and Qt. The built-in Qt class QGLWidget normally handles key and mouse press events. However, the normal key and mouse press events are overloaded to specifically work for control point manipulation. Standard actions for graphics are still available such as zoom, translate, and rotate. In addition to these actions, moving a selected control point was also added.

Table 6.2

List of Control Points

0: 0.300000	0.500000	0
1: 0.301186	0.512565	1
2: 0.305163	0.523734	0
3: 0.311934	0.533507	0
⋮		

If the user presses the left mouse button and drags it, he/she will in fact zoom in or out on the scene. If instead, the middle mouse button is pressed and the mouse is then moved, the scene is rotated. Similarly, if the right mouse button is pressed and the mouse is moved, the scene is translated. Actual control point manipulation is not done until a control point is marked as selected. The user selects a control point by holding the CTRL key and clicking the desired point with the left mouse button. If the point is already selected, it is de-selected. As a constraint to maintain certain desired features, multiple control points can be selected in this way and manipulated as a group. This constraint will be discussed in the next chapter.

Once a control point has been selected, it is then eligible to be moved by the user. The difference between the original mouse press and where the mouse is currently positioned is found. This difference is then added to the position of the selected control point, and the new location is determined. The scene then requires a redraw to reflect any changes in the control points and shape. Redrawing the scene to reflect the new position of the control point(s) requires a redraw of the entire FFD object which is where the shape is updated and drawn as well.

CHAPTER 7

RESULTS AND DISCUSSION

The final FFD tool developed in this study handles 2D meshes only. Overall, the FFD tool produces robust and intuitive results.

7.1 Initialization of the FFD Tool

Figure 7.1 is a screenshot of the FFD Tool initialized with an airfoil as the input data.

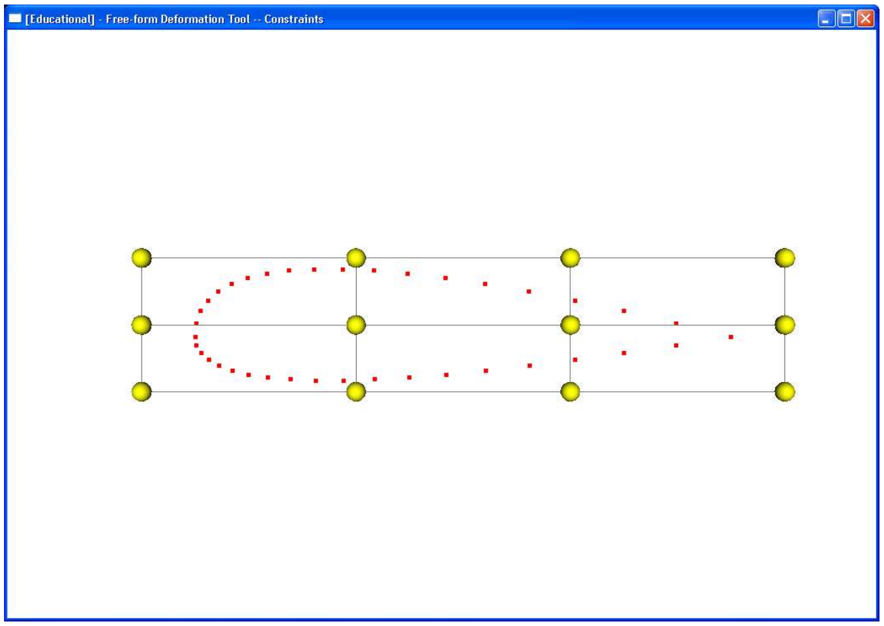

Figure 7.1

Initialized FFD Tool

In the final FFD tool, functionality for selection and manipulation of multiple control points was added. The addition of this extra functionality proves to be beneficial when manipulating different types of objects.

