A Finite Element Approach for Virtual Clay Simulation Using Boundary Cell Encoding

By

Ajay Anand

Department of Mechanical and Aerospace Engineering
State University of New York at Buffalo
Buffalo, New York 14260

A thesis submitted to the faculty of the Graduate School of the State University of New York at Buffalo in partial fulfillment of the requirements for the degree of Master of Science

January 2006

Acknowledgements

First of all, I thank Dr. T. Kesavadas for his guidance as well as his invaluable support and encouragement. I would like to thank him particularly for his remarkable idea of using subdivision surface based LPF. Without this contribution, the work would have been finished but not presentable.

My deepest gratitude to Dr. T. Singh for being in my committee. Special thanks are due to my committee member Dr. A. Patra for giving valuable suggestions and insights during the research.

I would like to thank Amrita, Govind and Kim for their resourceful help. I thank staff members, Bonnie Boskat and Carole Naab, in whose presence I never had to worry about any academic formalities during my busy research life. Finally my hard working colleagues will always remain my good friends and a source of inspiration.

Dedicated to

People of the Underdeveloped World

Table of Contents

List of Figures

List of Tables

Abstract

Despite of the facts that clay is a popular modeling medium and in the last one and a half decade a lot of work has been done on virtual clay modeling and sculpting, there are only a few commercial tools for this purpose. This can be explained by two reasons. Firstly, the fast and robust systems do not mimic the properties of clay accurately, making them less popular than freeform tools. Secondly, systems that represent the physical properties of clay in a realistic manner need high-end and specialized computation and display hardware which makes them less viable. In this work we explore ways to overcome the difficulties with previous systems. We address these problems by a novel method to represent virtual clay that fuses the best features of FEM, generalized subdivision surfaces and data compression techniques.

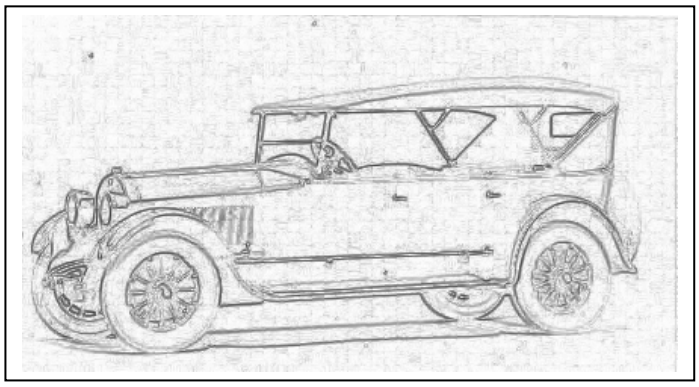
Chapter 1: Introduction

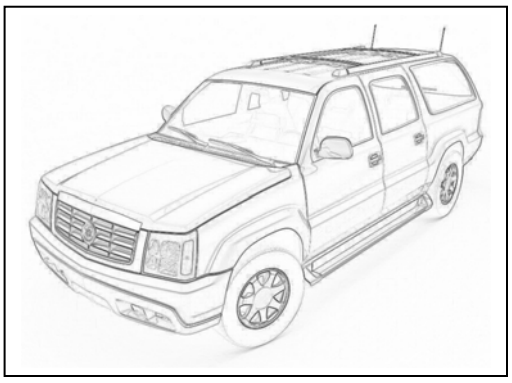
Over the years, product design has gone dramatic change in terms of visual aesthetics. Earlier practices laid more stress on functionality, therefore the conventional designs were more mathematical (Euclidian) and the geometrical features were more prominent in the final design (Figure 1.1). With scales and compass as the basic drawing instruments, the drawings were infiltrated with lines, arcs and French curves. During the last few decades, with the popularity of PC's and advancements in computational and graphics performance, computer aided design and drawing has become very common. The basic shapes and curves have become more intuitive with machine computed parametric shapes and curves. Modern design aims to strike a balance between the functional and aesthetic needs.

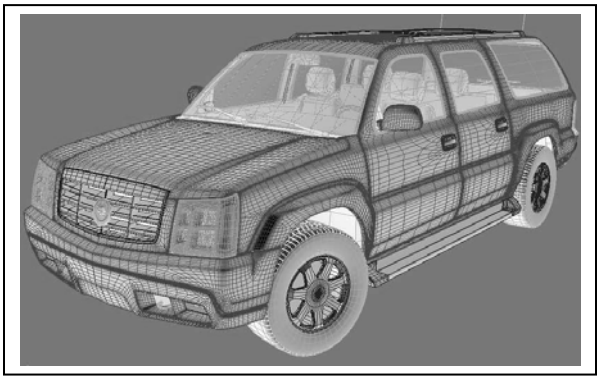
The down side of current CAD systems is that they are very complicated. Most of them have a learning curve associated with them. The user interface usually consists of mouse and keyboard. Both mouse and the screen are ideal for 2D user input. Even though they can be utilized to edit 3D objects, that requires changing the view frequently either by rotating the object or by switching between view ports.

The current CAD models are also passive, that is, they do not have active physical properties. The main reason behind this is that most 3D objects are surface based, which prevents them from having volumetric properties. Most complex 3D shapes that

(a)







(b) (c)

Figure 1.1: CAD systems made design more intuitive (a) Cadillac 1924 Model, (b) Cadillac Escalade 2004 Model, (c) Computerized Model of Cadillac Escalade 2004 (*Source:* WWW_Exchange3d)

are tree or hierarchy based (e.g. B-Reps, BSP) lack volumetric information. Real objects, apart from being volumetric, exhibit elasticity, plasticity, mass conservation and many other physical behaviors. Working with passive objects does not make modeling easy because user still may not get the intended shape by indirect manipulations of parameters and control points. Moreover it is impossible for user to manually manipulate each point on the surface consisting of thousands of vertices. The best way to avoid this complication is to give active physical properties to CAD models. Designers already have some experience of manipulating and deforming real life objects hence this favors manipulating complex models as if they were real objects.

1.1 Sculpting Clay

Clay is a popular modeling medium. The methodology of working with clay has been finely developed over generations and it is a very intuitive way of creating realistic objects. But modeling it is an unsolved problem too. The physical properties of clay change dramatically if one of the components, say moisture is varied. Unlike resins, soil clay is much more sensitive to compositional variations.

Sculpting is an art. We have seen that in the area of artwork like painting and drawing, computer software provide artists with tools and facilities impossible to achieve with physical canvas. Virtual clay too has several benefits over real clay. For example, layer by layer material addition or removal, engraving fine details, resizing, replicating the already made object, saving and retrieval of work (no dry out period), easy templates (analogous to wire fixtures), low cost of storage of models, possibility of direct integration with prototyping / manufacturing system and so on.

Here onwards, the keyword 'virtual sculpting' will be used to imply 'computer aided sculpting of clay like virtual objects'.

1.2 Essentials of a Virtual Sculpting System

The objective of using a CAD system is to design with the benefits that come with the use of computer. To achieve this objective, four basic components must be provided by every CAD system, based on (Requicha 1980):

1. *Data Structures that represent CAD objects*

   For example a line segment can be represented by its end points. The coordinates of the end points can be stored in memory with resolution allowed by number of bits assigned to each coordinate.

2. *Manipulation of CAD objects*

These are operations on the data structures which translate to physical manipulations on real objects. Such a mapping may be difficult to achieve.

3. *User Interaction*

The manipulations should be carried out according to user specified operations.

4. *Feedback*

The user may be able to see (or feel) and assess the results of the manipulations.

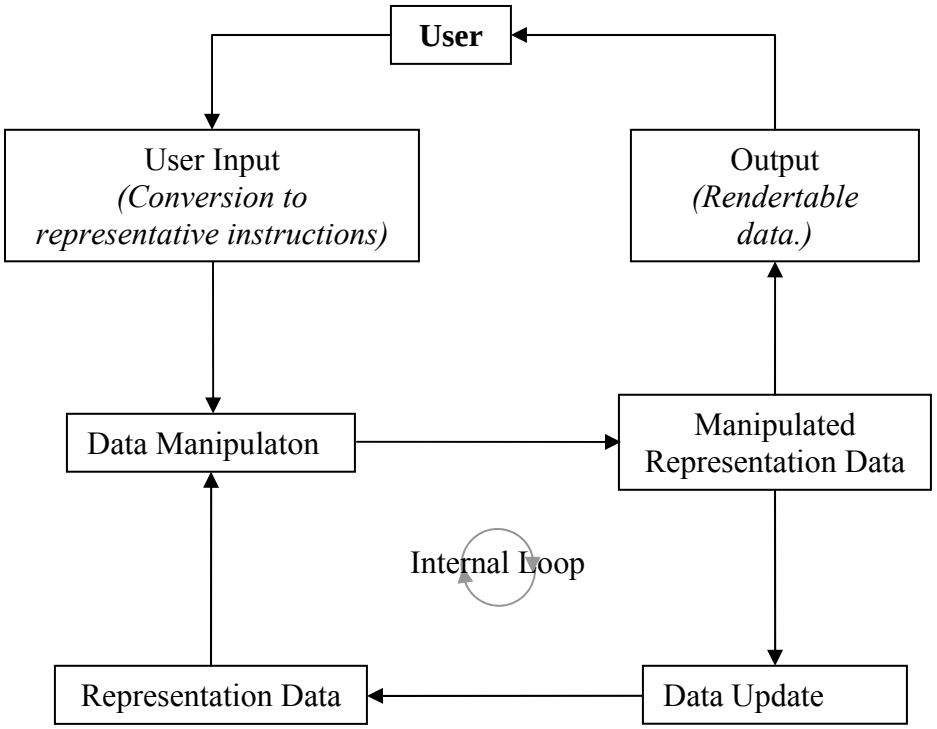


Figure 1.2: Essential components of an interactive CAD system

To determine a suitable CAD system to carry out virtual sculpting it is required to look at the restrictions or special requirements that are imposed on the above mentioned features in a virtual sculpting paradigm.

1. *Data Structures that represent clay objects*

   Clay is a solid object but unlike embedded solids, the sculptor is rarely interested in interior properties. The operations that are carried out are solely done on the surface. In an ideal case, the data will represent information about both, surface and volume.

2. *Manipulation of clay objects*

   Viscous, plastic and damped elastic properties of clay require direct manipulation of volumetric representations whereas surface tension requires information about surface. Similarly actions like pressing clay causes volume to flow while actions like joining involve surface as the interface.

3. *User Interaction*

   Conventional CAD objects can be decomposed into contributing points and curves. Using such technique will be good for geometric models but very non intuitive for a real life object like clay. This demands a need for manipulation of the model in 3D directly. Feeding single point force input is also easy to achieve. Multiple point force input requires special tactile or haptic devices.

4. *Feedback*

   Displaying the model is easy if surface representation is readily available. Providing haptic feedback will be a challenging task.

1.3 Clay Modeling as Practiced

At this point, the actual sculpting methodology deserves a review. To create a miniature sculpture one requires suitable material, equipment, accessories and skills (WWW_ Lyonstudio).

1. *Material*

   Epoxy putty is most popular clay medium. Clay materials have curing time which is defined as the time after that they have completely dried out. Epoxy has one of the longest curing period (about an hour). The curing period can be shortened or prolonged by changing the environment temperature. Epoxy retains some amount of elasticity after curing and is easy to work with.

2. *Tools*

   Most essential tools are steel spatulas, knife (X-acto) and very fine sandpaper.

3. *Accessories*

   Brass wires are required to create skeletal figures. Working with brass wires requires nose pliers, razor saw, wire nippers and soldering iron. Heat lamp is needed if some parts need to be cured faster.

4. *Technique*

   The underlying idea of miniature sculpting technique is to create a dimensionally proportionate wire figure of the object one intends to sculpt and then start building the model over it. For example, if one wants to create a human model, a stick figure of a man or woman is built using brass wire, bent in such a way to exhibit intended posture. The shoulders, length, feet should be dimensionally in correct proportion. The figure is secured by providing it a wooden or similar base.

A thin layer of putty is added on the stick figure. During this phase, information about muscles and clothing can be marked on the surface. Next, each body part is built one by one by adding more putty. While the parts are built, special care needs to be taken to maintain the proportion of each part. Details can be added later. Small protruding details can be stuck separately. Tricky methods may be required to build complex objects like creating holes to make a chain. Processes like riveting can also be done. The basic operations can be isolated and listed as tugging, pressing (to apply layer of material), slicing, and poking a hole

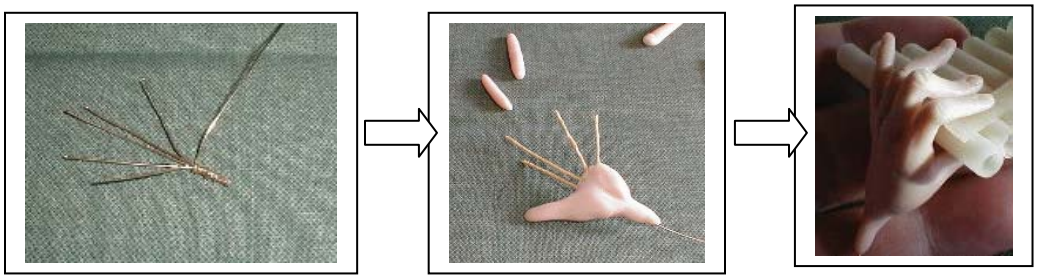


Figure 1.3: Sculpting a human hand (WWW_Faeryforest)

It becomes quite evident now that the operations that can be done on clay are not numerous and only one or two are needed most of the time during sculpting process. It is the exquisite properties of clay which really make it one of the best modeling media.

1.4 Thesis Outline

Chapter 2 defines the problem statements, the scope of current research and reviews classic and contemporary work done in the field of virtual sculpting. Chapter 3 describes in details the concepts that have been used in past for handling, manipulating and rendering virtual clay. Chapter 4 gives an overview and then details of our solution approach. Various data structures and algorithms used have been described in Chapters 5

and 6 respectively. Statistical tests and their analyses are presented in Chapter 7. In Chapter 8, the options for increasing the scope of current work have been suggested followed by concluding remark.

A set of ten appendixes provide information about material properties, FEM and subdivision related derivations and maple codes for stiffness matrix calculation. Apart from that detailed observation tables and graphs for statistical tests have been compiled in the last four appendixes.

Chapter 2: Research Objectives and Background

2.1 Problem Definition and the Scope

To understand the goal of this work it is required to split the sculpting process into three components

a. Sculptor

b. Clay Model

c. Intermediate Interaction System (I/O devices)

These components can be further broken down into subcomponents. For example interaction system can be split into three subcomponents viz. input device, output device and computation device.

Among the above mentioned components, this thesis is more concerned with development of a suitable “Clay Model”. Sculptor behavior and intermediate system are not in the scope of current work. The theme of this work is to develop an appropriate software model that is well suited for virtual clay applications. The underlying guidelines to achieve this goal are:

1. Enabling user to sculpt 3D CAD models that exhibit physical properties of clay during modeling process
2. Natural clay-tool interaction
3. Realistic influence of environmental constraints on clay
4. Realistic appearance
5. Interoperability with other CAD systems

Though the last guideline does not strictly match with previous four, it was important to consider translation between the new model and existing CAD models because interoperability combines the power of different systems. **In a nutshell, the objective of current work is to create a *physically based, real-time* modeling system which *enables arbitrary shaped clay object to be manipulated by an arbitrary shaped tool* rendering *realistic models* with basic *interoperability* with other CAD systems.**

2.2 Previous Work

A lot of work had been done on surface and solid manipulation (Barr 1984, Cobb 1984, Sederberg and Parry 1986, Wyvill *et al.* 1986, Terzopoulos *et al.* 1987), it was the pioneering work by Galyean and Hughes (Galyean and Hughes 1991) that introduced a practical virtual sculpting process that opened a new vast stream of research in this area. The present work also falls into that category. They achieved volumetric operations on clay using a volumetric representation of clay data which they called Voxmap. Using Polhemus Isotrack device to control the position and orientation of the sculpting tool they facilitated the process of carrying out local deformations on the volume in 3D space. Their sculpting schemes consisted of "toothpaste tube" to add material and "heat gun" and “sandpaper” to remove different amounts of material. Their method has been used by Ferley *et al.* (Ferley *et al.* 2000), Perng *et al.* (Perng *et al.* 2001), Hui and Leung (Hui and Leung 2002) and more recently Dewaele and Cani (Dewaele and Cani 2003).

CSG based modeling methods have also been used in which virtual chisel performs carving operation on a workpiece by Boolean addition and removal of chisel shaped pieces. Thus very complex forms can be created by repeating the process. Mizuno *et al.* (Mizuno *et al.* 1998a) used such a method to carry out sculpting with a chisel that

could be an ellipsoid, a cube, a cylinder or combination of them. They also provide a facility of virtual printing (Mizuno *et al.* 1998b, Mizuno *et al.* 1999a, Mizuno *et al.* 1999b).

Complex sculpting tools were described in Ferley *et al.* (Ferley *et al.* 2000) in which they could add, remove, paint, smoothen and stamp on virtual surface. The material structure was voxel based and the operations were carried out by interactions between spatially sampled scalar fields of tool and the object. For rendering purpose, they created an isosurface for the tool and clay object. Apart from this they had freeform tool and the tool shape could be user defined. To control the tool they used, Spacemouse or virtual trackball with a mouse. McDonnell *et al.* (McDonnell *et al.* 2001) provided more sophisticated sculpting facilities by incorporating topological and physics based tools. Their topological tools could perform deletion, extrusion, local subdivision and sharp feature generation. The physics based tools included curve manipulation, stiffness and mass painting, deflation and inflation. An artistic tool for spray painting was also been provided. In their work they had integrated several approaches like free-form and spline-based solids and procedural subdivision solids.

Perng *et al* (Perng *et al.* 2001) used 3D Tracker with 6DOF to handle the sculpting tool. Their system could perform carving, stuffing, pottery making and painting operations. They achieved this by changing the state of voxels near the boundary of cutting tool. Hui and Leung (Hui and Leung 2002) used FEM on boundary elements to deform surface. They generated voxel based volumetric data for addition and removal of material.

Perry and Frisken (Perry and Frisken 2001) used Adaptive Distance Fields (ADF) on CSG models. Kuester *et al.* (Kuester *et al.* 2002) used ADF with their own surface generation method to perform modeling. Dewaele and Cani (Dewaele and Cani 2003) in their volumetric representation used three coherent layers that handled large scale deformations, mass conservation and surface tension respectively. They basically used the fact that real clay had properties of both viscous fluids and plastic solids. They used cells in 3D volume, density function and isosurface extraction features. They could achieve bending, folds, drilling holes and making prints.

Kameyama (Kameyama 1997) developed a haptic CAD system to give the sculptor a realistic sculpting experience of holding and touching the material though a haptic device. The author used a special input device which consisted of a 3-D position tracker and a tactile sensor. The position tracker was used to move the virtual object and the tactile sensor captured the amount of force to be applied on the virtual object. The system was specially designed to be used for direct virtual sculpting process. The type of surface that he actually manipulated was flat and the manipulative motion was only vertical. The surface was polygon based and used a data translator with a solid modeler for producing solid model data. The polygon data was displayed using a *graphical viewer* software. The system could be interfaced with a production system for manufacturing the solid model.

In a similar work, Kamerkar and Kesavadas (Kamerkar and Kesavadas 2004) used a pressure sensitive input device called ModelGlove to edit NURBS surfaces. Their input device consisted of force and 3D position sensors worn on hand in the form of a

glove. A virtual block made up of NURBS surfaces was deformed using pressure and tool position input from the device.

Sensable Technologies markets a commercial volume sculpting system that uses voxels for sculpting (WWW_Sensable_A) and it has also released a plugin for Rhino™ to carry out virtual clay sculpting (WWW_Sensable_B).

In our work, we have utilized the concept of using discrete elements (voxels) to model solids. Unlike conventional ways to handle voxels, we propose a compressed format to store voxel data. We develop time and memory saving algorithms that operate directly on the compressed data and enable development of faster algorithms needed for collision detection, Boolean operations and surface generation. Furthermore, our system simultaneously maintains volumetric and surface representations of virtual clay in a compressed format. We also introduce a new concept of incorporating an embedded layer of updated Lagrangian FEM solver that derives its structure from volumetric data and updates the surface data. For better rendering and export of the aliased clay model, we derive smoothing filters from the equations of subdivision surfaces. Finally, the ability to import and export data in standard CAD formats makes the work complete.

In next chapter different aspects of the techniques used by above mentioned authors and additional concepts related to virtual sculpting will be discussed.

Chapter 3: Virtual Clay Concepts

Constructing a virtual world requires the ability to quickly and accurately simulate the physics of complex, multibody environments (Pentland 1990). Pentland argued that the state of simulations in 1990 was simple and static and that they had to rely more on simulation of physics in complex environment. Pentland identified four critical areas as rendering, dynamics, collision detection and constraint satisfaction. Even today with all the advancements in computation and computer hardware, all of them are crucial in virtual sculpting more than any other system. In this chapter we will look at different schemes for data representation, manipulation and display of virtual clay keeping on mind their influence on the four critical areas.

3.1 Surface Representation

Almost all 3D modeling packages have extensive tools to manipulate surface models. Real-life objects are rigid and opaque. We see and touch their outer surface only. Practically every object can be represented as surface in the virtual world. In this section we will go over different available methods for surface representation. The methods listed are one of those which have been used in virtual sculpting implementations.

3.1.1 Mesh

A mesh is a well connected structure of polygons. Irrespective of how the surface is represented, it has to be displayed as a mesh of polygons. Few things need to be considered while modeling 3D objects using meshed surfaces. Firstly, the surface should be closed. A closed surface has all of its vertices and edges shared by polygons. For example t-junctions may be present in the surface and they may be difficult to recognize

visually until the deformation starts and they form cracks. If the object fails the Euler's number test (vetrices+polygons-edges=2), one needs to verify if all the edges are being shared by exactly two polygons. On deformation, the continuity of a meshed surface does not change because of topological constraints, but it may lose smoothness. The second consideration is preservation of smoothness. Finally, self intersection should be avoided.

Methods like FEM (Hui and Leung 2002) and free form deformations (Sederberg and Parry 1986) have been applied to deform a surface mesh. However these methods do not update the topology. Subdivision techniques (section 4.4) can be deployed to obtain a feature based topology.

3.1.2 Parametric Surfaces

Parametric surfaces include spline patches and NURBs. Cobb (Cobb 1984) discusses in details about B-Spline based surfaces. Author even describes operations like twisting and bending. Most of current CAD systems have B-spline features that implement that work. Parametric surfaces use knots, control-points and high order recursive functions to evaluate parametric patches. Given the knots and control points, standard B-Spline equations can be used to generate surface points using fast recursive algorithms. Thus it is easy to tessellate and subdivide them for rendering and other purposes. Parametric surfaces offer flexibility in actual topology of the surface but the control points have to be in a regular grid.

Tools for trimming parametric surfaces or blending two surfaces are available but they result in additional overhead on computation and data management. Moreover, shapes acquired by trimming, culling and blending are difficult to modify because they tend to lose their inherent quality of smoothness and continuity. Auxiliary data structures,

remeshing whole surface with additional control points for fine editing and carefully meshed dynamic patches are some additional solutions to overcome the limitations but these methods are themselves costly and sometimes highly inefficient. For example, if we are interested in collision detection it is easy to calculate bounding boxes for these objects but in absence of associated rigid data structures like B-reps or BSP it is very difficult to back calculate the intersection points. Some other problems with parametric surfaces are that they lack the sense of interior or exterior, they are difficult to blend and Boolean operations are difficult to carry out. Software like Rhino use parametric representation extensively for modeling with a great efficacy.

A generalization of the concept of parametric surfaces is a free form surface. A regular grid of control points is used to manipulate the surface. Barr (Barr 1984) developed the original idea of twisting and bending solid primitives. This technique was extended by Sederberg and Parry (Sederberg and Parry 1986) to achieve free form deformations. They deformed quadratics and parametric surface patches by moving the control points that define the coefficients for Berstein polynomial. To control 3D shapes they use trivariate tensor product of Berstein polynomial. Free form deformation has been implemented by Welch and Witkins (Welch and Witkins 1992). Their method used directly manipulated Free-Form deformation developed by Hsu *et al.* (Hsu *et al.* 1992). Chai *et al.* (Chai *et al.* 1998) used free form exoskeleton.

Since freeform methods are indirect, user has to mentally predict the behavior. The topology of control points is rigid. Coquillart (Coquillart 1990) eased the topological problem by using lattice structures to deform whole or a set of control points. This

increased the flexibility of gaining deformations but it also made the process more indirect.

3.1.3 Implicit Surfaces

Implicit surfaces (Wyvill *et al.* 1986; Wyvill and Gascuel 1995) are also called soft objects. They are constructed by blending surfaces defined by implicit equations of the form F(x,y,z)=c onto skeletal primitives (points, edges, polygons etc.). In other words they are isosurfaces around a set of blended primitives. The functions can easily give the sense of being inside or outside (F(x,y,z)>c or F(x,y,z)<c). They can be manipulated by changing their skeletal elements or by changing the blending functions. Such surfaces have found broad applications in modeling organic objects, especially animation of soft objects. They are easier to define than parametric surfaces. Thus, they have the capability of achieving good morphing and animations by controlling few parameters. The down side of using implicit functions is that when many operations are combined the processing becomes complicated. The rendering of such functions requires calculation of surface points. Moreover they lack efficiency in defining arbitrary shapes (Ebert 1996, Frisken *et al.* 2000).

3.2 Volume Representation

The next common and more natural way for 3D modeling is to use volumes. In this section we will look at several ways to describe solid objects.

3.2.1 B-Reps

B-Reps is a complex data structure that defines an object by two types of information - topological and the geometrical. Topology is the connectivity and parent-child relationship between faces, edges and vertices. The exact shape and location of each

geometric entity (point, line, curve, polygon) that represents the topological entities comprises geometric information. There is no information about the interior volume. The edges may be straight lines or curves. Similarly the faces can be polygons or blended parametric surface patches. Whatever surface objects (polygons or patches) are used, they should be non-overlapping. In case of polygons, the polygon stores pointers to bounding edges which in turn point to defining vertices. On the other hand a parametric patch stores pointers to control points. The data structure is hierarchical in nature with a depth of 3. The levels are based on dimensionality of topological primitives. The lowest level contains vertices, above that the edges and above that the face. The more complex an object, higher is the number of nodes on each level.

3.2.2 Binary Space Partition (BSP)

In BSP representation, the solid is carved out by a finite set of planes such that material is kept on one side and removed from other side of the plane. Each side of plane is labeled as inside or outside the solid. Thus the space is recursively partitioned using single operation (inside or outside). The labels are arranged as a tree in which each leaf node has two child nodes. As the tree is recursed more, the regions converge to the actual shape of the object they represent and their contribution size diminishes.

3.2.3 Constructive Solid Geometry (CSG)

CSG models are built from Boolean and transformational operations on solid primitives (cubes, spheres, cylinders). Primitives are defined by their basic dimensions and locations. Data is stored as tree of operations among the primitives which form the leaf. Because there are very few primitives, the depth of tree may become large. The tree is traversed in depth-first manner. It is easy to manually handle CSG models than

corresponding B-Reps. The Boolean operations used to combine primitives are union, intersection and difference. The transformational operators are translate, rotate and scale. A leaf may be a primitive or a motional operator.