7.2 FFD Manipulation of Input Mesh

Any type of mesh, structured or unstructured, or any pixelated data such as graphic images can be manipulated using the FFD tool, but due to the way the mesh is displayed, only manipulation of discrete unconnected data points will be demonstrated. The following screenshots demonstrate the manipulation of several different examples of input data. Figure 7.2 shows manipulation of a standard airfoil; Figure 7.4 shows manipulation of a geometrically complex structure; Figure 7.6 shows manipulation of a blood flow microdevice; and Figure 7.7 shows manipulation of a mesh that contains multiple disconnected objects.

Notice that the airfoil in Figure 7.2 is a simple geometry, and as a result, it can be manipulated using either one point or multiple points while producing basically the same output mesh. Figure 7.3 demonstrates this. The same four control points were moved to the same approximate locations in each image in order to increase the thickness of the airfoil. In the first image, the points were manipulated individually whereas the points in the second image were grouped and manipulated at the same time. This is the reason that only one point is selected (denoted by dark colored spheres) in the first image and four points are selected in the second image. Although the use of multiple point selection reduced the time taken to manipulate the airfoil, basically the same results were achieved.

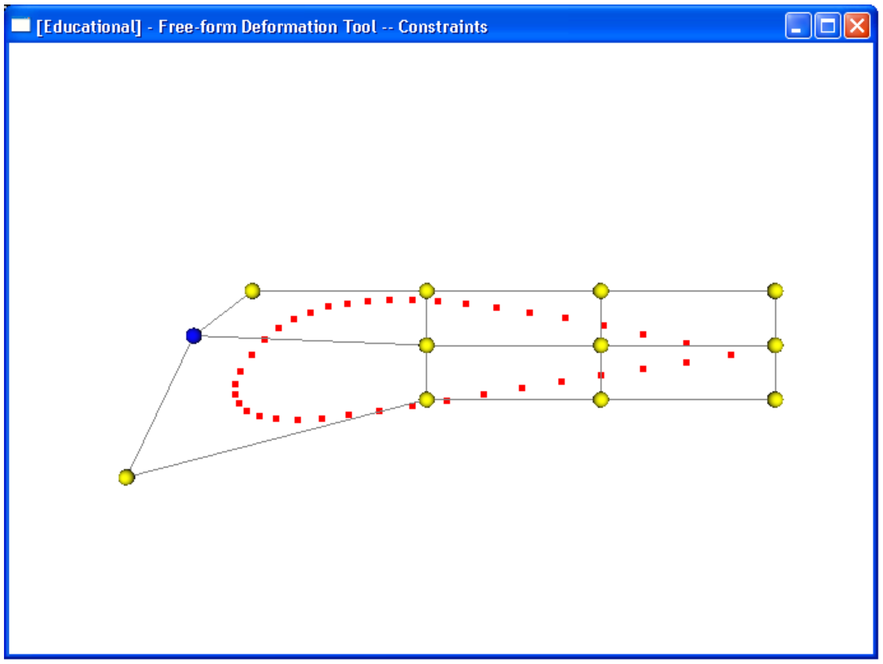

Figure 7.2

Manipulation of an Airfoil

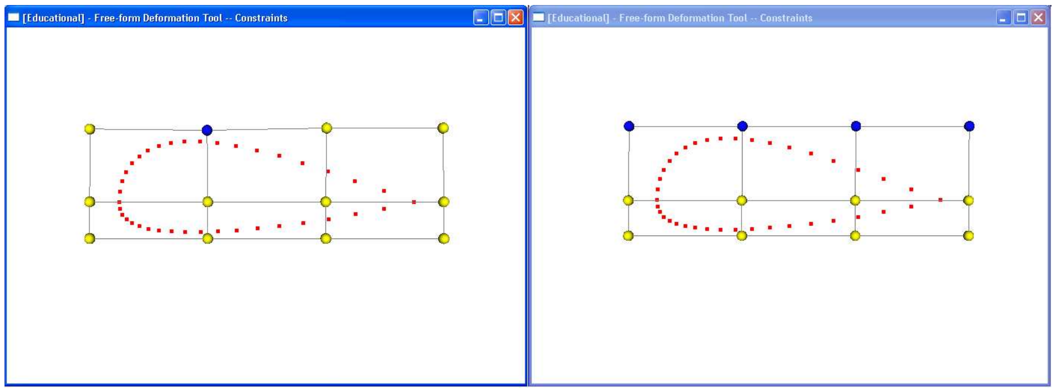
[Educational] - Free-form Deformation Tool -- Constraints
[Educational] - Free-form Deformation Tool -- Constraints

Figure 7.3

Single Point Manipulation v Multiple Point Manipulation of an Airfoil

However, it will be shown that multiple point selection/manipulation may produce better results depending on the design intent and/or geometry of the original mesh.

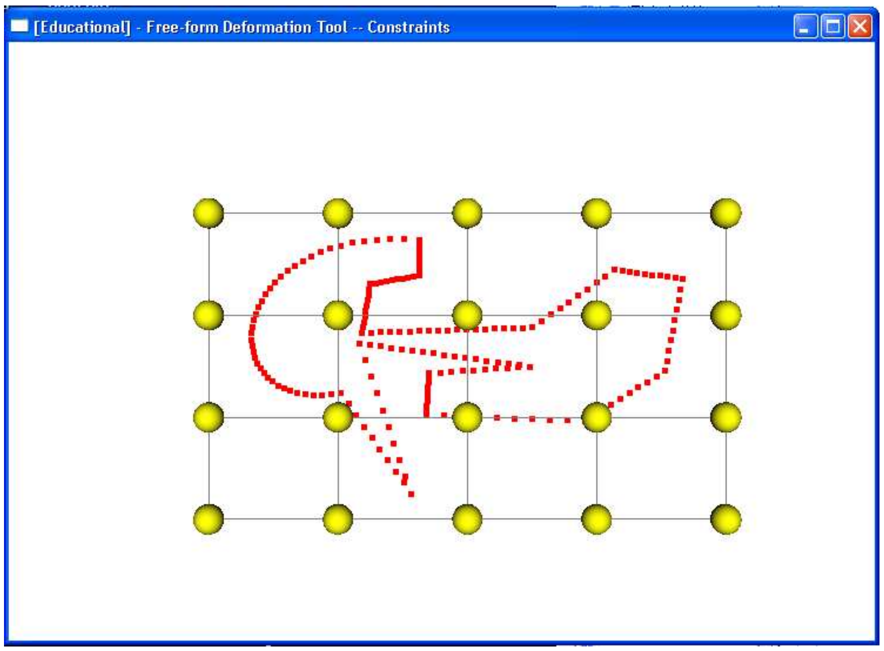

Figure 7.4

FFD Initialization of Complex Structure

A geometrically complex structure such as the one in Figure 7.4 can be easily manipulated using the FFD tool. The ability to select and manipulate multiple points at one time guarantees the user the ability to maintain certain features of the original mesh. Figure 7.5 shows the difference between manipulating one point versus manipulating a group of points. Single point manipulation is shown in the figure on the left, and multiple point manipulation is shown in the figure on the right.