Naylor (Naylor, 1990) developed a sculpting system in which he uses CSG to carve volume between workpiece and tool. Mizuno *et al.* (Mizuno *et al.* 1998a) expressed their sculpting model by CSG solid objects based on surface primitives. Their CSG solid object is generated using closed curved surfaces.

3.2.4 Sweep

Sweeping is a method to describe a solid by an object moving in space. For example a plane moving along a curve defines a solid corresponding to the volume swept. A solid can also be generated using rotational sweep, but such solids will be axis symmetric. Manufacturing operations like extrusion, milling and turning involve sweeping a volume by punch or cutter. This makes this method of solid representation popular in manufacturing.

3.2.5 Discrete Cells

This is the most popular approach of solid representation in medical field. The solid is represented by a set of basic volumetric elements (bricks, tetrahedrons etc). The elements may bear different dimensions and orientations. They may be uniform and orthogonally oriented to each other (axis aligned as in case of Spatial Occupancy Enumeration). The easiest approach is to decompose the workspace into a three dimensional grid of uniform cubical elements. The elements that fall fully or partially inside the volume will be used to represent the shape. These elements are analogous to Pixels in 2D and are called Voxels (Volume Elements).

Like pixels, voxels may also have several associated attributes like color, density, tissue properties and temperature. If the attribute is density and we define a parametric function for density say, $\rho=f(x_i)$ where x_i is the i^{th} parameter (discrete or continuous) then $f(x_i)=0$ represents the exterior and $f(x_i)=1$ represents the interior of the object. $f(x_i)=k$ $(0<k<1)$ is an isosurface with k as the isovalue that represents the boundary of the material.

The method is extensively used in medical imaging to visualize interior structure of human body in 3d space by reconstructing 3D solid from scanned 2D data. Using attribute value stored in voxels, it is easy to handle transparency and 3D textures. Volumetric operations (needed for sculpting) can be done with great convenience.

The data structure allows reconstruction of data between sample points from local values for rendering or other processing. Moreover, arbitrary topologies can be constructed and accommodated.

A severe disadvantage of using discrete cells is that dimensionality causes data size to shoots up. For example, surfaces made up of up to 10,000 ($=100^2$) polygons are common. If their enclosed volume had to be decomposed into cells, it would have involved 1 million volumetric elements (100^3). To avoid memory overhead, resolution has to be compensated which causes aliasing problem during rendering. Conversion to smooth surface may required further processing and computationally demanding algorithms.

The voxelization of general curve and surface objects was first studied by Kaufman (Kaufman 1987) and more recently by Chen and Fang (Chen and Fang 1999). Lee and Requicha (Lee and Requicha 1982) proposed point classification method for

spatial enumeration to calculate volumes and other integral properties for solids. This work has been extended and studied in detail to invent new algorithms for voxelization of CSG objects. The methods however are not suitable for real-time volume sculpting because of their slow speed.

Galyean and Hughe (Galyean and Hughe 1991), Dewaele and Cani (Dewaele and Cani 2003), Hui and Leung (Hui and Leung 2002), Perng *et al.* (Perng *et al.* 2001) and Sensable Freeform™ (WWW_Sensable_B), all use discrete cells or voxels for volumetric representation.

The memory and processing requirements for voxel based data has been overcome by octree representation (Meagher 1980). This is hierarchical data structure in which the space is recursively partitioned into octants. Though this representation is not suitable for volumetric modifications, but for rendering and collision detection, it reduces the amount of processing needed.

3.2.6 Subdivision Solids

Subdivision solids are basically 3D polyhedra called cells. Polygon, edge and vertex information is used to carry out subdivision operations derived from generalization of tri-cubic B-spline solids. They fall into a category between surface and volume representation schemes because though they represent volume by splitting it into cells, the volume information for each cell is not available.

McDonnell and Qin (McDonnell and Qin 2001) used FEM to deform subdivision solids. In another work McDonnell *et. al.* (McDonnell *et. al.* 2001) applied spring mass system to deform subdivided solids.

Subdivision surfaces are very convenient and general for creating models in which only the boundary is important. The user can express complex topologies with great flexibility. However, the manipulation requires user interaction to define and move vertices during different levels of subdivision. There is no suitable method for data exchange available for subdivision solids and they do not represent the volume they occupy.

3.2.7 Particle Based Modeling

This method is popular for representing natural objects and phenomena like trees, sand, sea waves and clouds. The object is represented as a set of numerous small geometric primitives. These particles are generated and removed in a stochastic manner. The benefit is that simple rules governing the particles may translate to complex behavior of the entire system. Thus it needs fewer parameters to control the system behavior. Effects like elasticity, energy conservation, thermodynamics etc have been successfully applied on these systems. The great drawback is that a huge memory may be required to store the properties and states of all particles.

3.3 Display Schemes

Surfaces can be easily displayed through conventional polygonal rendering. However, volumetric data can be displayed using special methods which have been described in following subsections.

3.3.1 Visibility Ordering

This method is especially used to display BSP structures. The method involves arranging the primitives in a list so that the occluded primitives appear before the occluding primitives in the list. The list would change dynamically with change in view

point. The sorted primitives are then drawn using ray casing for each pixel. The main problem with this method is that primitives have to be arranged and that takes $O(n^2)$ of computation. Since the ordering data can easily be extracted from BSP tree, this method is commonly used to display BSP trees. Computation required for each frame may become excessively large for big BSP trees.

3.3.2 Slicing

This method is used mainly in medical imaging where the intersection of the solid object with a particular plane is to be displayed. The voxels that intersect the plane are displayed.

3.3.3 Ray Casting

Discrete volumetric data and CSG data can be conveniently rendered through ray casting technique. In this method, rays are projected from the eye location to each pixel on the screen. The pixel is directly updated based on the integrated effect of the part of material the ray passes through. Main advantage is that pixels need to be updated locally near the tool. However display hardware is not currently optimized for such type of rendering and in case if the whole model changes, the display performance drops down significantly. Wang and Kaufman (Wang and Kaufman 1995) successfully used ray casting with good rendering rates. Dewaele and Cani (Dewaele and Cani 2003) used ray casting to display their sculpted models.

3.3.4 Isosurface Extraction

The most popular method of generating a surface mesh from a volumetric data is Marching cubes. The objective of algorithm is to determine the surface from a given isovalued function, a scenario that arises especially while dealing with voxmaps.

Marching cube uses divide and conquer approach for locating the surface (Lorensen and Cline 1987). The whole volume is split up into cubes (voxels). Given a cube, any of its eight corners may lie inside or outside the isosurface. Lorensen and Cline (Lorensen and Cline 1987) identified 256 possible configurations and generalized them into 14 unique non-trivial configurations. The generalization was possible because of the symmetric structure of cube. Finally based on what edges the isosurface intersects, triangles were formed. This was done for all the voxels covering whole of the volume and thus constructing the surface that represented the isovalue.

The algorithm is fast and the resulting surface is of high quality. Galyean and Hughe (Galyean and Hughe 1991) used incremental marching cubes to update the marching cubes locally. Adaptive marching cubes (Shu *et. al.* 1995) extends the technique to hierarchical data structures like octrees (Baerentz 1998).

To display sculpted models, Ferley *et al.* (Ferley *et al.* 2000) and Perng, *et. al.* (Perng, *et. al.* 2001) used this algorithm for surface generation. Cabral *et al.* (Cabral *et al.* 1994) used 3D textures with marching cubes for volume rendering, but 3D textures demand large memory that usually require high end graphics cards.

Gibson (Gibson 1998) suggested using distance maps in contrast to field generated by density function on voxels. The distance map value for a voxel is its signed distance from the closest point on isosurface. The advantages of distance maps are that they smoothly cover the whole space and their gradient can be easily calculated. Inside or outside testing becomes trivial. Information like nearest point on surface and it's direction is easier to obtain Other benefits are that the reconstructed surfaces are smoother, the derivatives of distance maps indicate surface normals and presence of

sharp features can be easily detected and appropriate filters can be applied to smoothen without blurs on edges.

3.4 Modification Schemes

The schemes discussed in this section pertain to manipulation of volumetric data.

Galyean and Hughes (Galyean and Hughes 1991) achieved material removal or addition by combining tool and the object voxmaps. The highest resolution grid size that they used was 30*30*30. To avoid aliasing affect on tool-object interaction, they sampled tool 4 times higher than the object. The combining operation involved min and max functions. Material removal process was achieved by assigning the minimum of object's original and corresponding tool voxmap values. Heat gun effect was achieved by applying the maximum among 0 and the difference between object and tool voxmap. Similarly, building tool used minimum of 1 and the sum of object and tool voxmap values. Sandpaper was applied by assigning each voxel the weighted value of its own and the six neighboring voxels. Density parameter has also been used by Perng *et. al.* (Perng *et. al.* 2001) to sculpt material.

Dewaele and Cani (Dewaele and Cani 2003) used diffused deformation field technique. They divided the whole volume into cubic cells with each cell having some density value. The deformation field worked in a manner that the amount of material lost near the tool was added in the non-intersecting area. This was precisely done by balancing the mass flux between neighboring cells. Pseudo distance between constraints, tools and clay determined the deformation magnitude. Surface tension effects were achieved by avoiding cells with very low material densities.

Ferley *et al.* (Ferley *et al.* 2000) used tools defined by the isovalue of a scalable potential field that existed within the bounding box of the tool. The interference of tool's field and object's potential field defined the tool action. Two types of tools were available, ellipsoidal tools and freeform tools. The freeform tools could be generated by defining a discrete potential field around the surface obtained by their software. They carried out material removal in a smoother way by subtracting tool's potential contribution from the potential of the Corners. The less smooth way was to directly delete all Corners where tool had nonzero potential contribution. Similarly, painting could be done in smoother or non smooth way. Low pass filtering method to smoothen the surface by smoothing the potential filed was also implemented. Stamping operation was achieved using local deformations.

An innovative work in solid simulation was done by Gourret *et al.* (Gourret *et al.* 1989). They simulated grasping task using FEM for deformation calculation of both hand and object. They determined frictional and repulsive forces between colliding objects based on amount of overlap.

Hui and Leung (Hui and Leung 2002) used FEM on boundary elements (polygons) to achieve deformations. For addition and removal of material they maintain voxmap data. Their methodology is explained in details in section 4.3.

McDonnell *et al.* (McDonnell *et al.* 2001) employed time dependent equations to evaluate bulk deformations. They used a user-defined control lattice generated on subdivided solids. The lattice was equipped with normal and angular springs on the edges and faces, while the lattice points were assigned mass properties. To give stability to the spring mass structure they also supplied cross springs along the diagonals of faces which

provided necessary shear forces. The spring statics was governed as usual by Hooke's law whereas the dynamics of each spring was governed by time dependent Lagrangian equation of motion. Initially the user specified number of subdivisions of the control lattice. Once subdivided, they assigned physical properties such as mass, damping, and stiffness. The Lagrangian equation of motion was augmented with PDE for spring (energy based Lagrangian equation of motion)

Mizuno *et al.* (Mizuno *et al.* 1998a) performed sculpting by mapping chisel on the display screen. A list of intersecting points along each viewing line on the screen was maintained. The intersection points were the points that intersected viewing line and the surface of material. The location and number of these points could change according to the change in cross-section along the viewing line when tool was applied for addition or removal of material. The points corresponding to each viewing line were arranged in order of their distance from the viewpoint. After each operation, the list was updated and the object was redrawn.

3.5 Discussion

During the discussion of surface based schemes we see that surfaces are the easiest entities to display and they enjoy a tremendous hardware and software accelerations developed over years (display listing, shading and texturing for example). Whether we have a discrete definition of surface (individual polygons, edges and vertices) or in the form of a continuous function (implicit surfaces), there is no information about the volume enclosed by the surface. It is, however, difficult to store volume related data. (say, density if one is interested in mass conservation or the distribution of elasticity or thermal properties over the volume).

Another problem is that surface and volume characteristics don't go hand in hand. It is very easy to get two surfaces with same area but very different enclosed volumes. Mathematically, a finite surface ensures a finite volume but a finite volume does not ensure a finite surface which means a lot of volume loss or gain may be experienced during surface based manipulations. For example, filling clay in the pit on the surface of the blob (figure 3.1) to make it a complete sphere will increase volume and at the same time, decrease surface area.

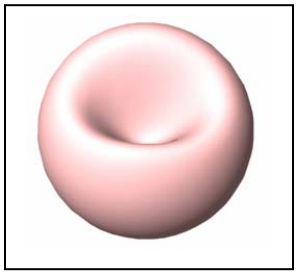


Figure 3.1: A deformed sphere

From the discussion of volumetric schemes we find that once any model is represented in cellular form, it becomes easier to perform different sculpting tasks. The only but major drawback is severe restriction on resolution. Being limited in resolution, the sculpted objects are not realistic and impressive. Table 3.1 shows different volume representations evaluated (weighted) according to accuracy and ease of performing memory management, display and sculpting. Among the modification schemes described in section 3.4, it is easier to see that apart from FEM or spring mass schemes none of the schemes efficiently mimic clay properties. However both the schemes, FEM and spring-mass, require large linear matrix systems to be solved.

	Memory	**Display**	Accuracy	**Physical Sculpting**
Mesh	2	3	3	1
Parametric	1	3	3	1
Implicit	1	3	3	1
B-Rep	2	3	3	1
BSP	2	3	3	2
CSG	2	1	1	3
Cell Decomposition	3	1	1	3
Particle	3	1	1	3

Table 3.1: Comparison of solid representation methods

The object representation and modification schemes used in current work will be discussed in next chapter.

Chapter 4: Solution Approach

In section 2.1 goals of the current work were defined as:

1. Enabling user to sculpt 3D CAD models that exhibit physical properties of clay during modeling process
2. Natural clay-tool interaction
3. Realistic influence of environmental constraints on clay
4. Realistic appearance
5. Interoperability with other CAD systems

The first three requirements could be satisfied by a physics based method like spring-mass method, finite difference method (FDM) or finite element method (FEM). There was no doubt that all of these methods required volumetric data. Applying these methods only on surfaces might give a realistic effect of constraints by constraining specific areas on surface but it could severely fail to serve first two requirements. We planned to apply these methods on volume instead. Assuming that volumetric data was made available, we needed to find most suitable scheme among the three to deform the volume. All three methods required setting up physical structure inside the clay volume, suitable for that particular method. Finite difference method needed a regular grid of node points inside the volume. Finite element required tetrahedral or brick elements to be defined inside the material. Spring-mass system needed the clay volume to be populated with mass points connected to each other with springs (generally cubic with or without diagonal springs). It was felt that clay is more plastic than elastic, therefore mass-spring being predominantly elastic was not suitable for calculation of clay dynamics. Both FEM and FDM methods were tested on sample scenarios. The same brick elements used for

FEM were represented by points at the center of mass for FDM approach. It was found that though FEM took more time for solution than FDM, the accuracy of solution was better. Constraint handling through flux balancing gave rise to iterative solution schemes in FDM. FEM, on the other hand, resulted in a set of linear equations which could be solved in direct or iterative manner. Since we had already implemented Frontal solution technique (Section 4.3.2) and applied it on some models we decided to continue with FEM. Moreover, we had an option of trying iterative solution schemes on finite element matrices too.

Irrespective of what method we used, it was clear they were not suitable for real time virtual clay simulation because enormous calculations of $O(n^6)$, where n is the number of linear divisions, was involved. (Volume and number of nodes is of $O(n^3)$ and squaring that is the order of matrix size) These methods were very generic and could handle environmental constraints easily. In order to have a broad idea of how the material would behave under given forces and constraints, we decided to use FEM algorithm at low resolution levels. At low resolution, the entire clay volume would be represented by bigger and fewer discrete elements. At high resolution, the same volume would be defined by discrete elements that were smaller in size and more in number. It was supposed that once some deformation values were obtained at strategic locations inside the volume of the material, those values could be interpolated to get the deformation on the clay surface. The basic idea was to generate a vector field from low resolution grid of a model which could be used to evaluate deflection values on high resolution version of the same model.

The scheme just mentioned assumed that we had high resolution data available too. Thus it became clear that we would need to maintain a high resolution description of clay model. Two questions still needed to be answered. Firstly, how would we maintain high resolution volume data which might become astronomical in size? For example, a simple 3D grid with 100 divisions on each edge contains one million elements. If the divisions were to be increased from 100 to 1000 in order to accommodate complex features on a particular model, the spatial data size would increase from one million to one billion! Secondly, how would we display such data to achieve our goal of rendering realistic model?

The second question was easy to answer. We already had high resolution data and its surface could be easily extracted and rendered after some antialiasing. It was noticed that we did not have to use density field (section 3.2.5) with high resolution data, as it is done with voxels. Since the tool-clay interaction happened on surface, it was sufficient to carry out deformation on surface only (volume flow automatically follows that) which meant that evaluation of deflection field value deep inside the volume was unnecessary. Finally we found that we did not have to store any attributes such as color, density, temperature etc., with the elements of high resolution data. This allowed us to explore data compression techniques (section 4.2) to store the bulky high resolution data. Now an appropriate low pass filtering method was required for antialiasing. We got inspiration from schemes used for B-spline based sub-division surfaces to derive a suitable low pass filter mask in an analogous manner.

In this chapter we will first go over the finalized solution scheme and then explore each step in details.

4.1 Overview

On assembling each phase of solution approach discussed above, we get an overall picture of the new virtual sculpting model as shown in (figure 4.1). The flowchart in figure 4.1 has been broken down into two subcomponents shown in figure 4.2. After understanding the individual subcomponents, the reader will not find it difficult to understand the flowchart in figure 4.2.

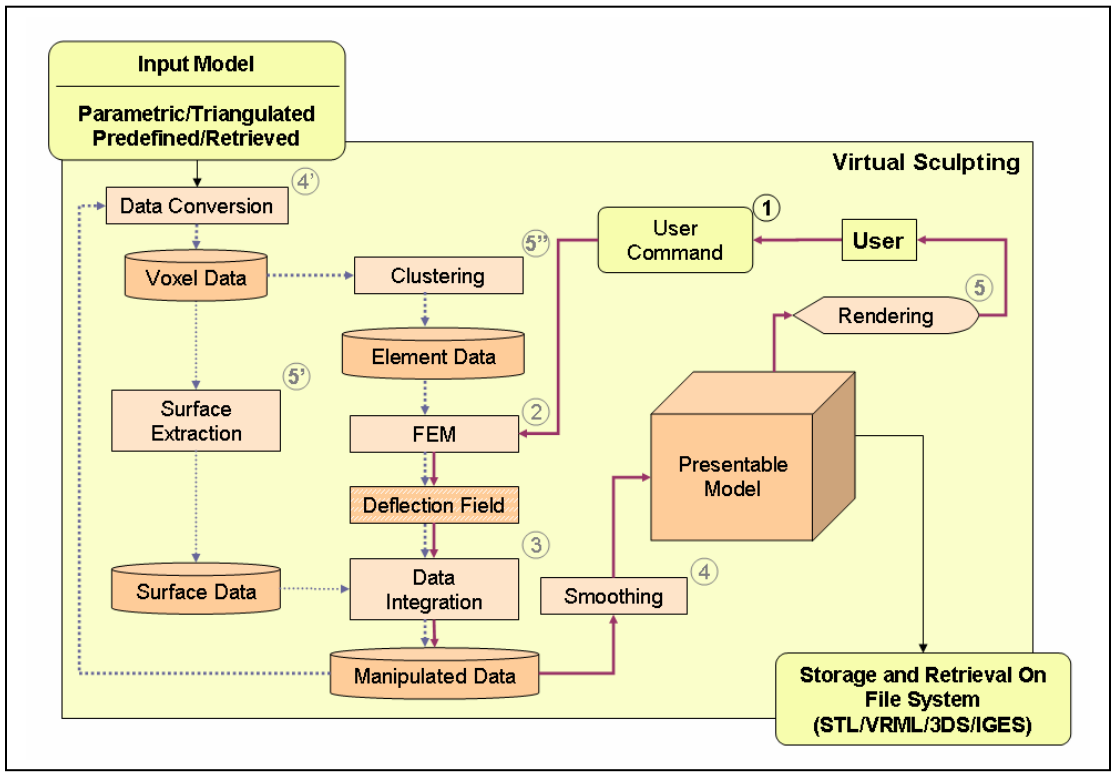


Figure 4.1: The Virtual Sculpting System
(Steps like 5, 5' and 5'' should be interpreted as *simultaneous steps* running in parallel.)

The main components of the flowchart are the data and process entities. The types of data structures and processes that appear can be isolated and looked independently (figure 4.2). They are described as follows.

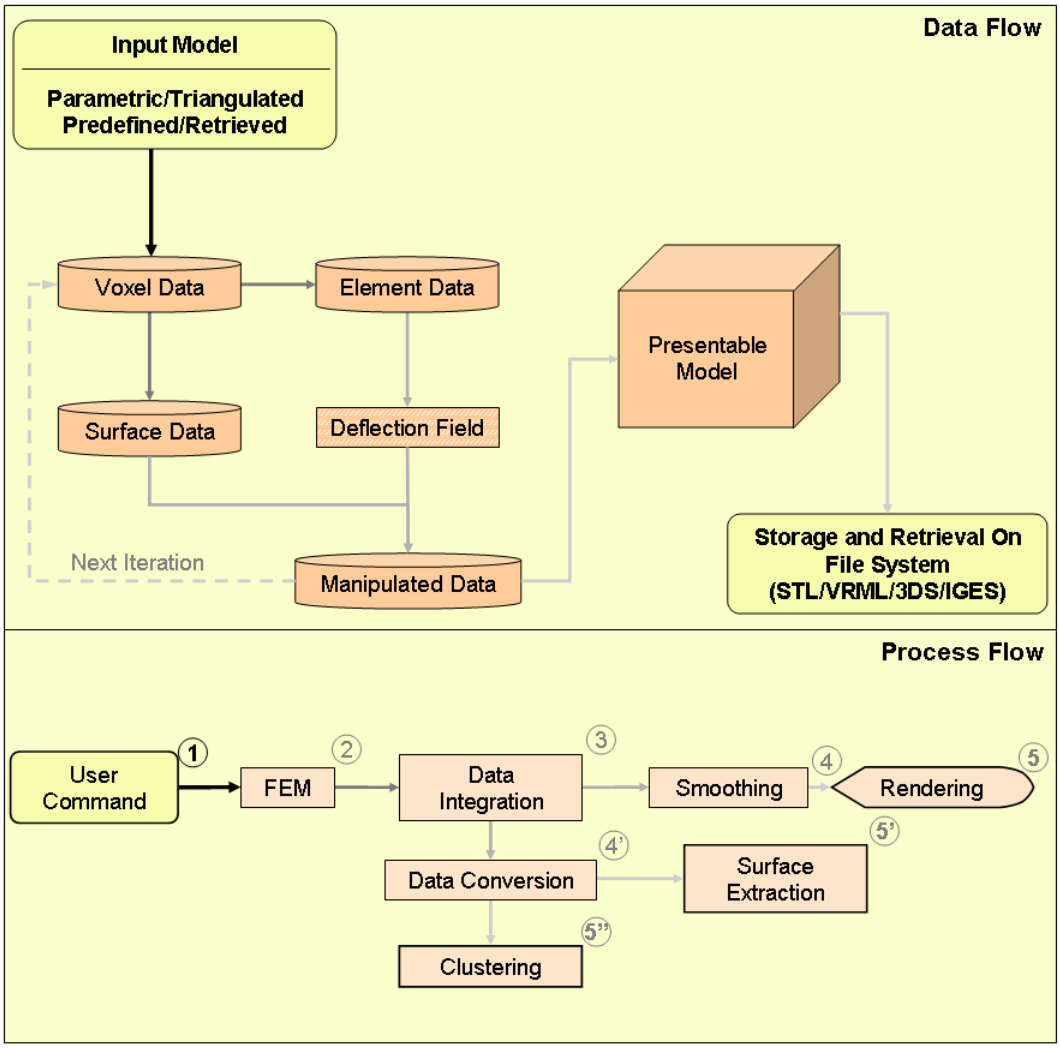


Figure 4.2: Data Flow and Process Flow in Virtual Sculpting System

The creation of data structures in the sculpting system starts from an input model (figure 4.2, Data Flow). The model should be a closed volume, preferably a triangulated surface model. The model may be inbuilt primitive or may be loaded from a file. This conventional CAD data is converted to compressed volumetric data called *Voxel Data* (we did not use this data as voxels but we called its elements *compressed voxels* to differentiate them from *Finite Element Data*). High resolution Voxel Data were used to define FEM elements stored in structure called *Element Data*. The same Voxel Data was

used to generate a data structure that contained surface definition called *Surface Data*. Another data structure called *Deflection Field* was generated when FEM dynamics was applied on Element Data and the result was interpolated for Surface Data. In the actual implementation, the Deflection Field resided with surface data within the Surface Data. The combination of Deflection Field and Surface Data resulted in *Manipulated Surface Data* or simply *Manipulated Data*. After antialiasing, the Manipulated Data could be rendered to the user or saved in a CAD file.

The FEM processing starts when user does some manipulation (figure 4.2, Process Flow). The result of FEM calculation is integrated with Surface Data during *Data Integration* step. The integrated data (Modified Data) is *smoothed* and *rendered*. At the same time it is converted to Voxel Data (*Data Conversion*) which is further used to *extract surface* and to perform *clustering* to define FEM elements.

Following sections are dedicated to the three main components of the Virtual Clay System, viz., data compression, FEM, and generalized subdivision based low pass filter.

4.2 Data Compression

To store the voluminous three dimensional grid of voxels, Boundary Cell Encoding (BCE) (Wan *et. al.* 1999) has been used. It is in essence Run Length Encoding (RLE) (Freeman 1974). RLE is a common compression technique used in image compression (e.g. jpeg images). The method replaces runs of a data value by one instance of the repeated value and the number of times the value repeats. A volumetric data composed of uniform elements clustered together must have many runs. An analysis shows that original data of $O(n^3)$ reduces in size to $O(kn^2)$ where k is the average number of runs in the run length direction and n is the number of linear discretizations. This

complexity is comparable to Octree representation (Montani and Scopigno 1990). The solid object can be looked as columns of voxels oriented in Z direction arranged in a grid on XY plane.