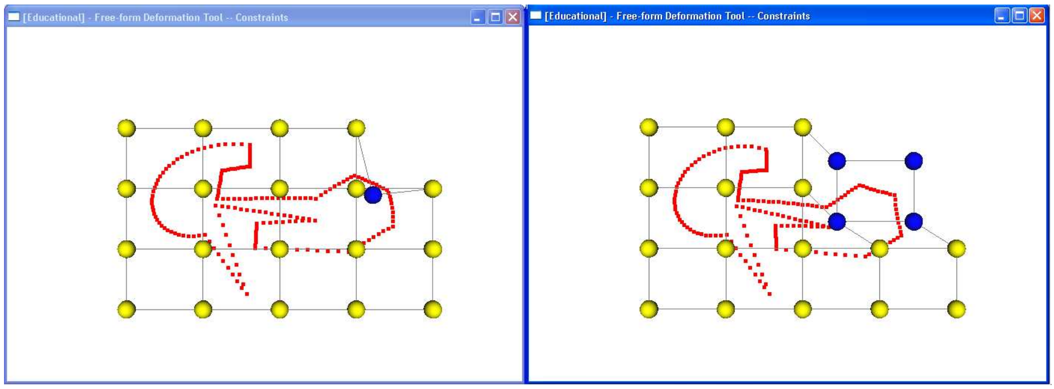
[Educational] - Free-form Deformation Tool -- Constraints
[Educational] - Free-form Deformation Tool -- Constraints

Figure 7.5

Comparison of Single Point Manipulation and Multiple Point Manipulation

An inexperienced user with the intent of shortening the rightmost extension of the complex geometry may decide to move only one control point (as in the left image of Figure 7.5). As such, the user may be disappointed to find that although the extension is shortened, the sharp corner feature is lost and is now rounded. However, if the control points surrounding the sharp corner are selected as a group and manipulated as a group (as in the right image), the extension is shortened, and more importantly, the sharp corner feature is preserved. Although the user would be able to eventually achieve the same results using single point manipulation (by moving the other three points surrounding the sharp corner), it is more convenient and efficient to group the control points and move them as one object.

Blood flow devices, such as the one used in Figure 7.6, can also be manipulated using the FFD tool. In this particular example, the length of the smaller tube connecting the two larger tubes is lengthened. This type of manipulation is in fact a practical example of bionengineering usage at Mississippi State University's High Performance Computing

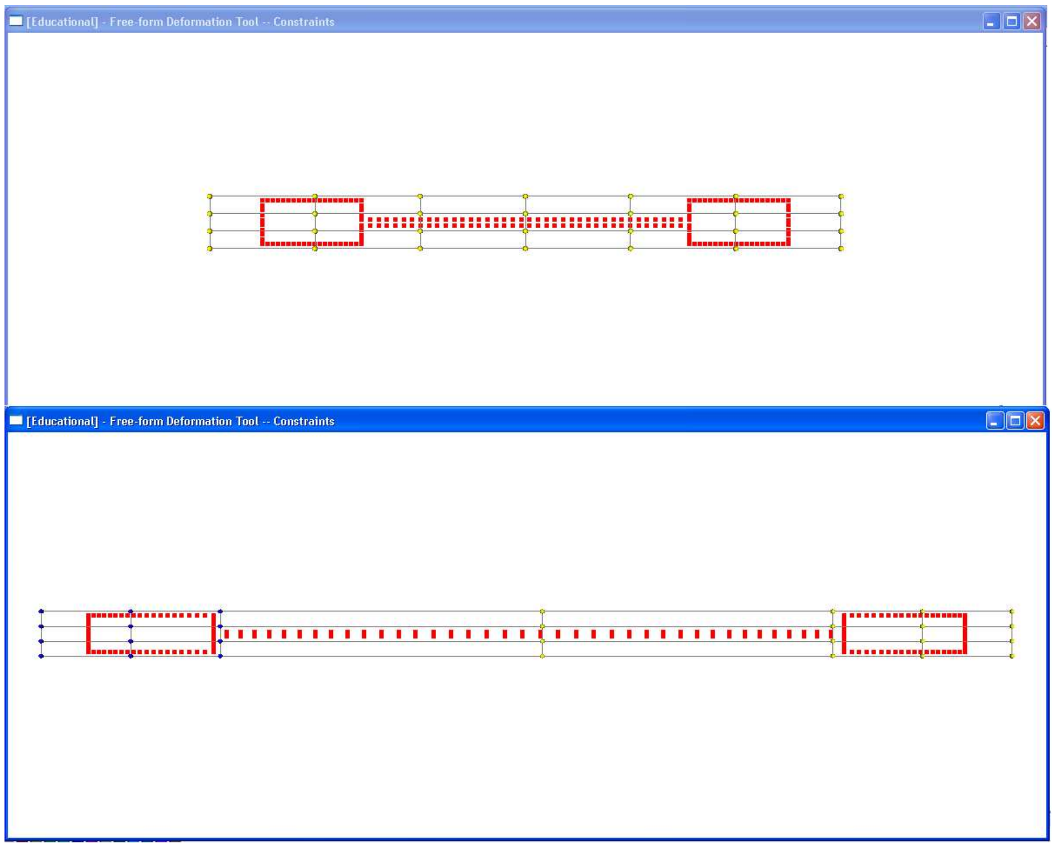
[Educational] - Free-form Deformation Tool -- Constraints
[Educational] - Free-form Deformation Tool -- Constraints