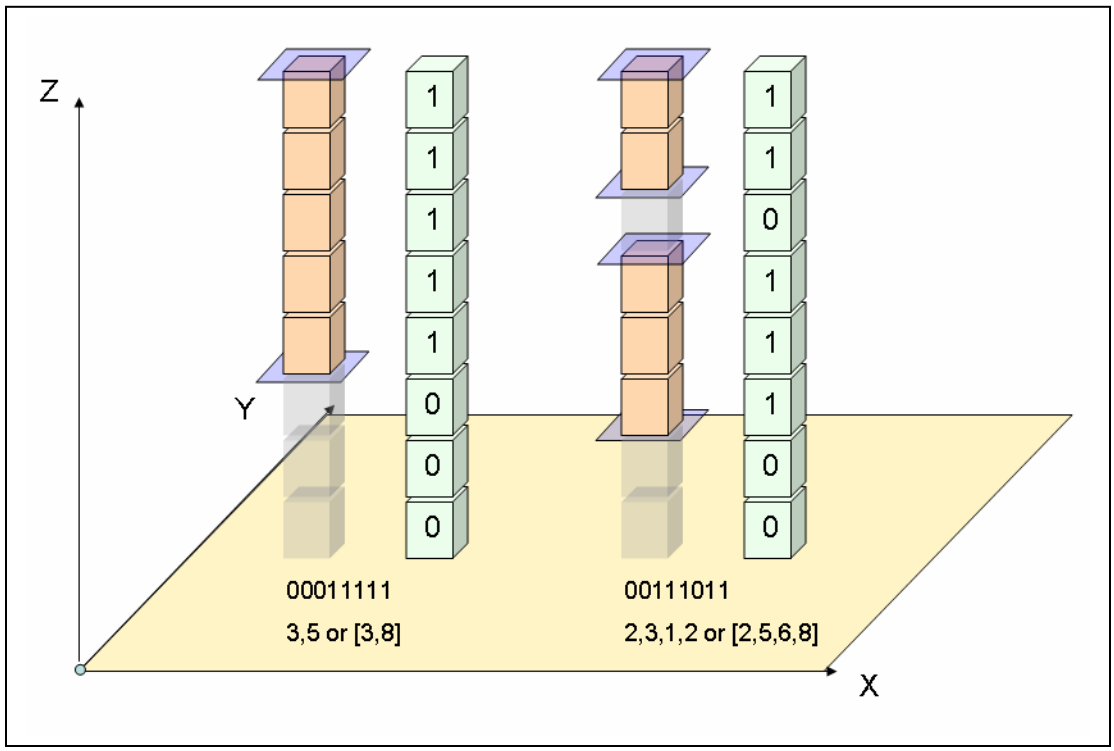


Figure 4.3: Boundary Cell Encoding

It is quite evident that *k* has a value of 2 for a convex object. The average value of *k* for regular concave solids should be somewhere between 2 to 4. Special cases (ladders) may have higher values of *k*.

Our BCE notation is slightly different from RLE. Figure 4.3 shows RLE and our BCE notation for two columns. For first column, the voxel status (starting from bottom) is 00011111. This means three empty regions are followed by five voxels. We can write it in RLE format as 3,5. In terms of our BCE notation, we may express the absolute location of boundaries (blue patches). The same boundary set 3,5 will then be written as

[3,8] in BCE notation. A Boundary Cell Column or simply column is defined by all pairs of boundaries. This is shown for second column, whose column structure is [2,5,6,8]. A pair of subsequent boundaries that enclose material is called *Run*. The second column contains two runs. Thus the number of runs in a column is half the number of boundaries. The benefit of using absolute position of boundaries is that they can have negative values. This representation should satisfy two requirements at any time. Firstly, the boundaries should always be in pairs. Secondly, every successive boundary should be strictly greater than the predecessor. The second requirement is very important for querying purpose.

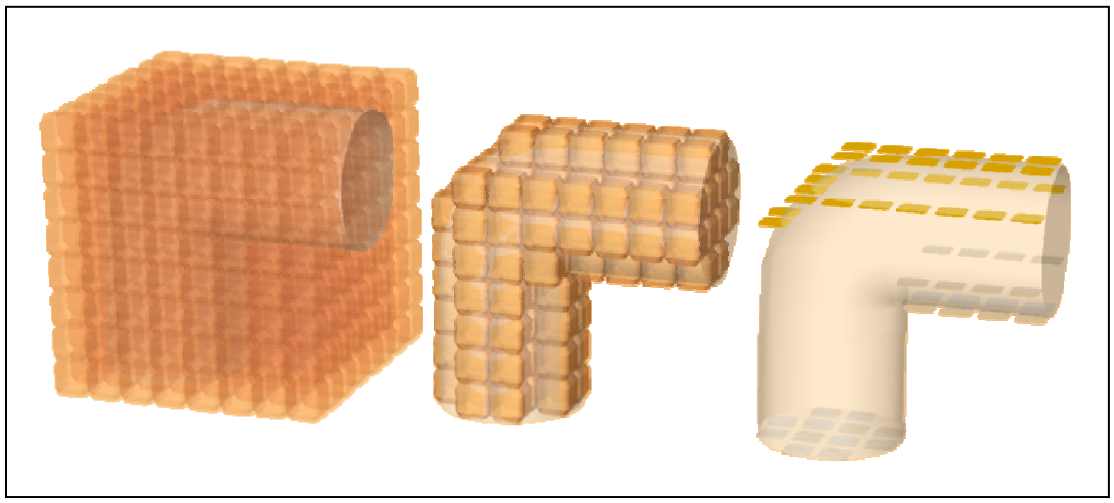


Figure 4.4 Boundary cell encoding of a pipe

Figure 4.4 shows the application of BCE on a pipe. The details of cellular operations on columns will be discussed in section 6.4.1.

4.3 Finite Element Method

Realistic simulations for volume deformation have been created by Terzopoulos in (Terzopoulos and Fleischer 1988 a&b) and (Terzopoulos *et al.* 1987). They applied linear and later nonlinear elastic model on curves, surfaces and solids. The equations

were solved using FDM. Hirota *et al.* (Hirota *et al.* 1999) achieved volume conservation during deformation of free-form objects using discrete level-of-detail representations. Du and Qin (Du and Qin 2003) solved elasticity equations implicitly using FDM to design, blend and reconstruct 3D shapes. All the methods just described were either not real-time or they involved several minutes of precomputation time. Gibson (Gibson 1997) proposed 3D ChainMail algorithm - a volume discretization using links chained together in 3D. An elastic relaxation method was employed to minimize energy function. This method was real-time but the way physical properties of material were handled was not physically appealing.

4.3.1 FEM and Solid Modeling

Bro-Nielsen and Cotin (Bro-Nielsen and Cotin 1996), Bro-Nielsen (Bro-Nielsen 1995) and Kunii (Kunii 1994) carried out FEM analysis on biological tissues to calculate deformation (such as skin and fat). These applications were specific to medical field. Hui and Leung (Hui and Leung 2002) efficiently used the boundary element FEM technique to deform surface. Their work used both surface and volumetric representation. For basic deformation they performed FEM calculations on boundary elements (surface mesh). Thus whenever the object was modified, they did not need to regenerate a solid mesh. Apart from basic deformations, if user wanted to carry out volumetric operations like engraving, gluing etc, they generated corresponding volumetric data using a simple approach of identifying the voxels lying inside the surface of the object. Based on tool size they carried out further convolution on the voxels.

They avoided generating a mesh of solid elements from the volume data. Alternatively they used boundary elements in which the displacement and forces were

evaluated directly on a mesh of polygons approximating the surface of the object. They argued that "since a triangular mesh of a volume model can be easily obtained using the Marching Cube method, the boundary element approach can be directly applied to the iso-surface of a volume model". They assumed that there were no boundary forces but only traction forces. Thus they evaluated the integration of weak form of Navier's equation of material flow for a boundary element which gave stiffness matrix to be assembled for all elements.

Our work has been highly inspired by their work. For example we have also used a combination of surface and volumetric approach. We have also used the same Navier's equation and FEM approach to solve our system. However, we use it directly on solid representation rather than boundary elements because of the reasons discussed in section 3.5.

Mandal *et. al.* (Mandal *et. al.* 1999) used FEM formulation on modified butterfly and Catmull-Clark subdivision based surface meshes. Later McDonnell and Qin (McDonnell and Qin 2001) used FEM on subdivided solids. They associated trilinear eight node hexahedral elements with each subdivided element. Gaussian quadrature was used to evaluate mass, stiffness and damping distributions and conjugate gradient method was used to solve the resulting set of linear equations after assembly. However this method was restricted to CAD models generated from solid subdivisions.

Debunne *et al.* (Debunne *et al.* 2001) used non-nested multi-resolution tetrahedral elements to create visco-elastic deformation objects. They used explicit FEM to solve each element. It was not clear what resolution of discretization they took and how stable their method was for complex objects with arbitrary topologies. Wu *et al.* (Wu *et al.*

2001) used non-linear FEM (involving quadratic strain) with mass lumping and explicit time integration on progressive triangular and tetrahedral meshes. This method compromised the physical behavior by manipulating the elemental stiffness matrix. Though they chalked out whole procedure for volumetric objects, they implemented their work on surface meshes only.

In our implementation of FEM, we update geometry incrementally by superimposing deformations on the geometry during each iteration (updated Lagrangian).

4.3.2 Formulation

FEM subdivides a solid or its boundary into small elements to solve PDE governing the material behavior. Physically, it imposes a set of forces such as internal stress and external pressure, and then calculates equilibrium of shape among the volume elements. This approach is very realistic because it incorporates physics beginning from the element level.

Behavior of an isotropic elastic material can be described in general terms by Navier's equation,

$$\Delta\tilde{u} + \frac{\nabla(\nabla \bullet \tilde{u})}{1-2\nu} + \frac{\tilde{F}}{\mu} = 0 \quad \text{where, } \mu = \frac{E}{2(1+\nu)} \tag{4.1}$$

Navier's equation for material flow is a PDE of the same order as Laplace's equation. Conversion of this equation to weak form and then to Galerkin form reduces it to a set of coupled linear equations (See Appendix – 2),

$$\mathrm{K}^{\mathrm{e}}\, d_o = f_o \tag{4.2}$$

for a single element which can be assembled over all elements to give,

$$\mathbf{K}\, d = f \tag{4.3}$$

(**K** is stiffness matrix, $\boldsymbol{K} = \sum K^e$, d is deflection vector and f is load vector)

The stiffness matrix, **K**, is sparse and quite often symmetric. Although K is sparse and may not be symmetric, K^e is a dense symmetric matrix. Therefore it is desirable to avoid solving a single large sparse matrix which results after assembly. According to linear algebra rules, an unassembled row cannot be added to or subtracted from other rows but a fully assembled row can be added or subtracted from other rows which may or may not have yet been assembled. This allows elimination of a node whose all possible elemental contributions have been accounted for. This trick to eliminate nodes as soon as they are fully assembled was the theme of frontal solution program for finite element analysis (Irons 1970). Irons technique has now been proved to be best suited direct solution of general sparse matrices. It is currently being used in finite element packages like ANSYS.

The frontal technique relies on maintaining a frontal matrix that consists of finished and unfinished nodes. This is solved during the assembly process itself. The frontal matrix is factorized and solved over the fully assembled rows. This keeps the matrix dense and small. Because of the high density of the matrix, any direct solution method (Crout's decomposition, Gauss elimination etc.) will perform equally well.

Current work employs single front solution using Gauss elimination (Appendix 4, Schur Compliment) to solve compressible (Appendix 2) and incompressible (Appendix 3) elastic materials. Maple codes for compressible elastic and incompressible elastic element stiffness matrix calculation are provided in Appendix 5.

4.3.3 Constraint Handling

The program deals only with fixed points whose displacement it zero. Since the displacement of these nodes is known to be zero, their contribution is skipped during assembly process.

4.3.4 Dealing with Irregular shapes

The 3D grid has some definite number of elements in each direction. If any single element is removed, its contribution to overall stiffness matrix will vanish. All this information is extracted from the *elementized* CAD model.

A set of brick elements represents a valid solid if all sets of *well* connected elements are *well* connected in one or more possible ways. Any two elements are well connected if they share a common face. Sharing a corner or an edge is not sufficient enough to be well connected.

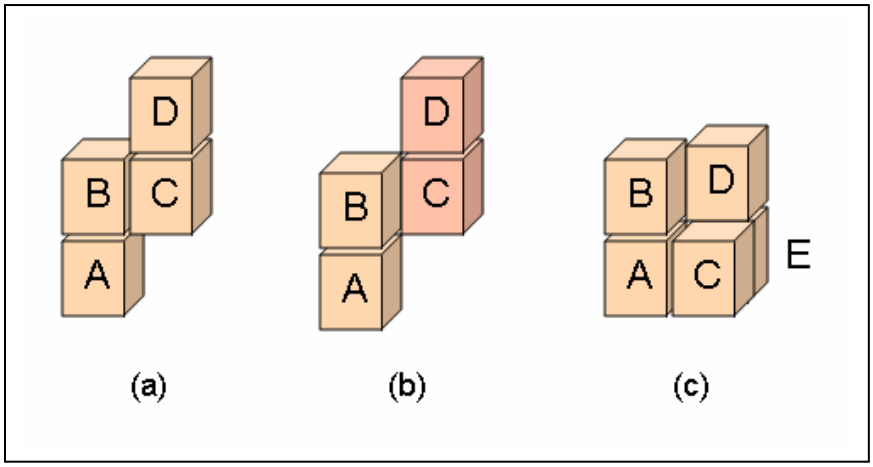


Figure 4.5: Connectivity (a) Valid, (b) Invalid, (c) Valid

In figure 4.5 (a), AB, BC and CD are well connected. In (b) AB and CD are well connected but the two sets are not directly connected via BC junction. This makes the solid invalid because it has dangling pieces. In (c) though set B and D are not connected

directly, elements B and D are well connected via route BA→AC→CE→ED, hence the system is valid. Section 6.3.3 describes an algorithm to determine well connectedness.

4.3.5 Assembly

An explicit 24x24 master matrix K for elastic (32x32 matrix for incompressible plastic) material had been derived for a brick element (Appendix 5).

$$K_{24x24}\, d_{24x1} = F_{24x1} \tag{4.4}$$

This was defined initially in the program. During assembly, the values were referred to from this matrix. If we look at it in more details, we can treat small 3x3 matrices as single entity. Then K can be considered as K_{8x8}. Now, K_{ij} element in the matrix represents the following dependency between force components (Fx, Fy, Fz) at node 'i' and displacements (u, v, w) at node 'j'.

$$K_{ij\ 3x3}\, u_{j\ 3x1} = F_{i\ 3x1} \tag{4.5}$$

Hence, $K_{ij\ 3x3}$ was the smallest block to be added to the matrix front at any time.

4.3.6 Solution

As described earlier, Frontal method running Gauss elimination was used. Solving a matrix of size NxN requires $O(N^3)$ of computation time. If *n* sub-matrices of size *x* each are to be solved, computation is of the order, $O(x^3 * n) = O(x^2 * xn) = O(x^2N)$

Now, $O(x^2N) << O(N^3)$

Thus frontal method could save a lot of computation time. Front size *x* could be minimized by optimizing element numbering. Prefront routines could be used to determine the maximum front size. The order in which elements were visited was the key to keep the front small. Therefore element numbering played a crucial role rather than node indexing, because which node would be completed first depended entirely on

element order. BLAS kernels could be used for dense operations to gain a good efficiency. Compact data storage techniques reduced memory requirements. Several of them have been discussed in Duff *et. al.* (Duff *et. al.* 1989). This project also exploited some of them.

4.3.7 Indexing

Though indexing of elements was important, but it was the life span of the node that influenced the front size. Therefore a node that was shared by minimum number of elements was to be preferred. But this could not be the only criterion because this would cause all the surface elements to be assembled first. The nodes that were shared by other internal elements would remain in a 'not fully summed' condition adding to the frontal size. This problem could be resolved by assigning a distance to each element from the initial node. At first level, the elements at the shorter distance would be preferred to any other set of elements. Next, the elements at same distance would compete on the basis of the first criterion. Shortest distance between the starting element and any other element could be determined in a number of ways. This distance was not the spatial distance, but the elemental distance. For example, shortest distance between two diametrically opposite elements on a torus is not the diameter but some other distance along the circumference of the torus that passes through minimum number of elements. Dijkstra algorithm (most commonly used in networking) could be used to determine the shortest tour between any two elements.

$$\text{Cost} = w1*\text{Distance} - w2*\text{Nodes added} \tag{4.6}$$

Alternatively,

$$\text{Cost} = w1*\text{ Nodes eliminated} - w2*\text{Nodes added} \tag{4.7}$$

Duff and Reid (Duff and Reid 1983) proposed minimum degree ordering of nodes in order to keep the front size minimum along with the number of boundary elements. In simpler terms, given a structure, the elements would be scanned along the smallest cross-section. The scheme is depicted in Figure 4.6.

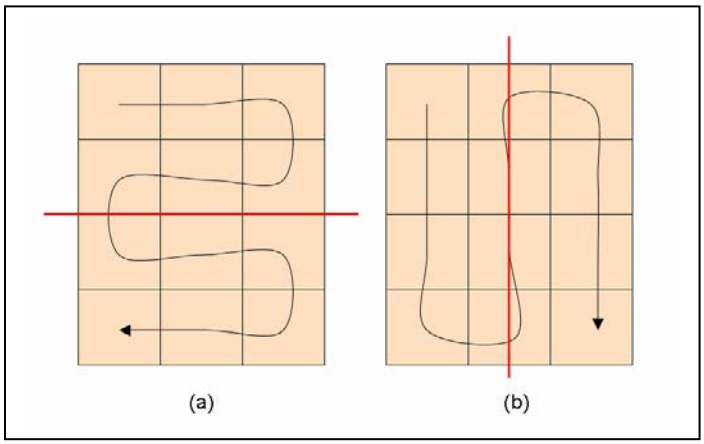


Figure 4.6: Assembly along (a) smaller cross-sections is desirable than (b) longer cross-sections

For this preliminary work on FEM application our method to index FEM elements was inspired from Patra *et al.* (Patra *et al.* 2003). Patra *et al.* proposed data structures pertaining to Space Filling Curve based indexing. This approach divides the domain into recursive subcubes. In our simple implementation of SFC, we arranged all recursive subcube levels in a hierarchical form and proceed in a depth first manner. SFC is actually the limit of scheme presented in figure 4.6 on recursive subdomains.

4.4 Generalized Subdivision Surfaces

4.4.1 Low Pass Filter

Surface Data extracted from Voxel Data had two characteristics. Firstly, it is composed of polygons that were squares. Secondly, all those polygons were either parallel or perpendicular to each other. Thus the surface was highly aliased. The polygons had been obtained from high resolution data. Their number was sufficient enough for

rendering purpose. The only requirement was to smoothen the surface without changing the topology (anyhow, topology changed automatically during every iteration because the generating Voxel Data changed). This meant antialiasing the surface points or in other words applying a low pass filter (LPF). A low pass filter is a mathematical operator that allows variations with frequencies lower than a predetermined frequency to pass and attenuates variations with higher frequencies. It can be said that the energy function obtained by integrating high frequency variations gets minimized.

In Surface Data, high frequency variation was the difference between adjacent corners and low frequency variation corresponded to the overall shape of the object. Thus a low pass filter would allow broad features of the clay object to exist and it would reduce the magnitude of inter-voxel variations. Applying a LPF would also remove highly localized features like spikes or cracks. We did not expect these features to exist in clay model. Moreover, it was less likely that these features would be captured by the discrete data.

The LPF should satisfy following two requirements:

1. *Preservation of shape and size*
2. *Compactness*

The filter will be applied on all surface points. It should be as compact as possible because a little overhead would be amplified multiple times. The drawback of using LPF was that there was data loss if multiple smoothing iterations were applied. For example, the volume of the object might reduce or sharp edges and corners would also get smoothed. In practice, it is easier to get sharp edges on clay by cutting rather than deformation. Cutting operations using sharp and flat tools might be applied easily to

overcome this problem. Shrinking of size caused by LPF was negligible. LPF was applied on the objects before rendering and exporting. This step was, however, skipped before rediscretization.

On the contrary, a high pass filter (HPF) attenuates frequency below a threshold. Applying a high pass filter (HPF) would make all the high frequency features in the clay model more prominent. This might be used to isolate and preserve sharp features discussed above.

4.4.2 Subdivision Surfaces

Subdivision surface is a surface generated by dividing an initial polygon mesh based on some division rules. The rules are mainly derived from the subdivision equations of parametric B-spline curves and surfaces. Hence the resulting surface is more refined and tends to conform to a smooth surface which is the convergence limit. The rules are in the form of LPF masks which express new vertices in terms of weighed average of old vertices. The mask may be based on a triangular or a quadratic topology. It may vary within the mesh (nonuniform subdivision) or it may vary from one iteration to other (nonstationary subdivision). Frequently, nonuniform subdivision is required to handle edges and corners separately. Apart from that nonuniform mask may be needed to handle extraordinary vertices. Extraordinary vertices are those vertices which are not found in regular meshes. For example in triangular meshes, regular interior vertices share 6 edges and regular boundary vertices share 4 edges. The number of edges shared by a vertex is called *valence*. In quadratic meshes, regular interior vertices have valence 4 and regular boundary vertices have valence 6. Vertices that share more or less number of edges than regular vertices are extraordinary.

Subdivision surface are more flexible in resolution and topology than parametric surface. A number of subdivision schemes have evolved. Most relevant among them are briefly discussed as follows:

1. *Butterfly Scheme*

   This scheme was developed by Dyn *et al.* (Dyn *et al.* 1990). It is a triangular mesh based scheme that uses a general averaging mask instead of mask derived from B-Spline functions. The mask is shown in figure 4.7. The new edge point P is the weighed average of points A through H with the weights indicated in the figure.

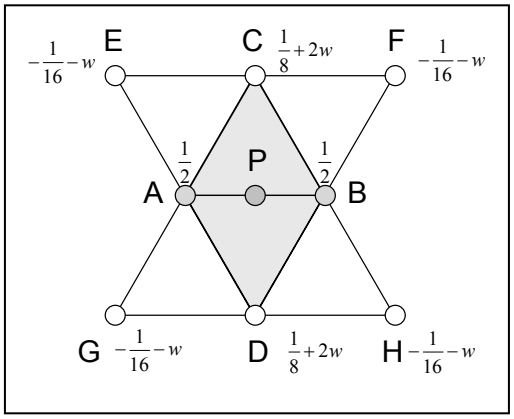


Figure 4.7: Butterfly Scheme

   Here w is tension parameter which controls the influence of neighboring points.

2. *Loop Scheme*

   Loop scheme is based on uniform quadratic box splines. The scheme is triangular in which a triangle is subdivided by generating new vertex and edge points. Different masks are used for vertex, edge points and extraordinary points. For brevity these masks are not provided and can be found in (Loop 1987).

3. *Doo-Sabin Subdivision*

   This scheme was introduced in (Doo 1978). It subdivides quadrilaterals based on subdivision of biquadratic B-spline surfaces. On the boundaries, however, a mask based on quadratic B-spline is used.

4. *Catmull-Clark Subdivision*

   Catmull-Clark subdivision (Catmull and Clark 1978) is similar to Doo-Sabian subdivision as it subdivides quadrilaterals but the order of B-spline used is one higher than that of Doo-Sabian subdivision.

Among all the methods mentioned above, Butterfly and Doo-Sabian schemes satisfy C^1 continuity in a regular patch whereas Loop and Catmull-Clark schemes produce C^2 continuity. On extraordinary vertices, however, the continuities are C^0 for Butterfly and Doo-Sabian schemes and C^1 for Loop and Catmull-Clark schemes.

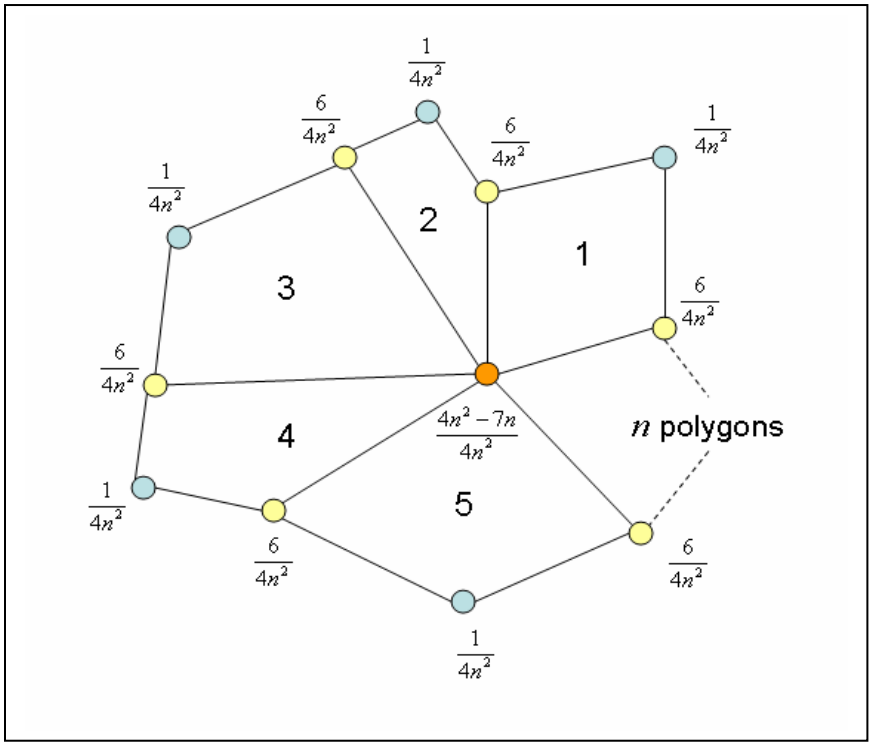


Figure 4.8: LPF mask used for smoothing

Current work used Catmull-Clark subdivision scheme to calculate LPF masks. The choice was made because of two reasons. Firstly this method was based on quadrilateral mesh and our Surface Data consists of only quadrilaterals. Secondly, the curve guaranteed C^2 continuity on regular points. The LPF mask is shown in figure 4.8. Detailed derivation of this Catmull-Clark based LPF mask is provided in section 6.4.3.

4.5 Hardware and Software Used

The software was developed in KDevelop using g++ compiler. The GUI was created on OpenGL using GLUT libraries. Red Hat Enterprise Linux WS 4.0 was used on a machine with Pentium-IV 2.7 GHz processor and 512 MB RAM.

Next two chapters describe the data structures and algorithms used for implementation of data flow and process flow in the software.

Chapter 5: Data Structures

This chapter describes different data structures used to input, store and process data associated with the virtual clay model. We will begin with a brief description of a generic class that has been used to represent several data types.

5.1 Overview of Intensive Data Component

During data handling, the data was represented in as much compact form as possible. The essence of our data handling lies in the derivation of a new class template called SortedVector from vector template of C++ standard template library (STL).

A C++ vector object is instantiated as,

```
vector <ClassName> a;
```

A SortedVector object was instantiated as,

```
SortedVector <ClassName, ComparatorClass> a;
```

where ComparatorClass was defined as,

```
class CompareClass {
public:
        int operator()(const ClassName & __x, const ClassName & __y)
const {
                return (__x<__y)?1:0; // Or equivalent code
        }
};
```

SortedVector relied on powerful index() function. This function is a querying function that takes time of $O(\log_2 n)$. It works in a similar way as find() function of C++ Maps but it does not have key-value pair. If the query item is found, the location (index)

of the item and an affirmative(1) flag are returned. If the query is not found, a location and a negative(0) flag are returned. The location in this case is the index where the query item would fit keeping the array sorted. This is a feature that is not available in *sets* (C++ STL).

Several memory intensive data objects were of the type SortedVector. For example, voxels, FEM elements, constraints and intersections of colliding objects were all compressed objects of type SortedVector.

5.2 Stereolithography Files

STL or Stereolithography files are specifically suited for layered manufacturing used in rapid prototyping. STL files contain the description of solid approximated as a closed surface made up of triangular facets. A triangular facet consists of three vertices, and a unit normal (figure 5.1).

```
solid   Sphere
  facet normal -0.13 -0.13 -0.98
    outer loop
      vertex 1.50000 1.50000 0.00000
      vertex 1.50000 1.11177 0.05111
      vertex 1.11177 1.50000 0.05111
    endloop
  endfacet
  ........
  ........
  ........
endsolid
```

Figure 5.1: STL File Format

The vertices should be ordered in counterclockwise direction while looking from outside. The normal should point outward. Each edge should be shared by exactly two

triangles. The units are not required but all the vertex coordinates should be positive. It is preferable to sort the triangles in increasing order of their z coordinate.

In our implementation, the data from STL file was directly stored in an array and was passed on for generation of volume data. The output STL file was generated from the final triangulated surface after applying single iteration of smoothing on deformed data.

5.3 Volume Data

Our approach used both volumetric and surface models of the clay object. The volumetric model was directly manipulated. Surface model served three main purposes. Firstly it could be directly used for rendering, secondly it could be exported to STL files and finally, it could be rediscretized to generate new Voxel Data.

The volumetric model consisted of encoded values of boundary elements. If we consider a dataset of pure voxels, they represent workspace as a uniform 3D grid. The 3D grid can be considered as a 2D grid where each grid point on the XY plane has a column (or a number of columns) of material. This is illustrated in figure 5.2.

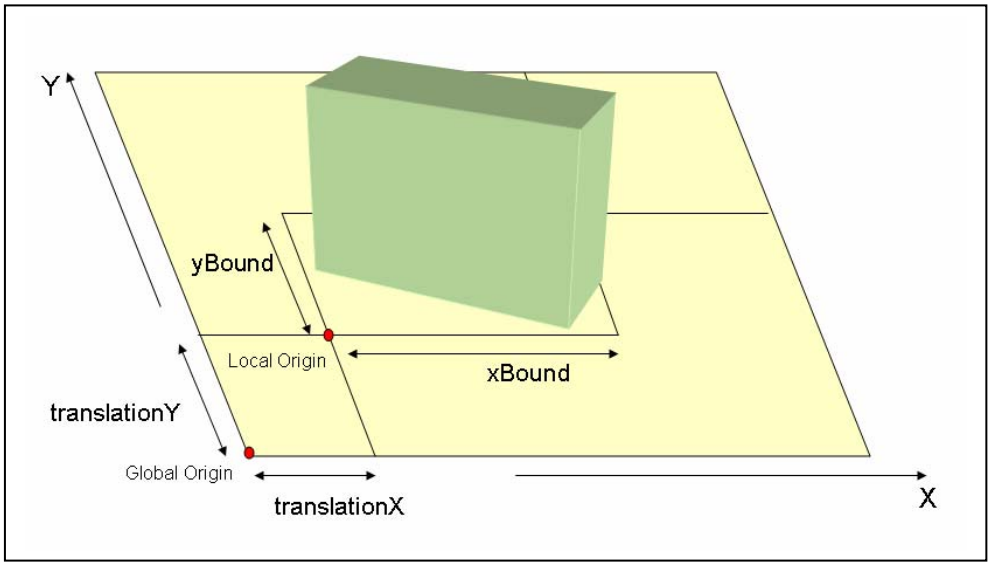


Figure 5.2: Volumetric Data

The locations where the columns started and the lengths of columns were stored in a single one dimensional array. This array (column) represents the volumetric fingerprint of the particular grid point on XY plane. Combining all such column data on the whole XY grid accurately defined the Voxel Data saving huge memory costs.

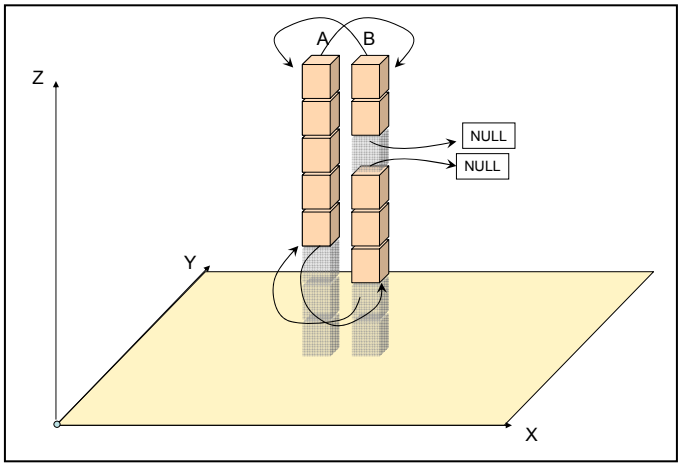


Figure 5.3: Pointers for fast surface traversal

Additionally pointers to neighboring boundaries (in X and Y directions) could be stored along with the boundaries as illustrated in figure 5.3. This enabled fast traverse on the surface during operations that needed surface scanning.

The 3D object shown in figure 5.2 has a XY grid of size xBound by yBound. The reference point is located at point with coordinates translationX and translationY. The abscissa and ordinate, both are integral with a step size of unitLength. Hence xBound, yBound, translationX and translationY are all integers. The world coordinates (x, y) of any gridpoint (X, Y) can be expressed uniquely as,

$$(x, y) = ((X + translationX) * unitLength, (Y + translationY) * unitLength) \quad (5.1)$$

$0 \le X < xBound\ and$,
$0 \le Y < yBound$

There were a few issues with this data structure especially in the virtual clay context. One might need to know the boundary between two adjacent columns. The union and intersection of columns were of special interest when two pieces of clay collided. Fast algorithms were required to extract the information needed from the compressed data because there was a potential for huge number of calls for these basic operations.

The access speed was obtained by keeping the boundary data sorted. Which meant that the boundaries were stored in a particular order, say the order in which they appeared when one moved in positive z direction starting at the bottom of the material. Once sorted, the data retrieval would involve O(log(n)) comparisons in the worst case, the average being much smaller. We use the SortedVector template class derived from STL vector class. The data passed on to this class was guaranteed to be sorted without repetition. These two conditions mimicked the actual physical conditions that the data represented. Figure 5.4 shows *voxelized* representation of two solid objects.

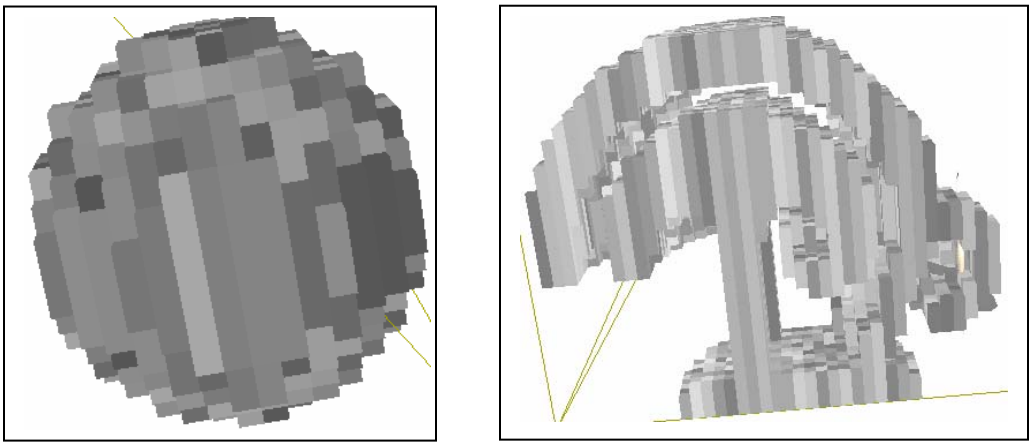


Figure 5.4: Solid models represented using VoxelData

A possibility also exists that the data structure defined a noncontiguous physical model. A separate algorithm was designed to deal with such inconsistencies in data. The

columns above a particular XY gridpoint might also overlap due to ill-defined data. Another type of error that could happen was the presence of dangling voxel clusters. These were clusters of voxels that were linked to each other through a point or an edge rather than a face. Those clusters, though connected through edges and points, were pooled with completely non-contiguous clusters and treated alike.

Class	**Data/Function**	**Description**
VoxelData *Base Class to store and manipulate volumetric data.*	`population`	Number of Voxels
	`translation[2]`	Position of Local Origin in terms of Global coordinates
	`xBound` `yBound`	Number of discrete steps in X and Y direction
	`unitLength`	Size of step
	`gridPoint`	2D array of columns (see Class *Span*)
	`sparse()`	Construct FEM Elements
	`force()`	Calculate force using Tool Intersection
	`rank()`	Identify noncontiguous volumes within the data set
	`collision()`	Check collision with another Voxel Data
	`unshared()`	Given column A and column B, find A-(A∩B) and B-(A∩B)
	`shared()`	Given column A and column B, find (A∩B) and (A∪B)
	`reconstruct()`	Eliminate any discrepancies and error in data.
Span	`z`	*SortedVector* of integers to hold boundary encodings.
SortedVector *Derivative of STL Class vector. The data at any time is sorted. Repetition is not allowed.*	`index()`	Queries data. Since the vector is sorted, time taken is $O(\log_2(n))$. Returns index and flag. If flag is 0, the query key not found and the index is the position where the key can be inserted keeping the array sorted. If the flag is 1, the key was found and the index is the position of the key in the array.

Table 5.1: Classes for volume data storage*

* The list is not comprehensive. Only important data and functionality listed.

Table 5.1 lists structure of classes used to represent volumetric data in the C++ code.

5.4 Surface Data

The approach for storing surface data has been inspired by Ferley *et al.* (Ferley *et al.* 2000). Ferley *et al.* used a concise method to store information about volumetric elements which they called *Cubes*. That method was extended in our work to store surface data.

Ferley *et al.* called the nodes of their elements *Corners*. Corners were defined as regularly spaced points in 3D space. Their Corners stored a potential field value, a color and some cached data, such as the field gradient, and the point location (to avoid its recomputation from the virtual grid indices and sampling steps). In addition to that they also possess a key to compare them and organize in the form of balanced binary search tree called CornersTree. The associated key was simply made up of indices (i j, k) of the Corner in the virtual grid implicitly defined by the regular space sampling. The potential field carried a value between arbitrary minVal and maxVale (in their case 0 and 3). If the value of a Corner dropped below minVal, it was removed from the CornersTree and if it exceeded the maxValue, the field value was clamped to maxVal. For all absent Corners, the potential carried a constant value of minVal. They called the volumetric elements Cubes. A Cube carried pointers to the eight nodes. For a Cube to 'exist' at least one of the pointers should not be null. Each cube kept a key that was the key of the one of the eight points of the Cube that had the minimum coordinates. Using Cubes key, a CubesTree was generated in a manner similar to CornersTree. Cube also stored an index value which was based on the intersection configuration of the cube with the isosurface. The exact index

was determined from a precomputed index table for different intersection configurations. Apart from that, twelve pointers to the 12 edges were also there. A list of Cubes that intersected the iso-surface called crossList was maintained separately. Edges were stored in another balanced binary tree. Edges were used to compute intersection of the Cube with iso-surface.

To avoid reexamination of each cube for iso-surface intersection after each sculpting process they used timestamp mechanism or a temporary tree to store only dirtied Cubes. Once the new intersection index value and the respected edges were determined (similar to Marching Cube method) the edge tree was updated. Surface normal was calculated using the potential gradient on each curve obtained by interpolating the scalar attribute on the end Cornes or the edge at the exact intersection point on the edge.

In our case we have voxels corresponding to Cubes and we have SurfaceData that is a set of Corners. The Surface Data essentially consists of quads that form the surface of voxels forming the outer layer. The data structure is similar to Voxel Data. The main difference between Surface Data and Voxel Data is that instead of boundary element values, the Surface Data stores each and every surface point that appears on a particular XY gridpoint. Since the X and Y coordinates are same as the location of the XY gridpoint, only z value needs to be stored for each surface point. We call it *zbase*. The actual coordinates x,y,z of the point may change after undergoing deflection and smoothening, but the zbase remains unchanged. The complete key of a point is (X,Y,zbase) and this represents a unique location in 3D space. To avoid the need to store polygons (because of topology was compatible with the surface of a Voxel Data), only

flags were stored which indicate existence of polygons adjacent to a surface point. It was found that only 3 flags were required to represent the whole topological interactions. There flags corresponded to xy, yz and zx planes. So if flag xy was 1 for a particular point (X,Y,zbase), the point formed a quad that contained point (X+1,Y+1,zbase) as the diagonally opposite corner. Similarly one could extend the definition for yz and zx flags. The value zero of a particular flag meant absence of corresponding polygon. Subdivision LPF required fast traversal within Surface Data. To ease this each surface point was also supplied 6 pointers that pointed to immediate next surface points in positive and negative principal directions. These pointers were NX, NY, NZ, PX, PY and PZ. Here NX and PX stand for negative and positive X-directions respectively. Let P be the pointer to current point. Then P→PZ→xy refers to a polygon parallel to and above P→xy polygon, whereas P→NX→NY→xy refers to another polygon in the same plane as P→xy polygon. This is illustrated in figure 5.5.

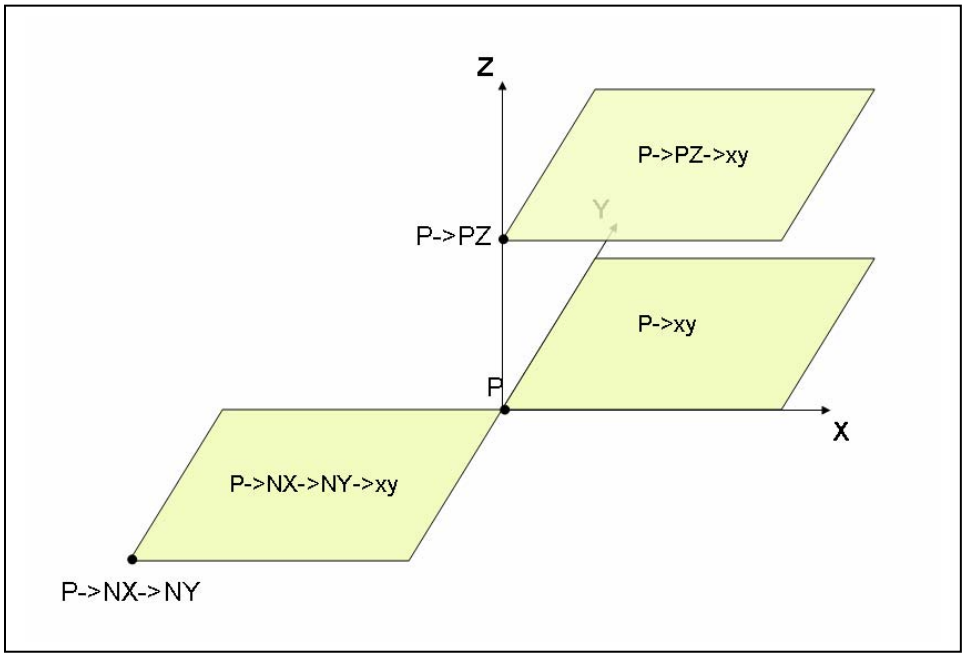


Figure 5.5: Surface Data Structure

Table 5.2 lists structure of classes used to represent surface data in the C++ code.

Class	Data/Function	Description
Surface	`population`	Number of Voxels
	`translation[2]`	Position of Local Origin in terms of Global coordinates
	`xBound` `yBound`	Number of discrete steps in X and Y direction
	`unitLength`	Size of step
	`gridPoint`	2D array of Surface Data columns (see Class *SurfaceDataColumn*)
	`smoothen()`	Construct FEM Elements
SurfaceDataColumn	`z`	*SortedVector* of S*urfaceData* to hold boundary encodings.
SurfaceData	`zbase`	Integer to store z coordinate
	`xy,yz,zx`	Flags that indicate if a quad is available on xy, yz or zx planes along the positive coordinates.

	`defX,defY,defZ,`
	`tempDefX,tempDefY,tempDefZ,`
	`dX,dY,dZ`
		Variables to store FEM deflections and surface smoothening data

	`nx,ny,nz`	Variable to store vertex normals
	`NX, PX, NY, PY, NZ, PZ`	Pointers to vertices in 6 Cartesian directions

Table 5.2: Classes for surface data storage

* The list is not comprehensive. Only important data and functionality listed.

5.5 FEM Data Handling

The Element Data that was passed to FEM assembler actually was an instance of VoxelData class with a resolution lower than Voxel Data. Because the Element Data was discretized, it had an integral bounding box. A complete 3D grid was created within this

bounding box with same resolution as the Element Data by the assembler. The classes used to store, solve and process FEM data are listen in table 5.3.

Class	**Data/Function**	**Description**

FemSolver	`deflections[]`
	`forces[]`
	`xmin, ymin, zmin,`
	`xmax, ymax, zmax,`
	`numNodesX, numNodesY,`
	`numNodesZ`
	`nodeID[]`
	`FS`
		Different data types to store deflections at nodes, information about vacant or constrained nodes, forces etc.
		FS is the frontal solver to which all nodal and elemental information is passed

	`getDeflection()`	Once solved, it returns the deflection at a particular node.
	`solve()`	This function with appropriate parameters is called to begin the solution.
FrontalSolver	`solve()`	Information about elements, nodes, forces and constraints is passed and deflections are returned.
FemInterpolate		Class to interpolate FEM deflection from the FEM element data to the Voxel Data.
Plane		A 3d Plane that constrains the Voxel Data.(clay object)
Voxel_Plane		This class is used to determine constrained cross section in the clay object.

Table 5.3: Classes for handling FEM data

* The list is not comprehensive. Only important data and functionality listed.

5.6 Other Classes

Some other classes were used to do auxiliary jobs like CAD file import/export, conversion of one type of data into another, handling graphics display and time keeping. These are listed in table 5.4.

Class	**Description**
Parser	Read/Write Stl Files
Stl	Stl Data Storage Class
Stl2Voxel	Discretize STL Object
Voxel2Surface	Extract Surface Points and polygons from Voxel Data.
Surface2Stl	Convert smoothed surface back to stl for renderting/export
Timer	Time keeping for different parts of the code.
GLCore	Open GL Class
Main	Main Container (Contains All Objects)

Table 5.4: Other classes

Chapter 6: Algorithms

This chapter describes various algorithms used in data conversion, data processing, FEM, rendering and collision detection. Processes like data conversion that involved different data types described in previous chapter were required to be called once during every manipulative move. Similarly collision detection, force calculations and FEM analysis were required to be calculated once in manipulative move carried out by user. We call these Container Algorithms because they encapsulated several algorithms to carry out specific tasks. Section 6.4 deals with data processing algorithms. Data processing algorithms were the underlying Core Algorithms. These algorithms operated on single bits of data and were called numerous times in each manipulative move depending on the size of data. The Core Algorithms needed very high optimization because the speed of Container Algorithms changed tremendously with them. For example a single scalar by vector multiplication could affect the performance of frontal solution by 0 to 30 milliseconds which could change the display frame rate by 15 %! Data compression saved substantial time within Core Algorithms causing substantial improvement in Container Algorithms.

6.1 Data Conversion

6.1.1 STL to VoxelData

STL files are triangulated. A polygon wise progressive dimensional discretization method was used to convert triangulated surface data into discrete volume. Since our objective was to set up column like *vertical* structures on a *horizontal* XY grid, each triangle was orthogonally projected on XY plane. The triangle edge covering the whole

length along the X axis was scanned and discretized. At each discrete step along X axis the projection was discretized along Y direction too. Thus the whole region in the interior of projection on XY plane was discretized as shown in figure 6.1. The active grid points lie in shaded projection. Z location on the triangle plane was calculated for each *XY grid point* (shown by light spheres). These Z locations were rounded to nearest integer and pushed into appropriate XY grid point in volume data object.

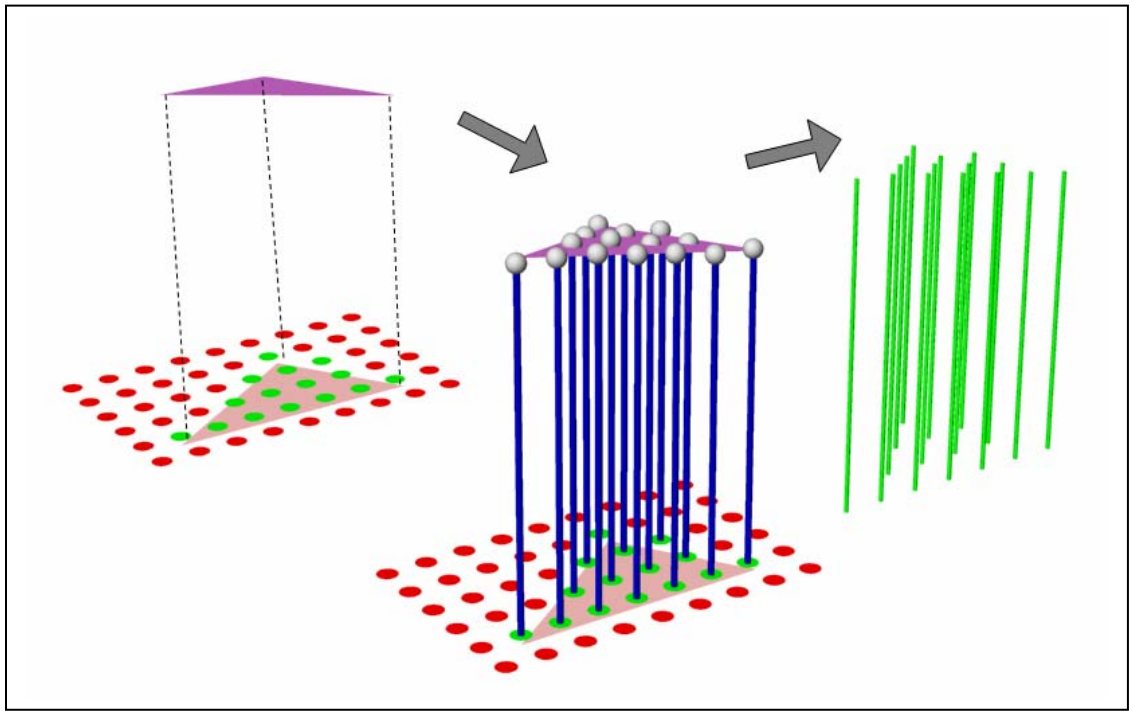


Figure 6.1: Generation of volumetric data

If the object was closed surface, it was expected that we would always get even number of points of intersection with any infinite line passing in 3D space because for every in-flux there would be an out flux. Since entire column lines were along Z direction, it was possible that a particular line could intersect at an edge shared by two polygons on extremities (line A in figure 6.2). This was special case of a line being tangential to an edge. Indeterminist situation might arise if the triangle itself were parallel

to Z direction (line tangential to a surface). To overcome these problems, each STL object was rotated and translated infinitesimally (0.012345% of 360^{o} for rotation, 0.012345% of step size for translation) before carrying out discretization..

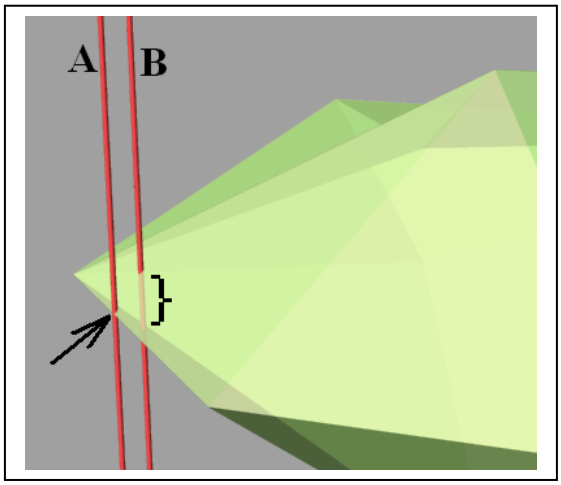


Figure 6.2: Line intersecting an edge

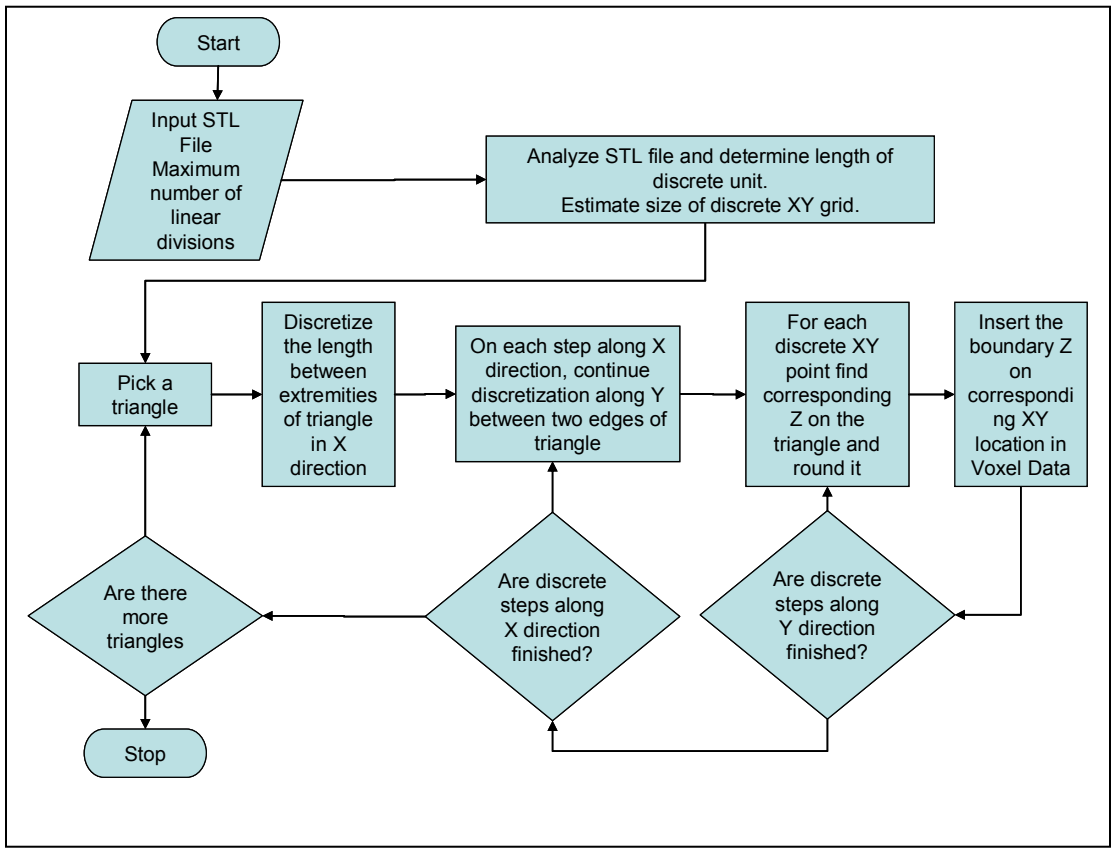


Figure 6.3: Discretization Algorithm

Figure 6.3 shows the complete flowchart of discretization process. The method is quite fast and it requires each triangle to be visited only once. The computation required for each polygon was proportional to the area (or projection of area on XY plane). Thus the total amount of calculation was not more than the surface area of the object i.e. $O(n^2)$. Moreover once calculated, same plane equation could be used for all XY grid points lying under a particular triangle.