Figure 7.6

FFD Manipulation of Blood Flow Microdevice

Collaboratory. This geometry is a specific geometry studied by Tammy Ladner during her Master of Science thesis investigation of blood flow in a capillary tube [5]. Blood flow in a longer tube may exude different flow characteristics or behaviors when compared to flow through a shorter tube. The FFD tool can readily facilitate this type of geometry change. Increasing the diameter of the smaller tube is another a practical example. A user-defined FFD around that particular area would be useful to increase the diameter of the smaller tube without affecting the larger diameter tubes.

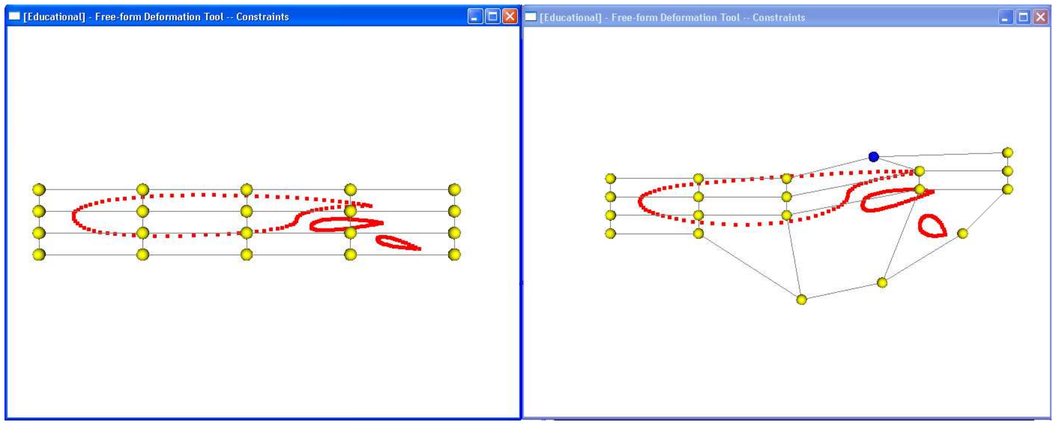
[Educational] - Free-form Deformation Tool -- Constraints
[Educational] - Free-form Deformation Tool -- Constraints

Figure 7.7

FFD Manipulation of Mesh with Multiple Objects

Finally, manipulation of a mesh with multiple distinct objects also produces efficient results, depending on the intent. Figure 7.7 represents a multi-element airfoil. Aerospace engineers may wish to see how changes to the larger airfoil affect its aerodynamics. The FFD tool works perfectly for this intent. The image on the right demonstrates single point manipulation to lift the tail and round the “cargo” area of the larger airfoil.

In summary, the FFD tool produces intuitive and robust results, especially when multiple points are selected and moved at one time. Using this feature allows more features of the original shape to be maintained.