6.1.2 VoxelData to FEM Data

The FEM elements were of same structure type as VoxelData, but they were of low resolution. VoxelData data structure contained one byte of information about resolution. Low resolution elements could be generated by either rediscretizing the input STL object with higher resolution or using the high resolution data structure. Though rediscretization was faster $O(n^2)$, we chose to count voxels present in high resolution Voxel Data ($O(n^3)$) because it provided smaller errors because of the higher dimensionality. Based on the capability of FEM solution method, only a smaller limited number of elements were generated. Equating the volume occupied by voxelized object to the volume occupied by FEM elements, the size of FEM elements could be estimated as:

$$U_l = ceil\left[\left(\frac{V_{VD}}{N_{FEM}}\right)^{\frac{1}{3}} u_l\right] \tag{6.1}$$

Here U_l is the unit length (resolution) for the FEM elements, V_{VD} is the total volume of clay object which is equal to number of voxels, N_{FEM} is the number of elements supported by FEM algorithms, u_l is the unit length of Voxel Data. The function *ceil* is ceiling function, that is, rounding *up* to the nearest integer. Bins (candidate

elements) of edge length U_l were constructed in 3D space and the number of voxels lying inside them were counted. If voxels covered 50% or more space inside a bin, those bins were accepted as elements (figure 6.4).

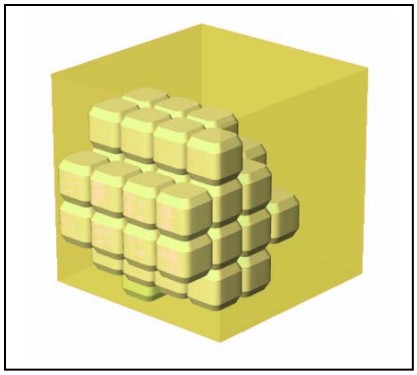


Figure 6.4 : Element Construction

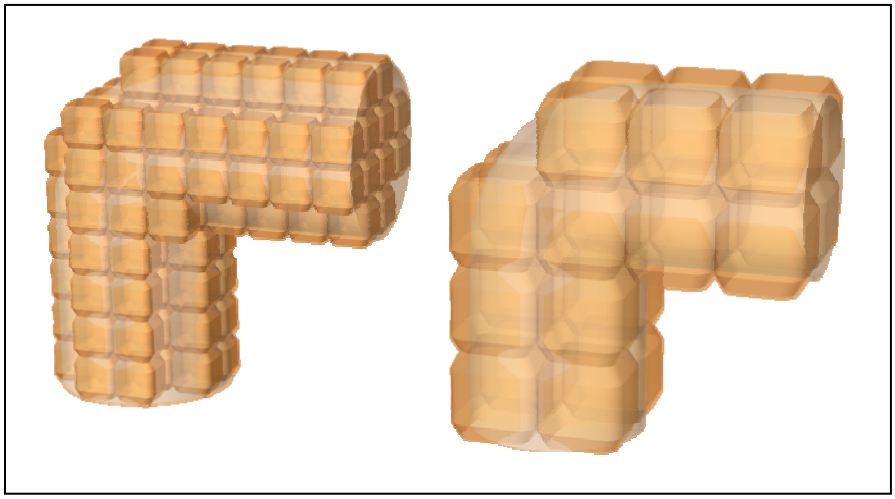


Figure 6.5: Small voxels are used to define relatively bigger FEM elements

The elemental structure thus generated was compressed using BCE method described in section 4.2.

This approach proved to be better than octree representation because for octree we have,

$$U_l = u_l * \exp\left(\log_e 2 \; * ceil\left[\log_2 \left(\frac{V_{VD}}{N_{FEM}} \right)^{\frac{1}{3}} \right] \right) \tag{6.2}$$

This expression has larger amount of rounding and hence more error. In fact U_l obtained from Octree based setup is too conservative.

6.1.3 VoxelData to SurfaceData

Surface Data was generated by a very simple scheme. The voxels exposed to free space were determined and their exposed surface was used to define *square* polygons of Surface Data. The top and bottom surfaces (normal to Z direction) of each column were directly available from the encoded values. The surfaces with normals along X and Y direction were difficult to determine. If some part of the *right* surface of a column did not overlap with *left* surface of the column on the *right*, we had a surface that was exposed and its normal points towards *right*. Similarly the *right* column would decide its exposed surface with normal pointing towards *left*. This is conveyed in figure 6.6.

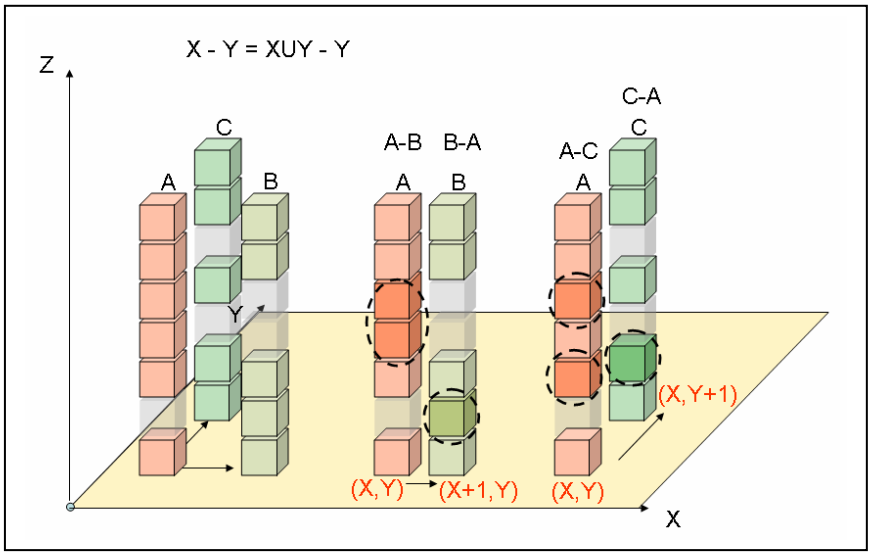


Figure 6.6: Exposed surfaces located between pairs AB. and AC.

(Note that B is towards right of A and C is behind A)

Columns A and B were tested with each other for non-overlap. This resulted in the unshared exposed polygons of A with outward normals pointing towards positive X direction and similarly unshared polygons of B with outward normals towards negative X direction. Voxels with non-overlapping surfaces are circled in figure 6.6. Non-overlap voxels were directly computed in their compressed form using Core Algorithm for operation A-B and B-A (see section 6.4.1).

6.1.4 SurfaceData to STL

SurfaceData consisted of quads (particularly, squares) positioned and oriented in 3D along the VoxelData grid. The vertices of these quads were enriched with information about their 3D deformation vector, normal direction and smoothing deflection. Deformation vector was interpolated from FEM results. Applying FEM deformation displacement on the original coordinates of the vertices gave their modified location which was used for smoothing filter. The surface was triangulated by splitting each quad into two triangles. Thus the number of polygons exactly doubled and the number of vertices remained the same. Thus it could be claimed that the STL object inherited its topology from the Surface Data and in turn Voxel Data. The new coordinates of the vertices were calculated by combining original location and FEM deformations for the calculation of new Voxel Data. Additionally the smoothing deflection was applied for rendering and exporting purpose.

6.2 Collision Detection

Once the tool and the clay object were *voxelized*, checking collision between them was fairly easy. Figure 6.7 shows the bounding rectangles for the voxelized objects on the XY Plane.

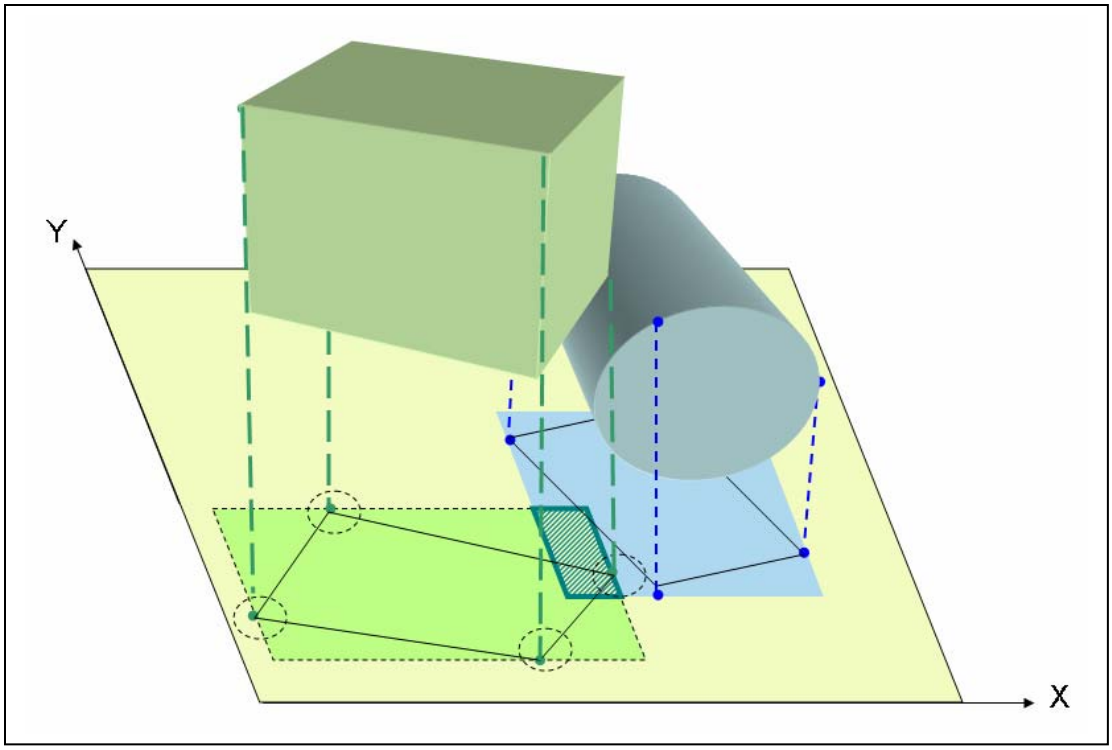


Figure 6.7: Collision Detection

Let location of the local origin (translationX, translationY) of a 3D object be denoted by (T_x, T_y), the unit length by U and the bounds in X and Y directions be S_x and S_y. To easily define a rectangle with diagonally opposite corners (X_1,Y_1) and (X_2,Y_2) we use the notation,

$$R([X_1 \quad Y_1] \quad [X_2 \quad Y_2]) \tag{6.3}$$

The bounding rectangle of a voxelized object can be defined as,

$$R([T_x U \quad T_y U] \quad [(T_x + S_x)U \quad (T_y + S_y)U]) \tag{6.4}$$

The rectangles bounding the projection of the two colliding objects are,

$$R([T_x^1 U^1 \quad T_y^1 U^1] \quad [(T_x^1 + S_x^1)U^1 \quad (T_y^1 + S_y^1)U^1]) \tag{6.5a}$$

$$R([T_x^2 U^2 \quad T_y^2 U^2] \quad [(T_x^2 + S_x^2)U^2 \quad (T_y^2 + S_y^2)U^2]) \tag{6.5b}$$

In our case, we found the collision between the most detailed versions of discretized objects. Since U=1 for the most detailed objects, we had $U^1=U^2=1$. The two bounding rectangles then become,

$$R\left(\left[T_x^1 \quad T_y^1\right] \quad \left[T_x^1 + S_x^1 \quad T_y^1 + S_y^1\right]\right) \tag{6.6a}$$

$$R\left(\left[T_x^2 \quad T_y^2\right] \quad \left[T_x^2 + S_x^2 \quad T_y^2 + S_y^2\right]\right) \tag{6.6b}$$

These are shown are shaded rectangles in the figure 6.7. The common region between the two bounding rectangles is shaded and highlighted.

Let common rectangular region be,

$$R\left(\left[A_1 \quad B_1\right] \quad \left[A_2 \quad B_2\right]\right) \tag{6.7}$$

Let i,j be a point (X_i,Y_j) inside the common rectangle. Then the local coordinates of the point corresponding to the two objects will be respectively,

$$\left(X_i - T_x^1, Y_j - T_y^1\right) and \left(X_i - T_x^2, Y_j - T_y^2\right) \tag{6.8}$$

Where,

$A_1 \le X_i < A_2$
$B_1 \le Y_i < B_2$

The columns at the local coordinates of the two objects had to be checked for collision,

```
Shared (
        Object1.gridPoint[ Xi- T_x^1 ,Yj - T_y^1 ].z,
        Object2.gridPoint[ Xi- T_x^2 ,Yj - T_y^2 ].z ,
        &union,&intersection
)
```

If intersection.size() was not zero, SortedVector *intersection* contained the data about intersection column. Finally, a new object was built up like this:

intersectionObject. gridPoint[Xi- A_1,Yj –B_1].z.push_back(intersection)

Thus we had not only detected collision, we had also obtained the volume shared by the two colliding objects.

6.3 Input for FEM Mechanics

We were able to determine the collisions and overlaps between the clay object and constraint plane and sculpting tool. To be useful this information had to be converted into suitable format to be fed to FEMSolver. This section deals with the details about different algorithms used to extract boundary conditions and forces for FEM analysis.

6.3.1 Input Force Estimation

The region of intersection of tool and clay was obtained during collision detection (section 6.2). Each node from FEM data (which is of type VoxelData) was transformed to global coordinates and was queried (section 5.1) for intersection data. If it was in the interior of intersection data, a force was applied proportion to the volume of intersection and along the direction of tool orientation.

6.3.2 Constraint Determination

A plane with a point on that plane was provided to constraint material. Practically, clay object is supported by a base or human hand while it is being deformed by the tool. This constraint plane mimics human hand holding the base of the clay object. The FEM elements that intersected with the plane were rigidly constrained at their nodes. Except

for the constrained elements, all other elements were assembled and solved because a multi-contact tool could simultaneously manipulate material on either side of the plane.

6.3.3 Cross-section Determination

The point on the plane was provided for special cases. The plane might be intersecting a concave object on multiple cross-sections (figure 6.8).

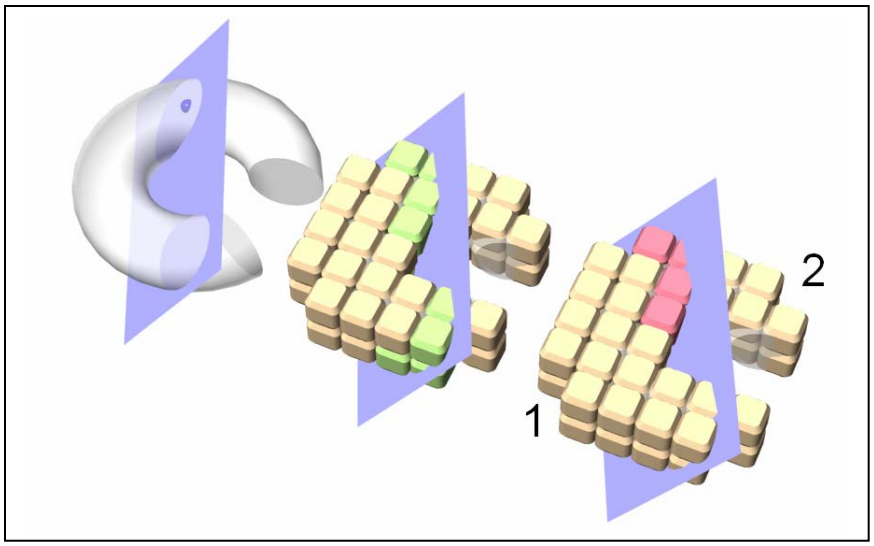


Figure 6.8: Cross-section Determination

Once the constrained elements were determined, they could be distributed over several cross-sections. The exact cross-section was determined by indexing algorithm. Indexing algorithm (figure 6.9) assigns contiguous elements same index (or ID) and different indices to different clusters of contiguous elements. All those elements that intersected the plane but did not lie in the constrained cross-section were treated as ordinary elements.

The flowchart in figure 6.9 checks the index with the right neighbor (along positive X axis). A similar check is done with neighbor along positive Y axis as well.

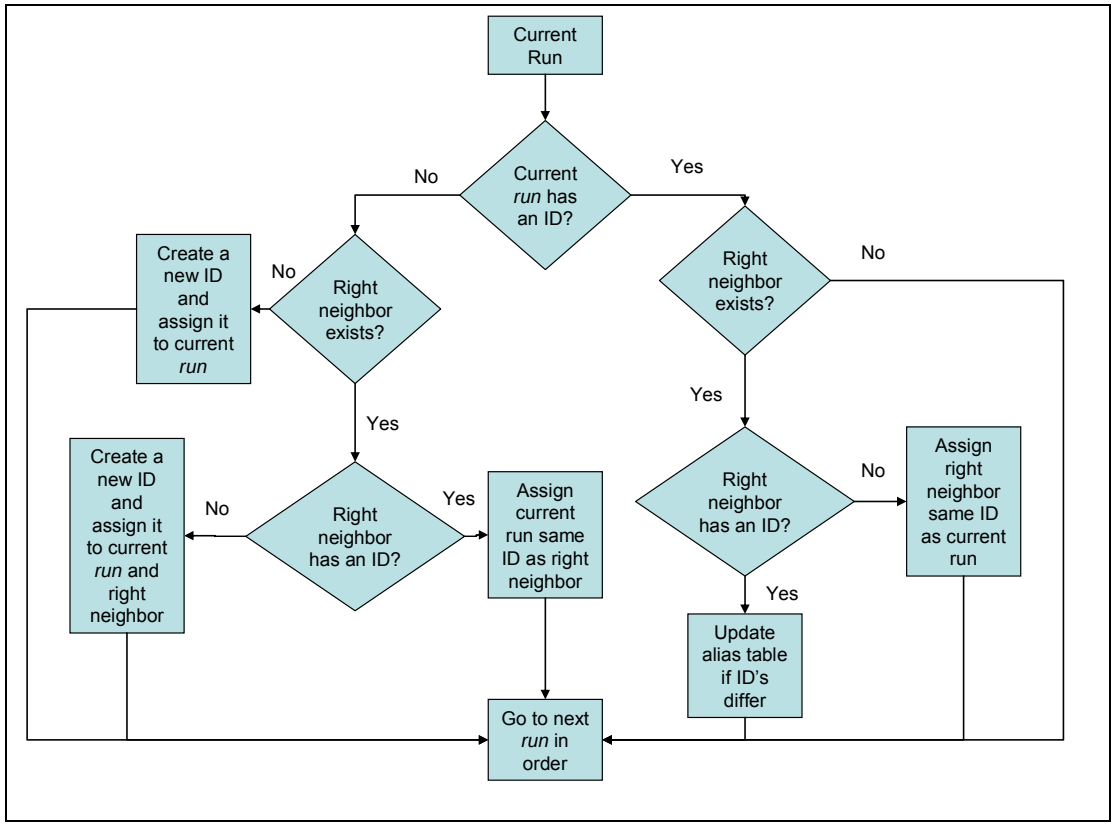


Figure 6.9: Indexing Algorithm

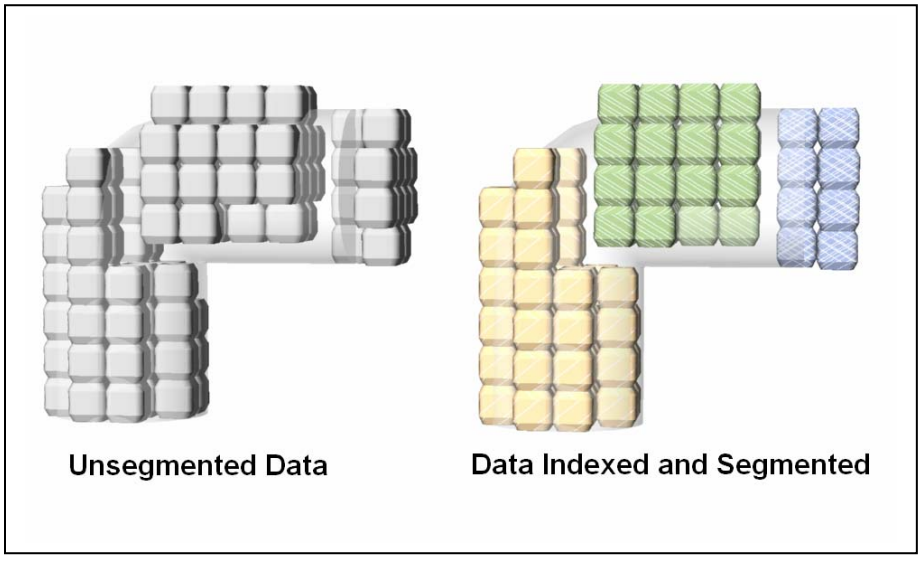


Figure 6.10 Indexing and Segmentation

The planar point would lie inside exactly one of the two (or more) cross-sections. To determine the correct cross-section and all the elements that belonged to it, the elements were segmented using indexing algorithm (figure 6.10).

6.4 Data Processing

The algorithms described here are the functions that operate at bit (column) level.

6.4.1 Basic Boolean Operation

These operations work on pairs of columns. It should be remembered that a column is a set of all runs (pairs of boundaries) at a specific location on XY grid. These *core* algorithms for Boolean operations (figure 6.11) directly use the boundary data and do not iterate on each voxel.

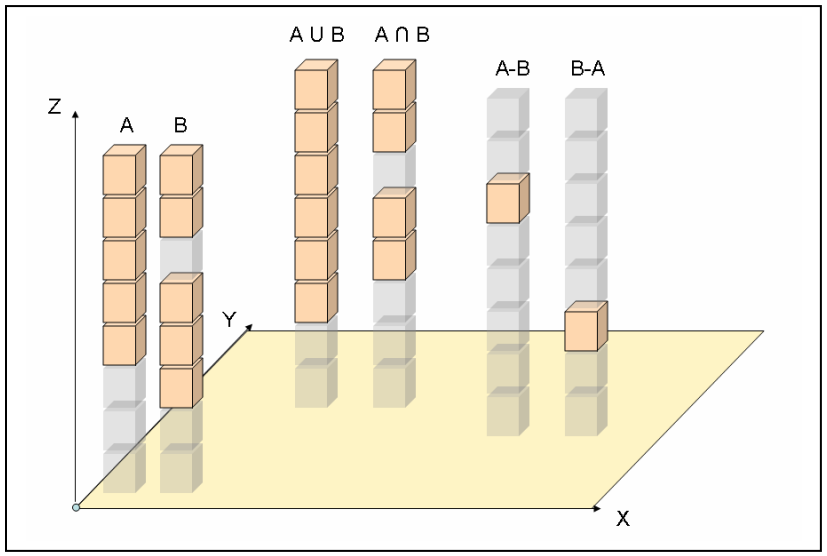


Figure 6.11: Boolean Operations

The inputs and outputs were compressed and intermediate steps for Boolean operations were directly carried out on the compressed data. This saved a great amount of computation time. Since these algorithms are big and complex to express, the flowchart

in figures 6.12a and 6.12b show the algorithm for evaluating A-B and B-A. Algorithm for AUB and A∩B is similarly derived. In the flowchart, the inputs are columns *a* and *b* and the output are columns *exp_a* and *exp_b* such that, $\exp\_a = a - b$ and $\exp\_b = b - a$. The algorithm basically iterates exactly once through all boundaries of two adjacent columns and determines the non-overlapping regions between the two columns. Flowcharts in figures 6.12a and 6.12b are linked through connectors A and B.

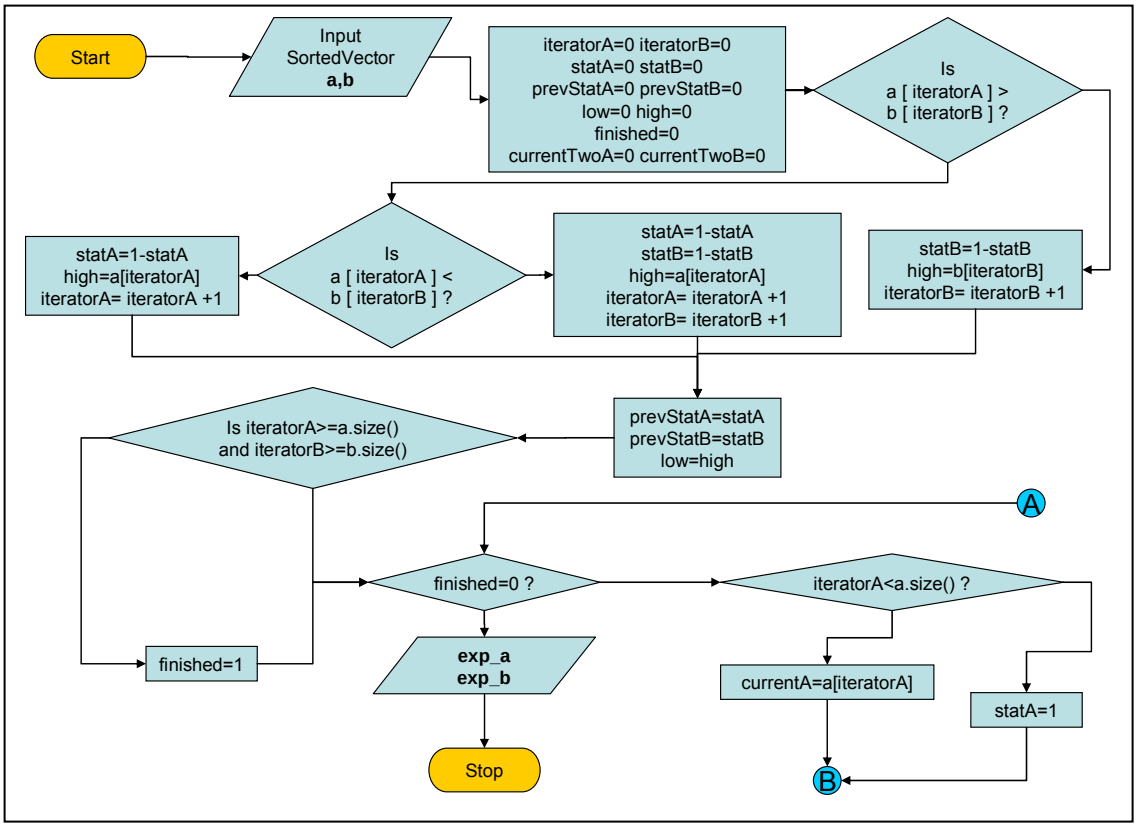


Figure 6.12 a: Flowchart of function unshared()

continued …

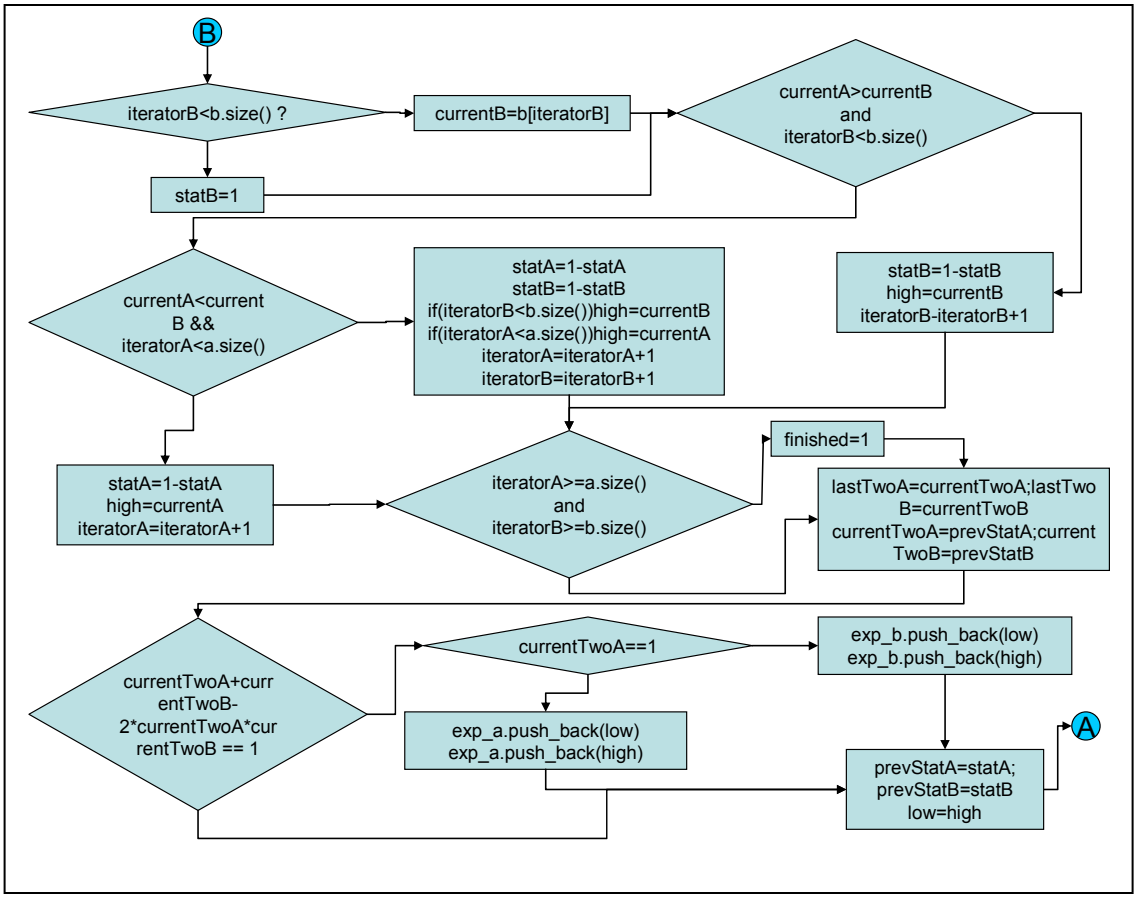


Figure 6.12 b: continuation of flowchart of function unshared()

(Note: Trivial situations where a.size() and/or b.size() are 0 should be dealt separately)

6.4.2 FEM Result Interpolation

The deflection field inside an FEM element was evaluated using trilinear interpolation.

$$d(x,y,z) = a_0 + a_1 x + a_2 y + a_3 z + a_4 xy + a_5 yz + a_6 zx + a_7 xyz \quad (6.9)$$

If the location was not inside any FEM element, the element having the *nearest surface* was used.

6.4.3 Smoothing Algorithm

This section describes the derivation of Catmull-Clark subdivision based LPF.

The bicubic uniform B-spline patch is subdivided into four subpatches. The new control points $P^1_{i,j}$ can be written in terms of original points $P^0_{i,j}$ as (see Appendix 6),

$$\begin{bmatrix} P^1_{0,0} & P^1_{0,1} & P^1_{0,2} & P^1_{0,3} \\ P^1_{1,0} & P^1_{1,1} & P^1_{1,2} & P^1_{1,3} \\ P^1_{2,0} & P^1_{2,1} & P^1_{2,2} & P^1_{2,3} \\ P^1_{3,0} & P^1_{3,1} & P^1_{3,2} & P^1_{3,3} \end{bmatrix} = M_S \begin{bmatrix} P^0_{0,0} & P^0_{0,1} & P^0_{0,2} & P^0_{0,3} \\ P^0_{1,0} & P^0_{1,1} & P^0_{1,2} & P^0_{1,3} \\ P^0_{2,0} & P^0_{2,1} & P^0_{2,2} & P^0_{2,3} \\ P^0_{3,0} & P^0_{3,1} & P^0_{3,2} & P^0_{3,3} \end{bmatrix} M_S^T \tag{6.10}$$

$$where, M_S = \frac{1}{8}\begin{bmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \end{bmatrix}$$

On expanding the matrix product and rearranging terms, the mask can be defined as

$$P^1_{face\,center} = \begin{pmatrix} P^0_{vertex1} & P^0_{vertex2} & P^0_{vertex3} & P^0_{vertex4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix} \tag{6.11a}$$

$$P^1_{edge\,point} = \begin{pmatrix} P^0_{end\,point1} & P^0_{end\,point2} & P^1_{face\,center1} & P^1_{face\,center2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix} \tag{6.11b}$$

$$P^1_{edge\,center} = \begin{pmatrix} P^0_{end\,point1} & P^0_{end\,point2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \tag{6.11c}$$

$$P^1_{vertex} = \begin{pmatrix} P^1_{edge\,cen1} & P^1_{ed\,cen2} & P^1_{ed\,cen3} & P^1_{ed\,cen4} & P^1_{face\,cen1} & P^1_{f\,cen2} & P^1_{f\,cen3} & P^1_{f\,cen4} & P^0_{vertex} \\ \frac{4}{16} & \frac{4}{16} & \frac{4}{16} & \frac{4}{16} & \frac{-1}{16} & \frac{-1}{16} & \frac{-1}{16} & \frac{-1}{16} & \frac{4}{16} \end{pmatrix} \tag{6.11d}$$

P^1_{vertex} in equation (6.11d) can be written as,

$$P^1_{vertex} = \begin{pmatrix} \overline{P}^1_{edge\,cen} & \overline{P}^1_{face\,cen} & P^0_{vertex} \\ 1 & \frac{-1}{4} & \frac{1}{4} \end{pmatrix} \tag{6.12}$$

where, $\overline{P}^1_{edge\,cen}$ and $\overline{P}^1_{face\ cen}$ are the averages of four edge centers and four face centers respectively. For a general vertex of valence *n*,

$$P^1_{vertex} = \begin{pmatrix} \overline{P}^1_{edge\,center} & \overline{P}^1_{face\,center} & P^0_{vertex} \\ \frac{4}{n} & \frac{-1}{n} & \frac{n-3}{n} \end{pmatrix} \tag{6.13}$$

Let *n*=valency

```
for (i=0;i<n;i++)
{
        A=A+AdjacentPoint1[i]
        A=A+AdjacentPoint2[i]
        B=B+DiagonalPoint[i]
}
```

This particular code will give

$$\begin{aligned} A &= 2\sum_{i=1}^{n} AdjacentPo\,\mathrm{int}_i \\ B &= \sum_{i=1}^{n} DiagonalPo\,\mathrm{int}_i \end{aligned} \tag{6.14}$$

By manually adding the edge centers of edges around a point of valence *n*,

$$\overline{P}^1_{edge\,center} = \frac{1}{2n}\left(nP + \sum_{i=1}^{n} AdjacentPo\,\mathrm{int}_i \right) \tag{6.15}$$

Using equation (6.14), equation (6.15) can be expressed as,

$$\overline{P}^1_{edge\,center} = \frac{1}{4n}\left(2nP + A\right) \tag{6.16}$$

Similary we can manually express average of face centers as,

$$\overline{P}^1_{face\,center} = \frac{1}{4n}\left(nP + 2\sum_{i=1}^{n} AdjacentPo\,\mathrm{int}_i + \sum_{i=1}^{n} DiagonalPo\,\mathrm{int}_i \right) \quad (6.17)$$

and combining (6.14) and (6.17) we can express,

$$\overline{P}^1_{facecenter} = \frac{1}{4n}\left(nP + A + B\right) \quad (6.18)$$

Expanding equation (6.13) we have,

$$P^1_{vertex} = \frac{4}{n}\overline{P}^1_{edgecenter} - \frac{1}{n}\overline{P}^1_{facecenter} + \frac{n-3}{n}P^0_{vertex} \quad (6.19)$$

Plugging in values from equations (6.16) and (6.18) in equation (6.19) we get,

$$P^1_{vertex} = P^0_{vertex} + \frac{1}{4n^2}(3A + B - 7nP^0_{vertex}) \quad (6.20)$$

Finally, if

$$P^1_{vertex} = P^0_{vertex} + \Delta P^0_{vertex} \quad (6.21)$$

then from equations (6.20) and (6.21) we can have a compact expression,

$$\Delta P^0_{vertex} = \frac{1}{4n^2}(3A + B - 7nP^0_{vertex}) \quad (6.22)$$

Equation (6.22) is the final implementation of our mask shown in figure 4.7. The vertex normal can be estimated from ΔP^0_{vertex} by normalization.

6.4.4 Data Integration

To generate triangulated data from Surface Data, the FEM deformation, *d*, was added to the original point coordinates P^i. P^i is same as the key (X,Y,zBase) of the surface point.

$$P^f = P^i + d \quad (6.23)$$

Before writing the output to an STL file, smoothing data, ΔP_s, was also applied along with the final FEM deformation.

$$P^f = T_0 + \left(P^i + d + \alpha \Delta P_s\right) \tag{6.24}$$

The factor α controls the contribution of smoothing operations which may be used to preserve some sharp features. In the current implementation, it is 1. The translation T_0 was applied to shift the object to first quadrant. The facet normal was calculated using cross multiplication of edge vectors.

6.5 Rendering

A separate triangulated object was maintained for rendering. The coordinates of the vertices in this object were calculated from the Surface Data by applying appropriate scaling.

$$P^f = sf * \left(P^i + d + \alpha \Delta P_s\right) \tag{6.25}$$

Here *sf* is the scaling factor.

6.6 Time Keeping

The C++ clock() function was used to query system clock time. This function worked with a resolution of 10 ms on Linux. A timer class was implemented to set multiple timers on and off before and after the part of code for which we were interested in measuring the execution time. The results have been analyzed in next chapter.

Chapter 7: Results

As seen in chapter 4, each user manipulation invoked a series of operations for data generation and processes. These operations accessed input data from the memory, performed calculations and wrote it back to the memory. Fetching data and performing calculations, both require computational time depending on the performance capabilities of processor and memory. This time can be a significant period if the operations have to be performed millions of time in each manipulative iteration. In our sculpting system, not only the tool and clay object could assume complex shapes, they could interact in complex ways. It was almost impossible to derive a complete analytical mathematical expression for the total computation time required for each clay manipulation. Hence a statistical study was performed.

7.1 Testing

In broad terms, the performance of the software was dependent on three factors (or *effects*). Firstly, the number of voxels that had to be handled effected the time taken for generation of volumetric and surface data. Secondly, the number of elements used for FEM determined the computational cost of FEM analysis. Lastly, the topology of the object influenced the ease of performing overall volumetric and surface operations. The size of surface data depended on amount of voxel data and its topology. Though the total time taken during each iteration was what one would be most concerned, we were also interested in break up for individual tasks. Responses like how much time it took to prepare for FEM, perform FEM calculations, apply deformations and smoothing on new surface and generate new volumetric data were important. A statistical experiment was

carried out and the results were analyzed to evaluate the nature and severity of the effects on responses just mentioned. Three different shapes were studied viz. a sphere, a torus and a slab. For each shape, combinations of different levels of discretization for voxel and element generation were used. The objects were discretized to data sets that consisted of 10000, 100000 and 1000000 voxels. For each level of voxel discretization, FEM structures of 300 and 500 nodes were constructed. This is summarized in table 7.1.

Object Type (3 levels)	Sphere	Torus	Slab
Number of Voxels (3 levels)	10,000	100,000	1,000,000
Number of Elements (2 levels)	300	500	

Table 7.1: Factors and Levels

A full factorial experiment for these three effects involved 18 (3x3x2) cases. We decided to go for two replications for each case which made it 36. Replications could be used to analyze the performance of system clock so that one can be assured that it could capture the variation in measured time. For each case, responses like time required for FEM preprocessing, FEM execution, surface generation, smoothing and voxel data generation were measured. The measured data are listed in tables A7.1 and A7.2 (Appendix 7). The frame rates in table A7.2 were calculated by calling time keeping function within the display() function. There rates are the based on time periods averaged over a number of frames. The hardware and software used to obtain the experimental results were same as those used during development (section 4.5).

7.2 Analysis

Analysis of variance (ANOVA) (Montgomery 2001) was carried out on the responses. The output is provided in Appendix 8. The data distribution was normal and free from trends and patterns (figures A8.1 through A8.6 in Appendix 8). The *p-values*

obtained for different responses towards different factors are summarized in table 7.2. A low p-value ($p < 0.05$) indicates that the variation in the measured quantity is significant and it is unlikely that it has occurred by chance.

	FEM Preprocessing	**FEM Execution**	**Surface Generation**	**Smoothing**	**Regeneration**	**Total**
Object	0.686	0.000	0.000	0.006	0.000	0.000
Voxels	0.000	0.425	0.000	0.000	0.000	0.000
Nodes	0.547	0.001	0.077	0.606	0.804	0.000

Table 7.2: ANOVA Summary

We see that FEM preprocessing time did not vary much between objects. During the process, bins corresponding to FEM elements were created. Each voxel was then picked up and dropped in appropriate bin to form elements. Therefore the process was more dependent on number of voxels ($p=0.000$) and not much on number of nodes (or elements). This is also evident from figures 7.1 and 7.2.

Execution of FEM solution depended on the degrees of freedom which was proportional to number of nodes ($p=0.001$). FEM solution also relied on the connectivity of elements which is characteristic of type of object ($p=0.000$). It was found that computational time for this was fairly independent of the number of voxels. In figure 7.3 we find that Slab took significantly more time. Because of uniformity and compactness there were dense fronts to be solved. The rapid increase in solution time for slab when the number of nodes was increased from 300 to 400 in figure 7.4 also indicates this.

The time required for surface generation depended on the topology of the object and number of voxels ($p=0.000$ for both). We also saw a good amount of variation in surface generation time that came from the number of nodes ($p=0.077$). This could be attributed to the fact that nodes were used to interpolate deflections on surface data. Since the Torus has a complex shape, it took more time to generate it's surface than other

objects (figure 7.5). Surface generation time was largely dependent on number of voxels and to a lesser degree on the number of nodes (because of interpolation step). This slight positive correlation between number of nodes and surface generation time was significant for cases with larger number of voxels (figure 7.6).

Smoothing operation on the other hand depended on topology and amount of discretization and was independent of number of nodes ($p=0.606$). Smoothing was computationally more time consuming for torus because of it's complex topology (figure 7.7). Smoothing time bore a strong relationship with the number of voxels (but not with number of nodes).

The creation of STL object and new Voxel Data (Regeneration) was dependent on object type and number of voxels. Time required for STL and new voxel data generation was influenced by number of voxels to be generated. These operations were significantly faster for sphere because of the simplicity in shape (figures 7.9, 7.10).

The total time, which was the sum of time taken for all steps, critically relied on all the three factors. As expected, total time was more for higher number of voxels and nodes. Sphere performed best because of its simple convex shape. Slab performed the worst(Figure 7.11). Figure 7.12 makes it clear that slab performed the worst because of slow FEM and surface evaluation. Slab, in fact, had the highest surface to volume ratio.

Results of more detailed statistical analysis of factors and their interactions for the full factorial experiment are presented in Appendix 9.