CHAPTER 8

CONCLUSIONS

The objective of this thesis was to build an innovative, unified, object-oriented free-form shape deformation tool with Bézier basis functions serving as the foundation for the deformation lattice. This tool was to focused on manipulating discrete geometries for computer-aided engineering analysis and design.

A free-form shape deformation tool was developed. Examples of manipulation of several types of discrete meshes were shown in Chapter 7. The decision to use OOD and OOP proved to be valuable. OOD aided in designing the program, and OOP aided in organizing and debugging the code.

This FFD tool can serve as the basis for a 3D shape deformation tool. Because it was built on OO principles, it should be easy to modify and add additional functionality. The use of Python will make it accessible for anyone to update because it is free, open-source, and easy to learn due to the simple syntax.

The FFD tool developed was object-oriented and used a Bézier surface as the basis for the deformation lattice. As a result, the objective of the thesis was achieved.

CHAPTER 9

FUTURE WORK

Work could be done to the FFD tool to make it user friendly and more functional to engineers and designers. Some suggestions would be:

- Add a GUI
- Add functionality for an embedded FFD
- Add functionality for a user-defined FFD
- Expand to 3D while maintaining functionality

The addition of the above mentioned items would make the program more user friendly, allow the user to focus manipulation on one section of the mesh, and give the user the ability to manipulate 3D objects in addition to 2D objects.

Addition of a GUI with standard pull down menus would prove to be very useful. For example, adding a pull down "File" menu with the ability to "Open", "Close", "Save", and "Exit" would allow the user to start on a new FFD without losing data or having to restart the program. An "Edit" pull down menu would also be useful if the functionality to "Redo" and "Undo" changes was added. This would allow the user to decide whether they like the most recent change or not. Currently, no functionality has been implemented for any of this.

Two additional modes of shape manipulation could be added to the present capabilities that would improve its usability under certain circumstances. These will be referred to as: embedded FFD and user-defined FFD. The basic idea of the embedded FFD is to enclose or "embed" a selected subset of the primary lattice's control points within a separate enclosing FFD. Hence, the enclosed primary control points become the input data to the embedded FFD. Manipulation of the embedded FFD would directly modify the control point locations of the primary FFD lattice and, as a consequence, modify the shape of the primary geometry data. This would enable, for example, a smooth and coordinated manipulation of a higher order (many control points) primary lattice via the manipulation of an enclosing lower order (fewer control points) embedded FFD. The basic idea behind a user-defined FFD is one of localized shape modification to geometric data. This FFD would be defined via a user-defined bounding box that surrounds the region of interest desired for shape modification. For example, one may desire to change the shape of only the middle airfoil of a multi-element airfoil geometry. A user-defined bounding box would be drawn to enclose the middle airfoil shape, and the resulting user-defined FFD would be populated with control points (see Figure 9.1). Manipulation of this FFD would locally shape-modify the middle airfoil only, leaving the remaining adjacent geometry unchanged. For tightly packed geometries, the user-defined bounding box may unavoidably enclose whole or partial geometry data from adjacent shapes. In this case, a deselection mechanism would be needed to remove the undesired geometry data from the user-defined FFD.

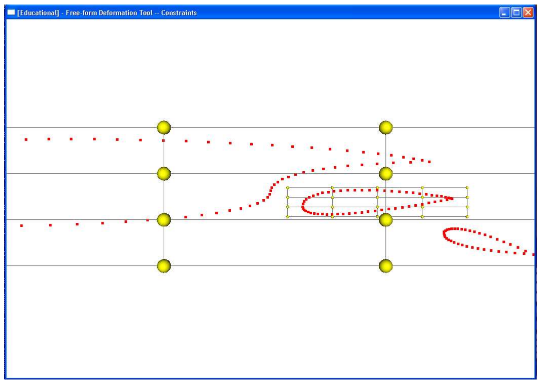

Figure 9.1

Preliminary User Defined FFD

Finally, extension of the FFD tool to 3D would give the user the ability to handle any type of discrete mesh. Once expanded, the FFD tool would be more useful for realistic problems of interest.