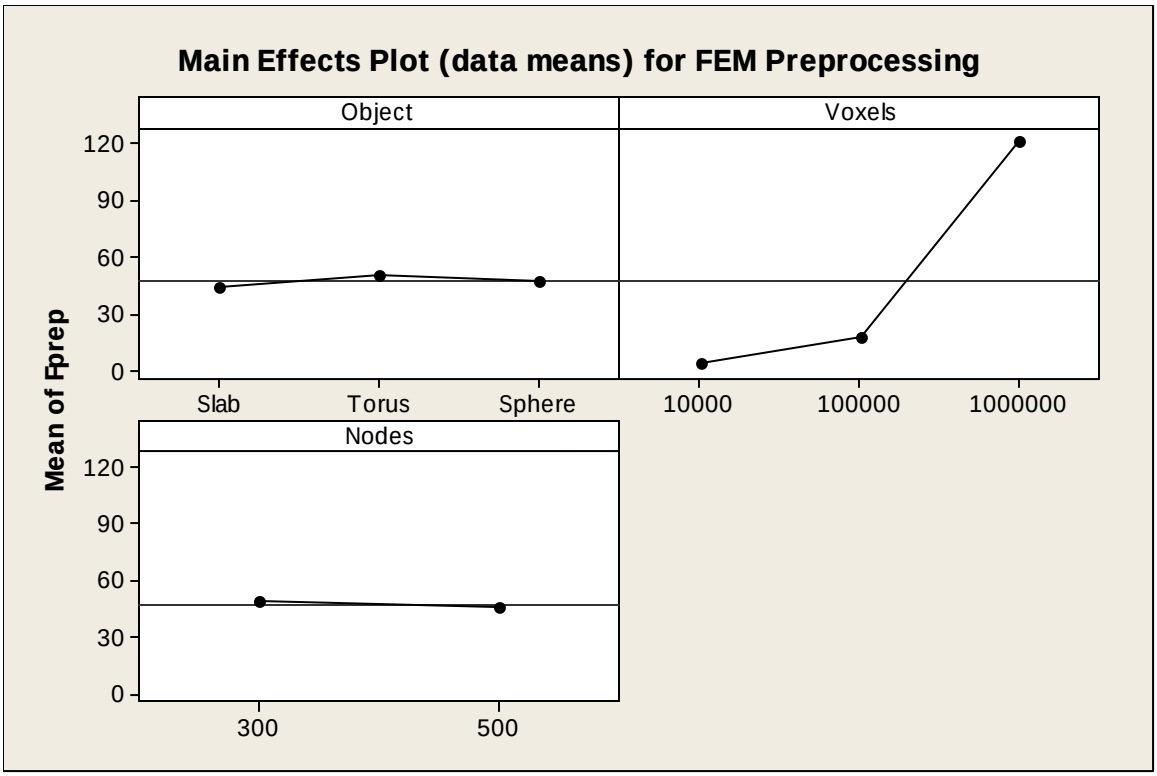


Figure 7.1: Main Effects Plot for FEM Preprocessing

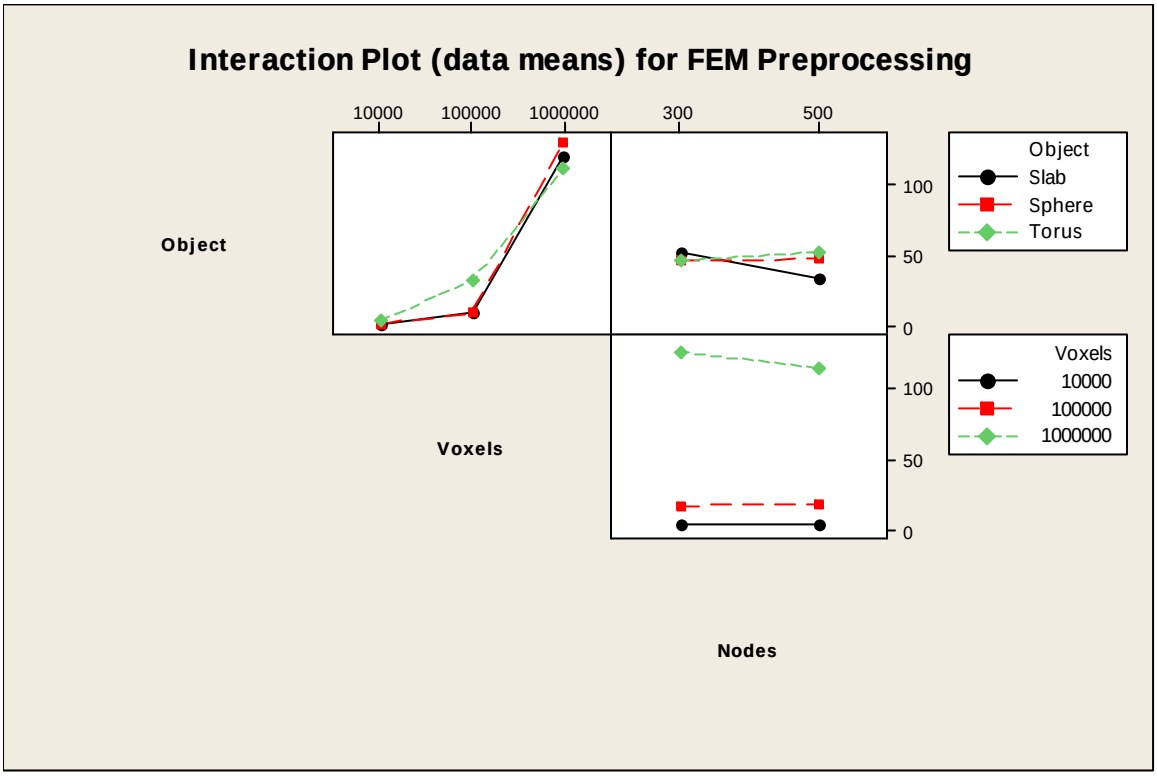


Figure 7.2: Interaction Plot for FEM Preprocessing

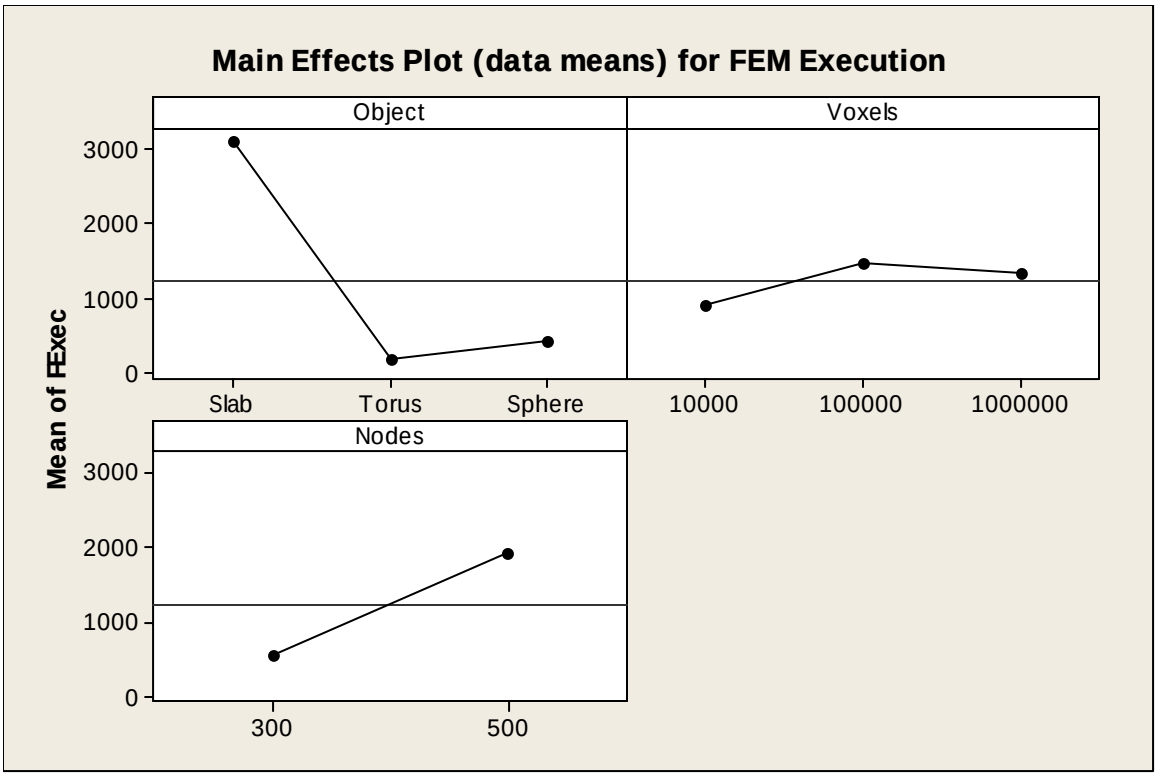


Figure 7.3: Main Effects Plot for FEM Execution

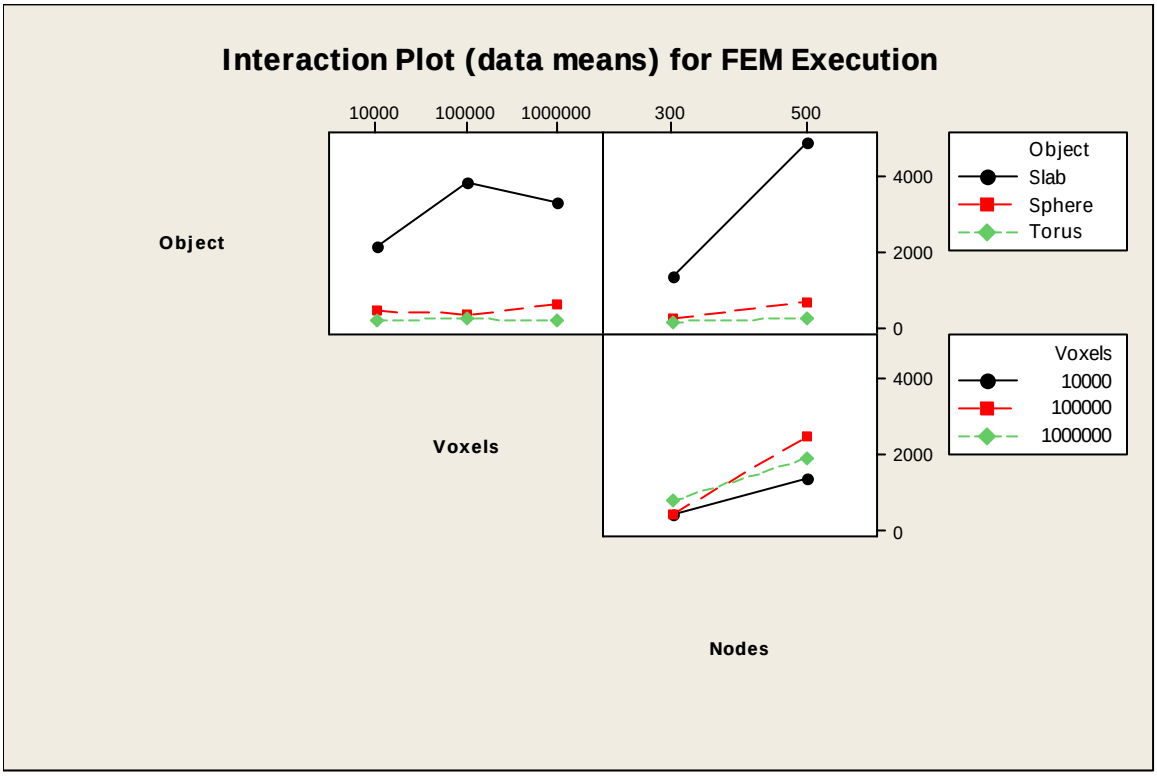


Figure 7.4: Interaction Plot for FEM Execution

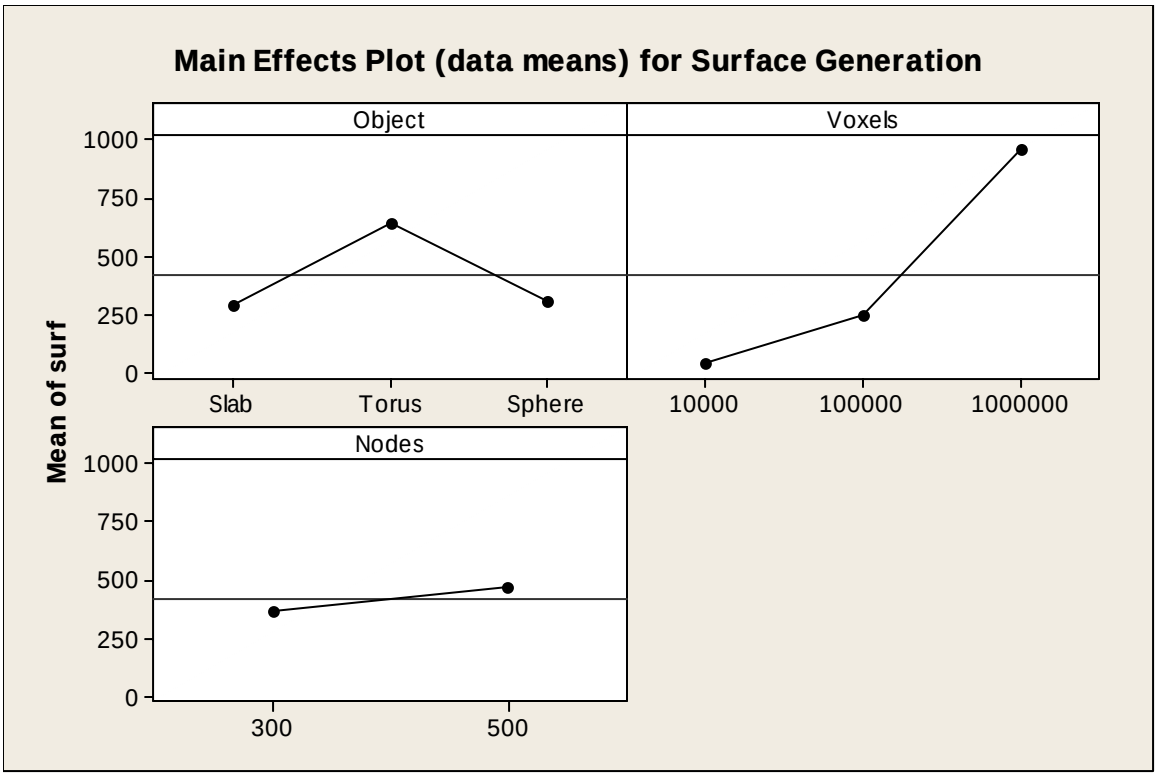


Figure 7.5: Main Effects Plot for Surface Generation

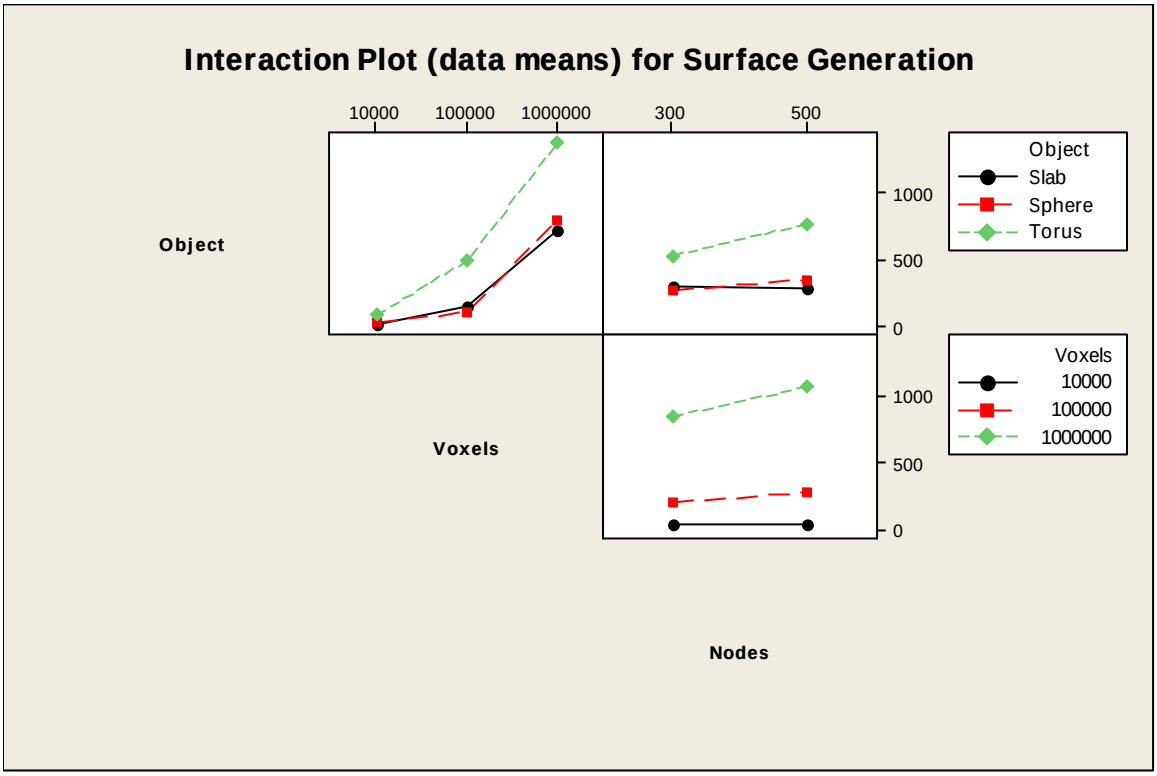


Figure 7.6: Interaction Plot for Surface Generation

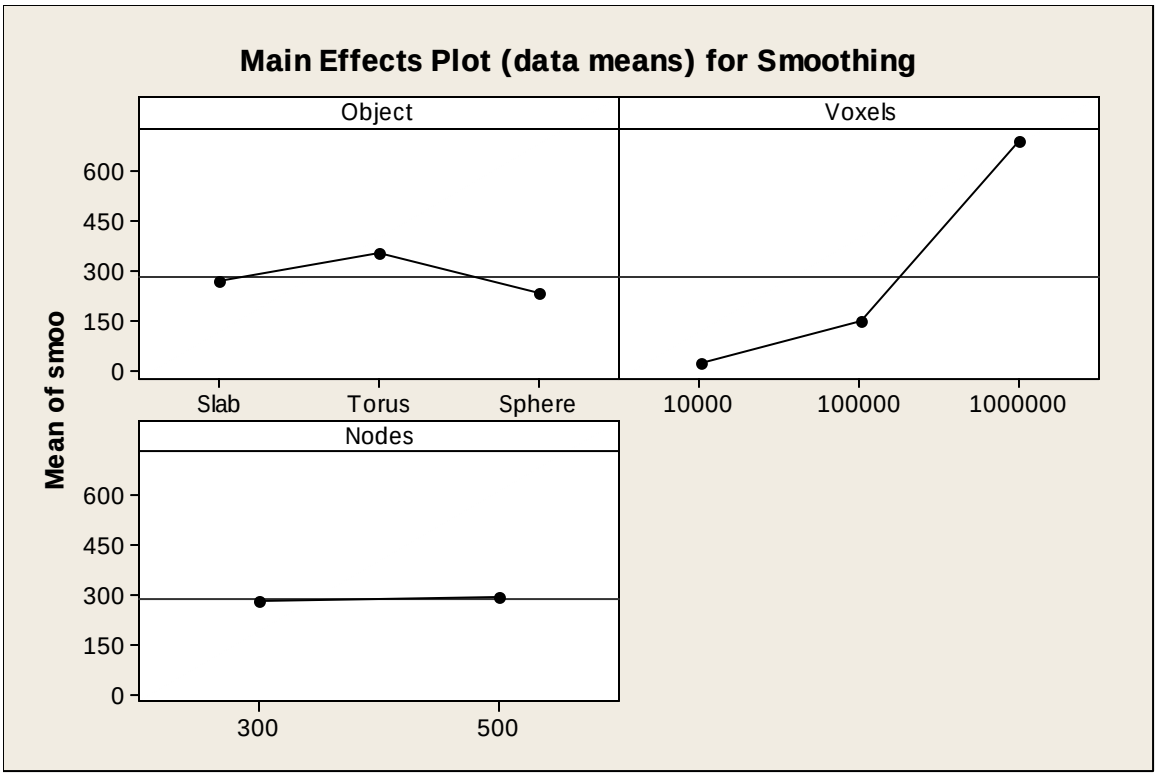


Figure 7.7: Main Effects Plot for Smoothing

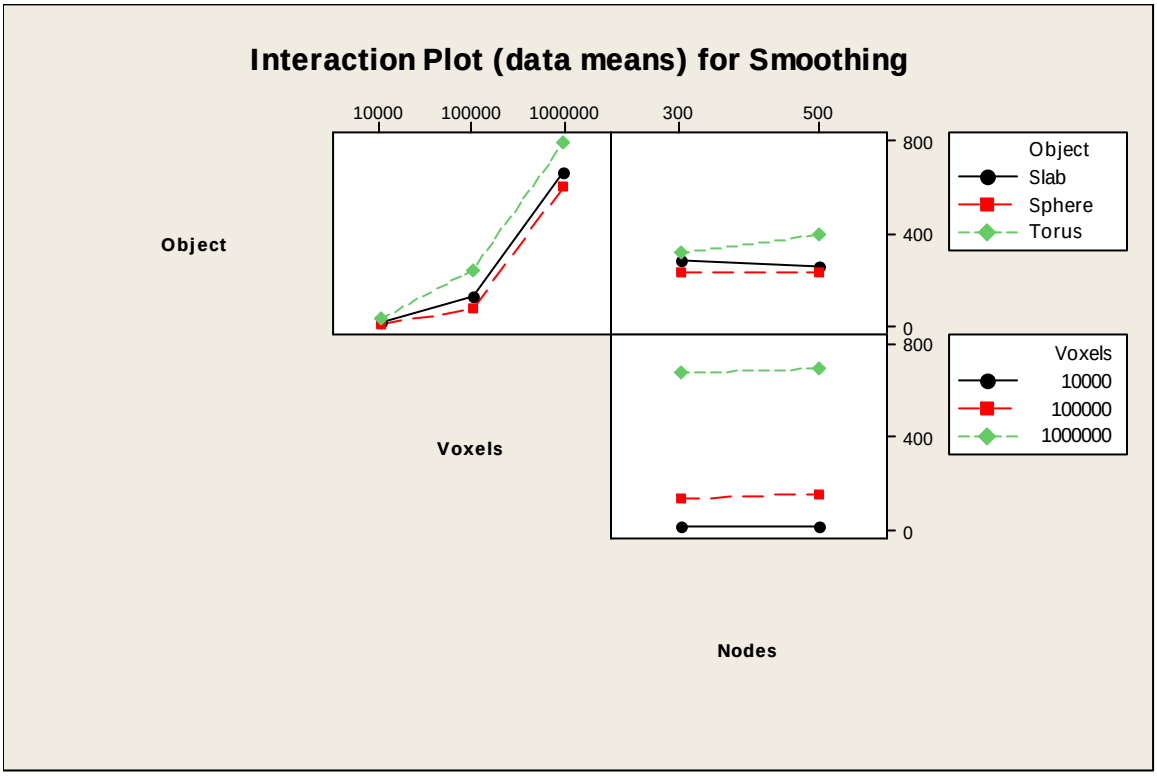


Figure 7.8: Interaction Plot for Smoothing

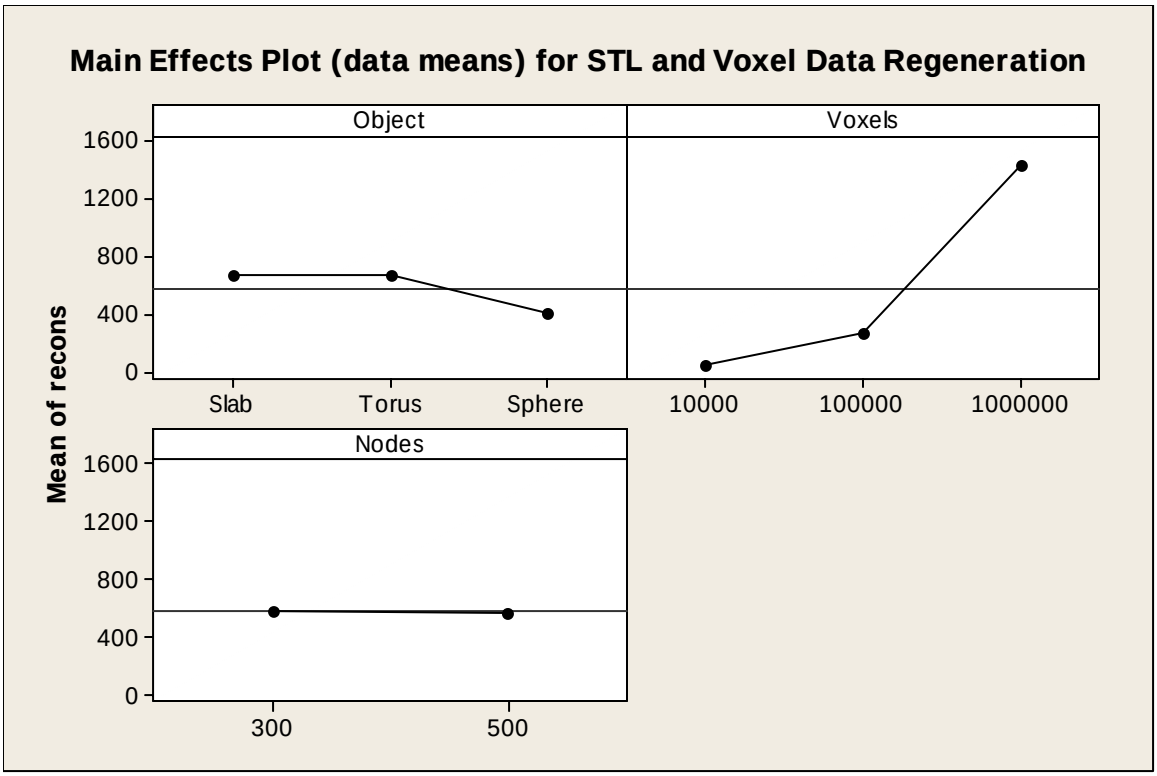


Figure 7.9: Main Effects Plot for Regeneration

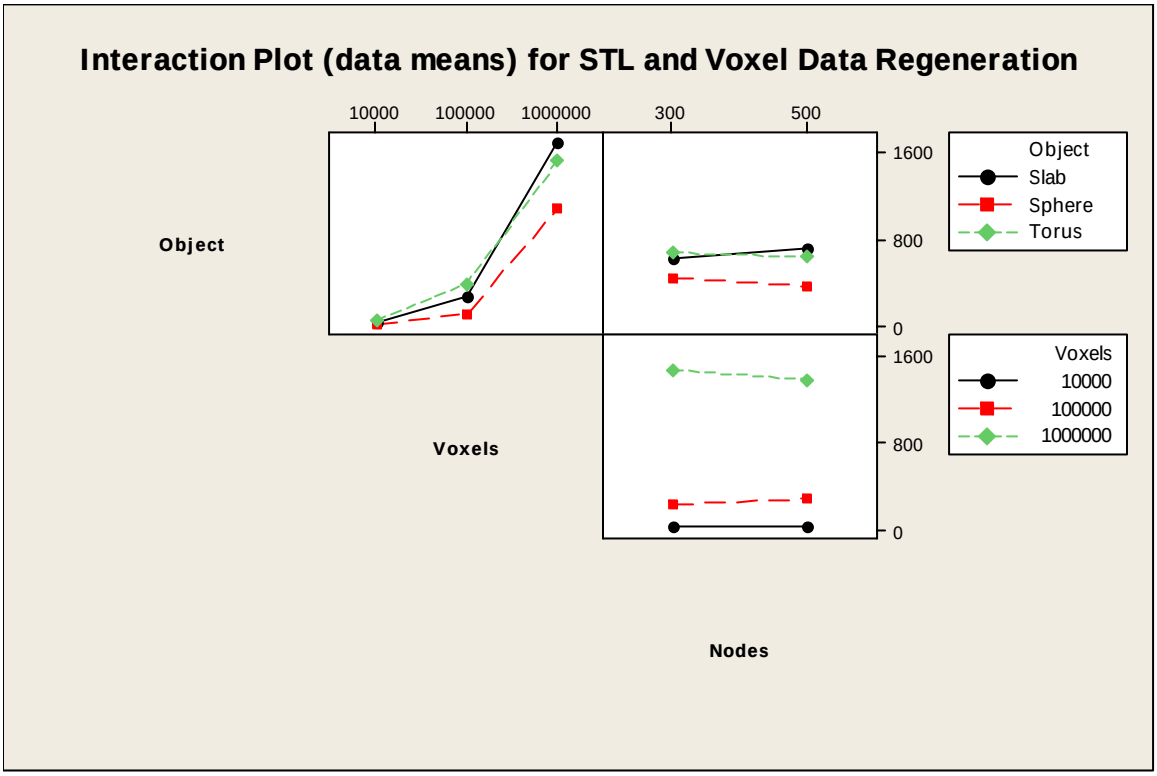


Figure 7.10: Interaction Plot for Regeneration

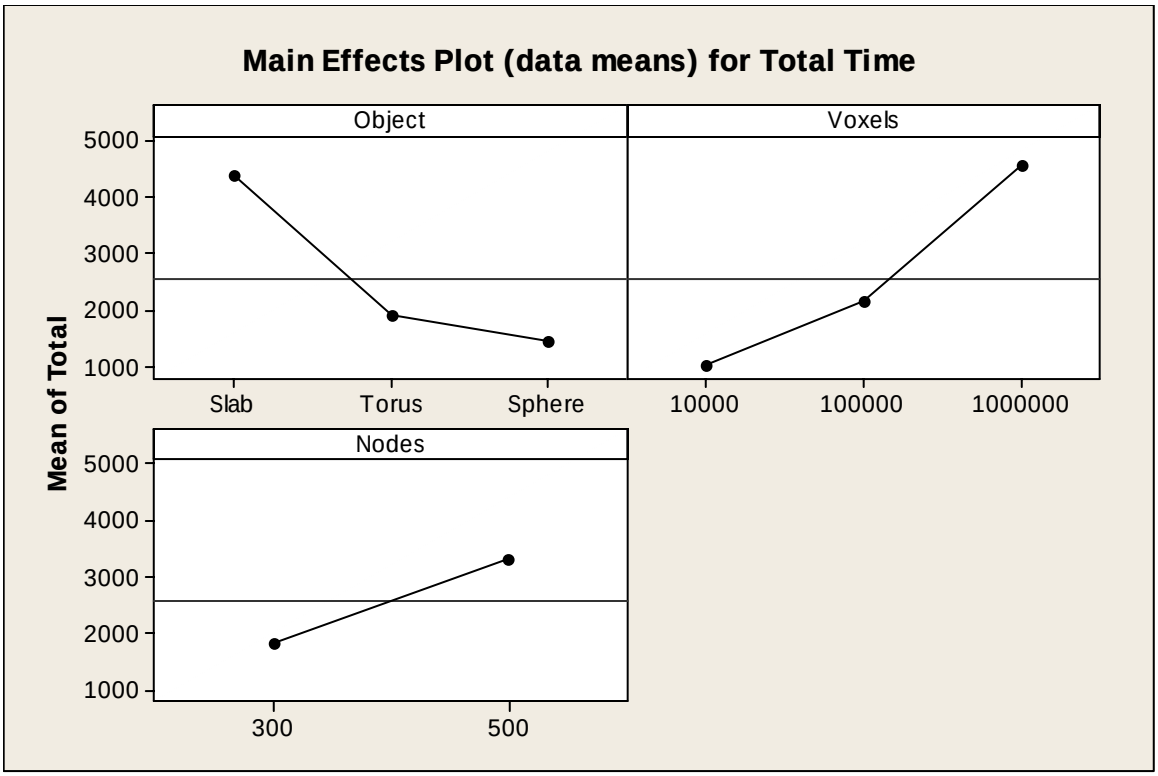


Figure 7.11: Main Effects Plot for Total Time

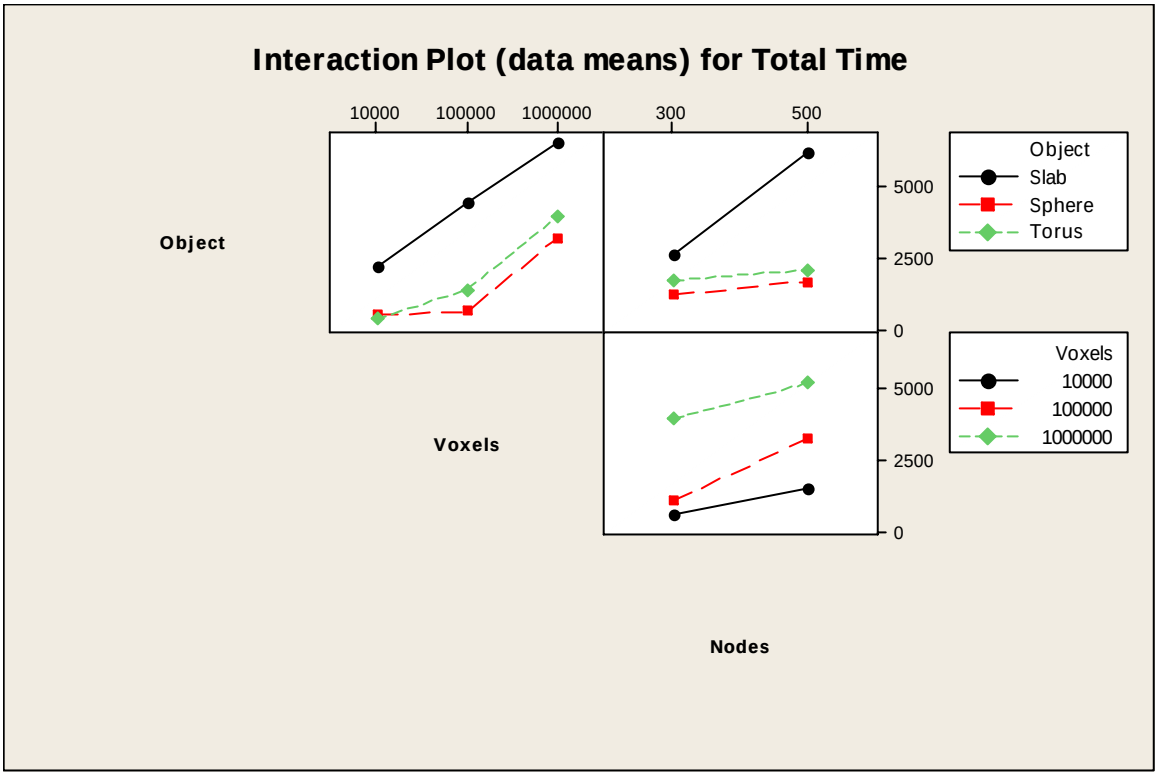


Figure 7.12: Interaction Plot for Total Time

Figures 7.13 through 7.16 show the contour plot of total time for sphere, torus, slab and *all objects combined* respectively. The scalability for sphere (figure 7.13) is fairly straightforward to interpret. Computation time increased with increase in number of voxels or number of elements or both. The straight parallel bands indicate that the gradient is smooth. The width of the bands representing larger duration of time (green) is lesser than that of those representing smaller duration of time.

A linear fit on the logarithmic values suggested following relationships:

$$\begin{aligned} time &\propto (voxels)^{0.47}, keeping\ nodes = 300 \\ time &\propto (nodes)^{1.4}, keeping\ voxels = 10000 \end{aligned} \tag{7.1}$$

The near-linear relationship of time with nodes is due to the fact that frontal solver tends to make the computational cost proportional to the degrees of freedom which in turn is proportional to (four times) the number of nodes. The dependency on voxels was somewhere between L and L^2, where L is linear dimension, i.e., $L=voxels^{.33}$ but closer to L^2 (surface area).

Sphere exhibited a well balanced response to the two main effects. Torus on the other hand (figure 7.14) was more sensitive to number of voxels. The almost vertical bands indicated that variation due to FEM calculations was negligible as compared to variation from volumetric and surface data operations. Torus, therefore, showed a strong affinity to number of voxels rather than number of nodes. This could be attributed to the complexity in its shape.