REFERENCES

[1] A. H. Barr, "Global and Local Deformations of Solid Primitives," *Computer Graphics*, vol. 18, no. 3, 1984, pp. 21–30, Proceedings of SIGGRAPH 1984.

[2] Y.-K. Chang and A. P. Rockwood, "A Generalized de Castlejau Approach to 3D Free-form Deformation," *Computer Graphics*, 1994, pp. 257–260, Proceedings of SIGGRAPH 1994.

[3] "Convex hull," Wikipedia, 2007, http://en.wikipedia.org/wiki/Convex_hull.

[4] A. Ferrier, "Real-Time Soft-Object Animation Using Freeform Deformation," Gamasutra, 1999, http://www.gamasutra.com/features/19990827/deformation_01.htm.

[5] T. Ladner, *Characterization of Blood Flow in a Capillary Tube*, master's thesis, Program of Computational Engineering, Mississippi State University, Starkville, MS, August 2007.

[6] J. Lander, "In This Corner... The Crusher," *Game Developer Magazine*, 2000, pp. 17–22, http://www.darwin3d.com/gamedev/articles/col0600.pdf.

[7] "Object Oriented Design," Wikipedia, 2007, http://en.wikipedia.org/wiki/Object-oriented_design.

[8] L. Piegl and W. Tiller, *The NURBS Book*, 2nd edition, Springer, New York, 1997.

[9] "PowerCLAY Overview," Exa Corporation at www.exa.com, 2003, http://www.exa.com/newsite/powerclay/powerclay_overview.html.

[10] "Python Programming Language," Python's Official Website, 2007, http://python.org/.

[11] "Qt - Cross-Platform C++ Development," Trolltech, 2007, http://www.trolltech.com/products/qt/features/index.

[12] "What is NURBS?," Rhinoceros® NURBS Modeling for Windows, 2007, http://www.rhino3d.com/nurbs.htm.

[13] C. Roach, "Programming Python," *MACTECH*, vol. 21, no. 3, 2007, http://www.mactech.com/articles/mactech/Vol.21/21.03/ProgrammingPython/index.html.

[14] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes, “Nonrigid Registration Using Free-Form Deformations: Application to Breast MR Images,” *IEEE Transactions on Medical Imaging*, vol. 18, no. 8, August 1999, pp. 712–721, http://www.cs.jhu.edu/c̃is/cista/746/papers/RueckertFreeFormBreastMRI.pdf.

[15] J. A. Samareh, “AIAA-2000-4911: Multidisciplinary Aerodynamic-Structural Shape Optimization using Deformation (MASSOUD),” *8th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Long Beach, California, September 2000, citeseer.ist.psu.edu/samareh00multidisciplinary.html.

[16] “Sculptor Overview,” Optimal Solutions, 2005, http://www.optimalsolutions.us/optimalsolution/software.

[17] T. Sederberg, “Chapter 8: Free-Form Deformation,” www.cs.byu.edu, 1997, http://students.cs.byu.edu/ tom/557/text/ch8.pdf.

[18] T. W. Sederberg and S. R. Parry, “Free-form deformation of solid geometric models,” *Computer Graphics*, vol. 20, 1986, pp. 151–160.

[19] C.-K. Shene, “Finding a Point on a Bézier Curve: De Casteljau’s Algorithm,” http://www.cs.mtu.edu/ shene/COURSES/cs3621/NOTES/spline/Bezier/de-casteljau.html.

[20] A. Sintes, *Sams Teach Yourself Object Oriented Programming in 21 Days*, 3rd edition, SAMS Publishing, Indianapolis, Indiana, 2002.

[21] C. H. Swaroop, “A Byte of Python,” Creative Commons Attribution-NonCommercial-ShareALike License 2.0, 2003, http://swaroopch.info/text/Byte_of_Python:Main_Page#License.