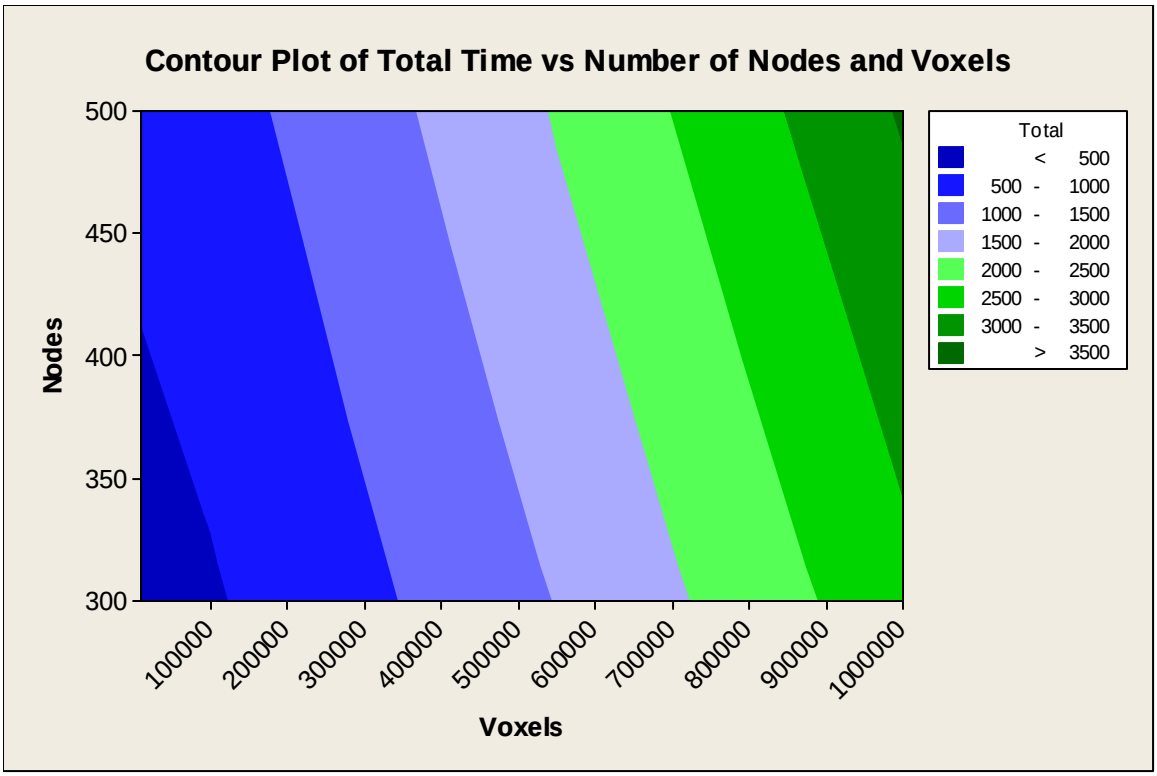


Figure 7.13: Contour Plot of Total Time for Sphere

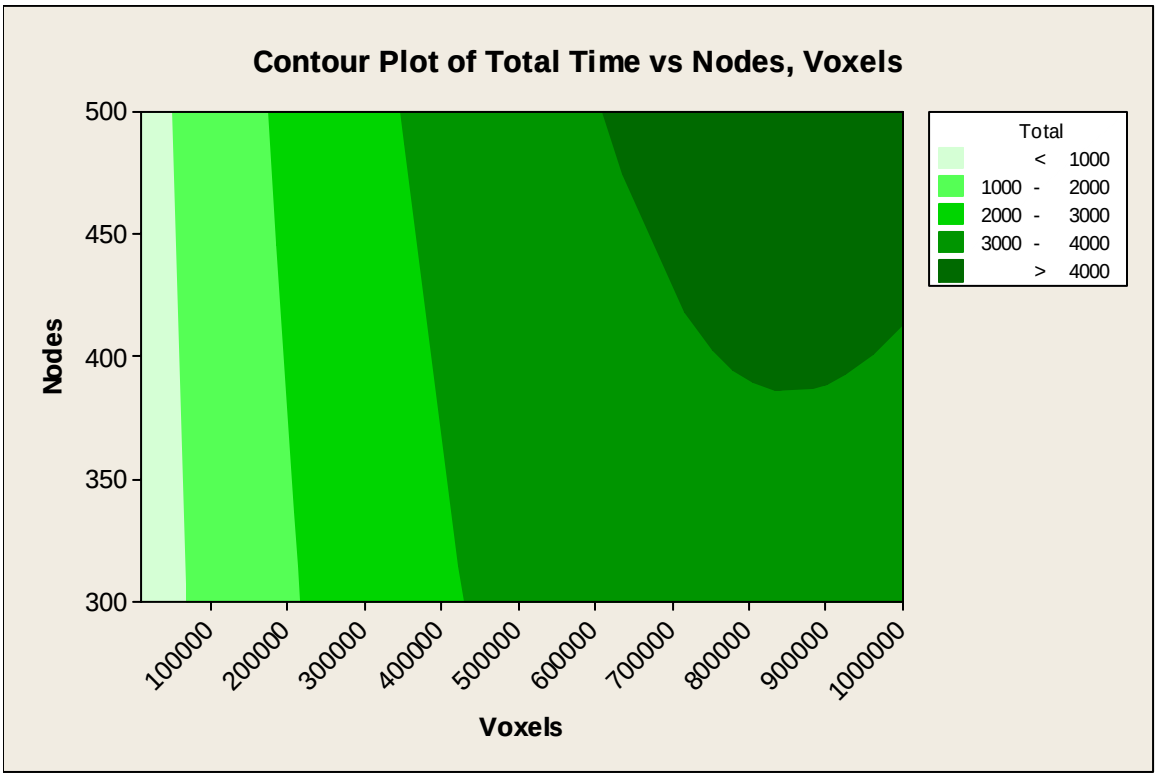


Figure 7.14: Contour Plot of Total Time for Torus

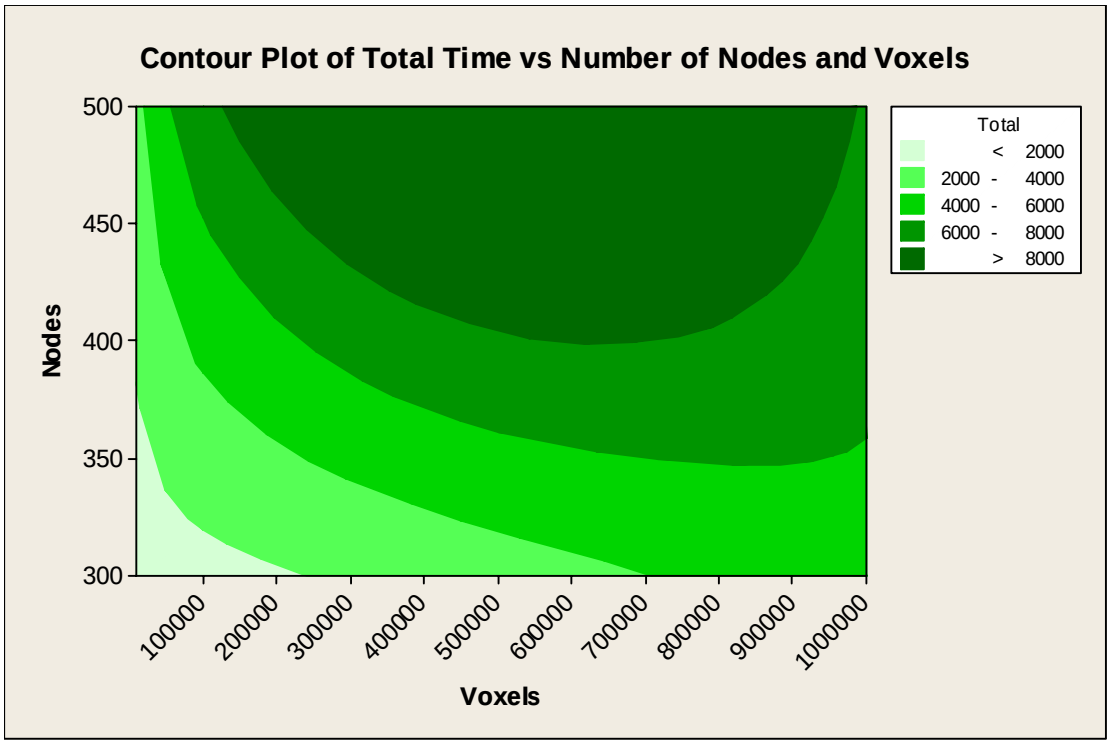


Figure 7.15: Contour Plot of Total Time for Slab

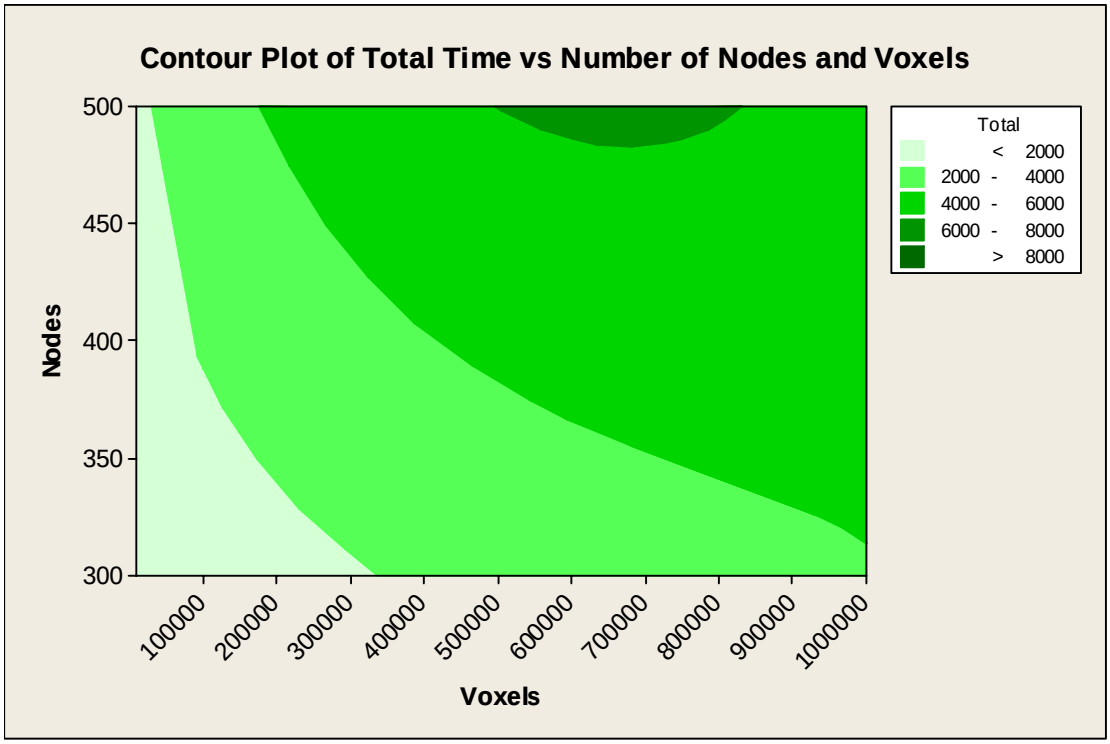


Figure 7.16: Contour Plot of Total Time for All Three Shapes

In contrast to torus, the contour plot for slab was filled with more or less horizontal bands (figure 7.15). Thus the performance of slab was more sensitive to number of nodes rather than number of voxels. This was because slab had a simple shape and thus the topology was not as complex as torus. However, slab being dense and uniform resulted in dense and bigger fronts in the FEM solver. Thus the contribution of time taken by FEM solver was more than contribution of time taken by other volumetric and surface data operations.

The effect of number of voxels and number of nodes has been combined for all three objects in figure 7.16. The contours enclose triangular regions with the axes. This means that in general, the number of voxels and number of elements both have positive correlation with the computation time and both have to be minimized to decrease computation time.

System clock was fairly effective in capturing the variation in responses. According to the Gauge R&R (Montgomery 2001) results listed in Appendix 10, the number of distinct categories is 8, which is quite favorable (>6). Gauge R&R is statistical analysis done on the measuring instrument to make sure that the variation due to limited resolution and precision of the gauge are negligible to the variations in the measured quantity. The Gauge R&R was done using two replications. The measured timings mimicked the distribution of time periods in actual data. Most of the time periods were shorter than 500 ms, few of them exceed 5000 ms. The program itself measured the time data and maintained a log for it. Hence there were no operators involved. The effect of operator and operator part interaction are therefore trivial.

7.3 Sample Output

Figures 7.17, 7.18 and 7.19 depict manipulation of sample objects. In figure 7.19, the constraint plane being inside the cylinder is not visible. It is shown in figure 7.19(b).

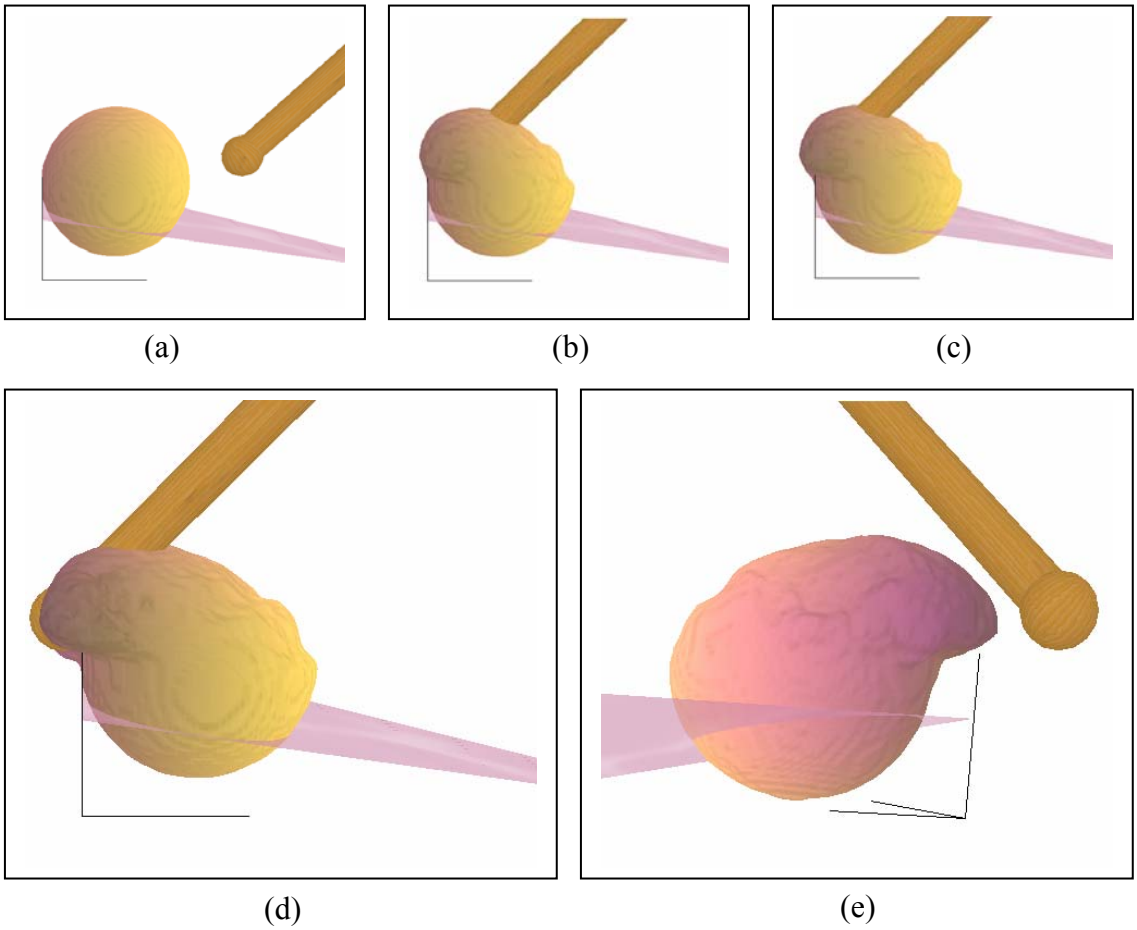
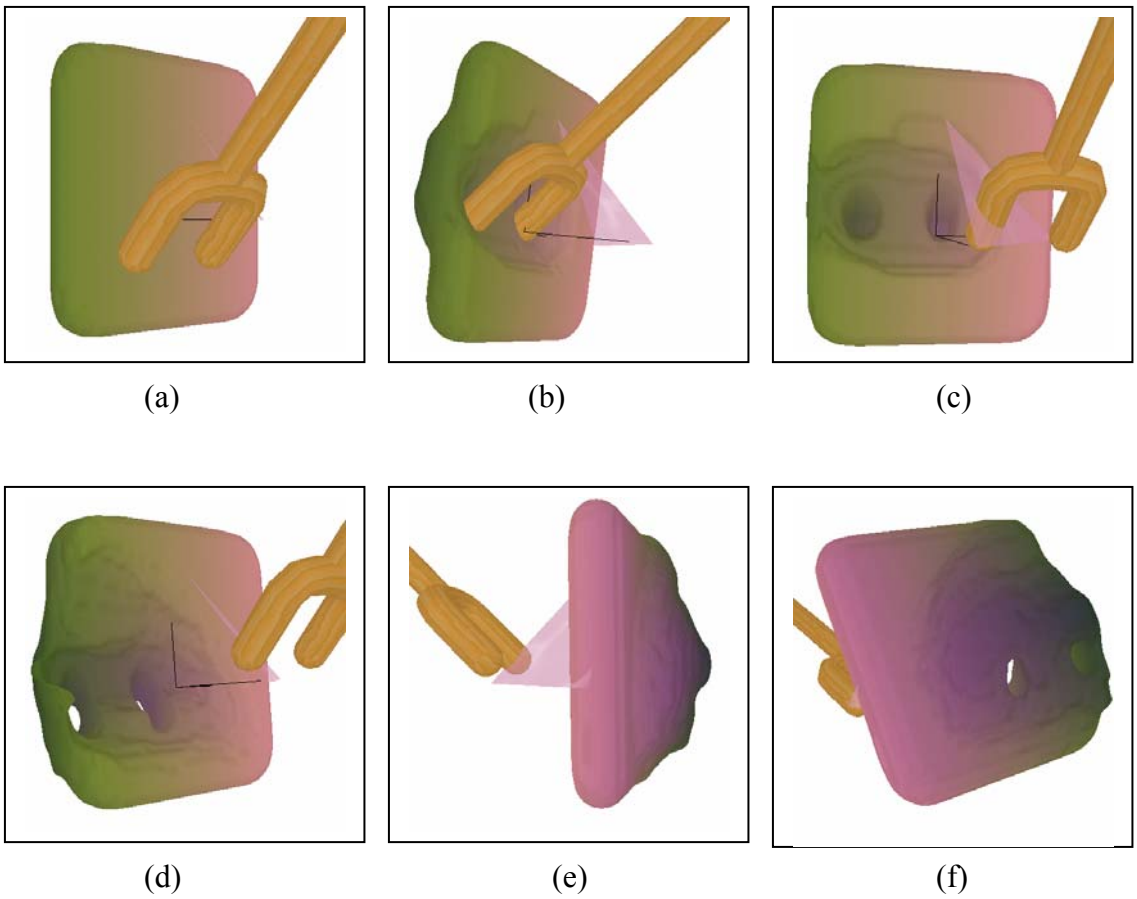


Figure 7.17: Deformation of sphere with a round tipped tool.



(a) (b) (c)

(d) (e) (f)

Figure 7.18: Deformation of slab with a multi-pronged tool.

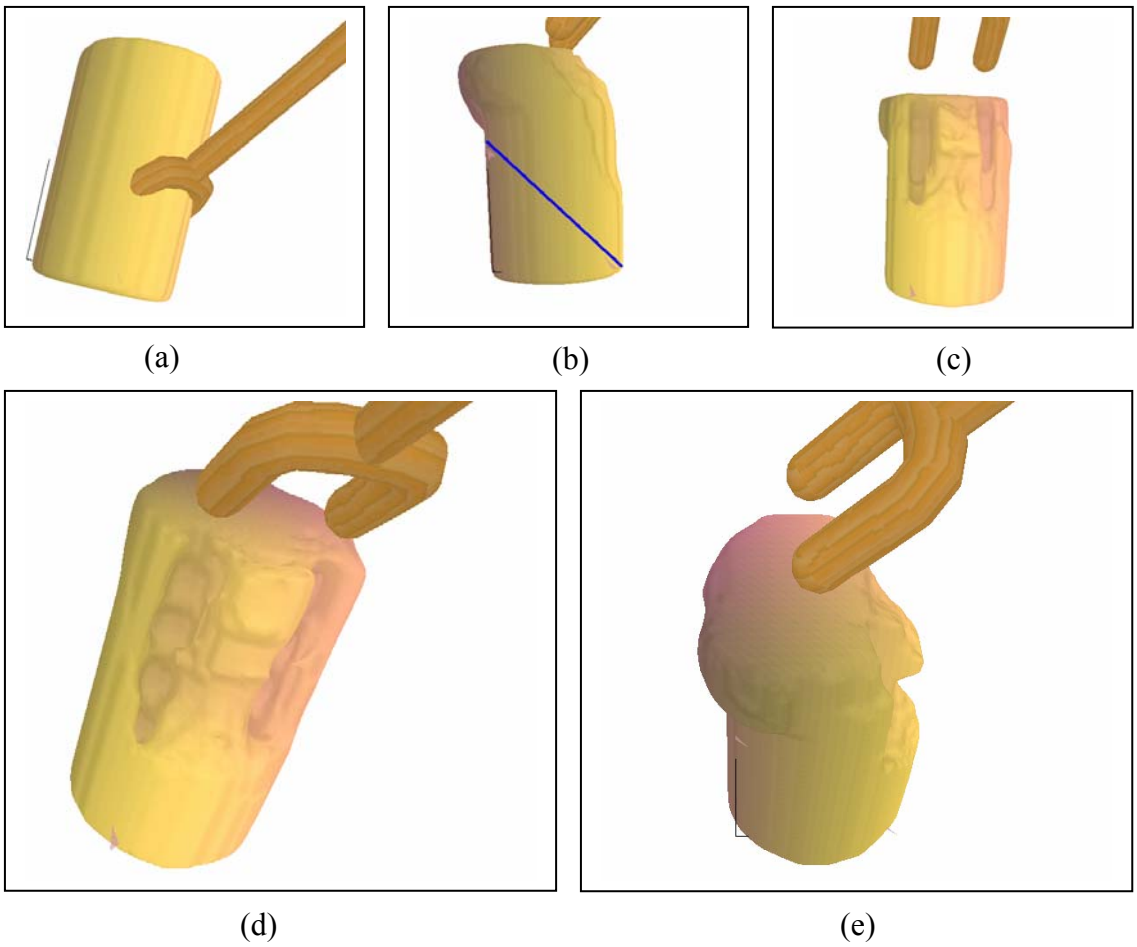


Figure 7.19: Deformation of a cylinder with a multi-pronged tool.

Chapter 8: Future Work and Conclusion

8.1 Future Work

Current work points to several directions in which more work can be pursued. For example, we have already implemented functionalities required to perform material cutting function. However joining two pieces of clay requires deformation which brings conformity on the interacting surface, followed by subsequent cohesion. This will be a challenging task to implement. It may require identification and removal of voids (if any) created during Boolean addition on pieces of clay. Currently, clay can have interior cavities or voids but tool cannot. In order to better deal with interior cavities and voids, an interior surface identification algorithm needs to be implemented. As an alternative to material joining process, a suction tool may pull the surface thus adding to the volume. Generating texture information can be very helpful not only in enhancing the appearance but also in implementation of new modification schemes. Currently the constraint plan is used to fix nodes to a plane. Optionally the constrained elements can be restricted to the periphery of the object. This will give a more realistic flow of material.

These were some ways to improve the capabilities of the sculpting system. The performance of current modeling system can also be improved by refining the FEM solution techniques. Standard solvers like MA62 (Duff and Scott 1999) that use BLAS functions can be used to improve the performance of FEM process. Thus more elements could be supported which would give better estimation of the deflection field. Multi-resolution structures may be used to reduce approximation error. Incorporation of

additional material properties like viscosity and surface tension would require integration (Gaussian quadrature**)**. This would allow the use of multi resolution grid for FEM.

The Surface Data produced is of too high resolution. A feature based method to reduce the resolution at specific points can be developed. Instead of modifying surface data, the voxels themselves can be made to follow the deflection field. It will require modification scheme to be looked at from a different perspective. Though seldom, undercuts do appear when the deformation is too large. This happens if the material is too soft. The problem can be solved by increasing the stiffness and number of elements. Increasing the stiffness means more iterations are required to achieve desired amount of deformation and increasing number of elements means more FEM calculations involved. Both call for higher computational capabilities.

Currently the force input is proportional to the intersection volume of tool and clay. Tool clay intersection is not desirable. This can be avoided by using a suitable force input device. Tool attached to a position handle will enable free tool movement and maneuverability compared to mouse and keyboard.

8.1 Conclusion

A new method of carrying out sculpting of digital clay has been successfully implemented. The method guarantees physically based handling of arbitrary tool-clay-environment interactions. The preliminary tests have been successfully carried out demonstrating that the proposed system gives reasonable performance.

Appendixes

Appendix 1 Material Properties (Representative Values Only)

Material	Tensile Strength (MPa)	Young's Modulus (GPa)	Density (kgm^{-3})
Soil Clay	<2 MPa	0.026-0.193	1000-2000
Epoxy	60 – 85	2.6 – 3.8	1,110 – 1,200
Steel	480 – 700	200	7,850

Table A1.1: Material Properties

Poission's ratio for soil: 0.45

Epoxy , % strain at failure : 1.5 – 8.0

Soil clay %strain at failure 12%

Appendix 2 FEM Formulation for Compressible Elastic Material

The strong form of Navier's Equation for elastic material flow is,

$$\Delta\tilde{u} + \frac{\nabla(\nabla \bullet \tilde{u})}{1-2\nu} + \frac{\tilde{F}}{\mu} = 0 \text{ (Strong Form)} \qquad \left(where\ \mu = \frac{E}{2(1+\nu)} \right) \text{(A2.1)}$$

Boundary Conditions are,

$\tilde{\sigma} \cdot \tilde{n} = T$ on Γ_1

$u = u_o$ on Γ_2

$u \in H^1(\Omega)^3$

The strong form can be expressed as individual equilibrium equations:

$$\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{yx}}{\partial y} + \frac{\partial \sigma_{zx}}{\partial z} + X_x = 0 \qquad \text{(A2.2)}$$

$$\frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{zy}}{\partial z} + X_y = 0 \qquad \text{(A2.3)}$$

$$\frac{\partial \sigma_{xz}}{\partial x} + \frac{\partial \sigma_{yz}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} + X_z = 0 \qquad \text{(A2.4)}$$

where,

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5-\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5-\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5-\nu \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{bmatrix} \text{(A2.5)}$$

(simply expressed as $\sigma = C\varepsilon$)

and

$$\varepsilon_{xx}=\frac{\partial u}{\partial x}\quad \varepsilon_{yy}=\frac{\partial v}{\partial x}\quad \varepsilon_{zz}=\frac{\partial w}{\partial x}\quad \gamma_{xy}=\frac{\partial v}{\partial x}+\frac{\partial u}{\partial y}\quad \gamma_{yz}=\frac{\partial w}{\partial y}+\frac{\partial v}{\partial z}\quad \gamma_{zx}=\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x}$$

For brick elements, the vector $\tilde{u}=\begin{bmatrix}u & v & w\end{bmatrix}$ can be expressed in terms of its values on nodes using 3D trilinear function space, $\frac{1}{8}(1\pm\xi)(1\pm\eta)(1\pm\zeta)\quad \xi,\eta,\zeta\in[-1,1]$.

This will give the expression for ε,

$$\varepsilon = B(\xi,\eta,\zeta)u_0 \tag{A2.6}$$

where u_0 is the augmented vector of *u* at all nodes.

Multiplying the equilibrium equations with *u*, *v* and *w* respectively and integrating will give the residual of each equation. Equating the sum of residuals to zero and replacing every instance of ε by equation (A2.6) will give the final equation,

$$Ku_0 = f_0 \tag{A2.7}$$

$$K=\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}B^T CB\left|\det J\right|d\xi\,d\eta\,d\zeta \text{ and } J=\begin{bmatrix}\frac{\partial x}{\partial\xi}&\frac{\partial y}{\partial\xi}&\frac{\partial z}{\partial\xi}\\ \frac{\partial x}{\partial\eta}&\frac{\partial y}{\partial\eta}&\frac{\partial z}{\partial\eta}\\ \frac{\partial x}{\partial\zeta}&\frac{\partial y}{\partial\zeta}&\frac{\partial z}{\partial\zeta}\end{bmatrix}$$

The term $\tilde{F}$ in strong form was force per unit volume and the term f_0 in the Galerkin form is augmented nodal force vector. Though skipped here for conciseness, the force term formulation should go hand in hand with rest of the formulation.

Appendix 3: FEM Formulation for Incompressible Material

The equation, $Ku = f$, derived for compressible material is replaced by:

$$\begin{bmatrix} K' & C \\ C^T & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix} \tag{A3.1}$$

$$K' = \int_{\Omega} B^T D B \, d\Omega \tag{A3.2}$$

$$C = \int_{\Omega} B^T m N_p \, d\Omega \tag{A3.3}$$

where,

p is the pressure field,

$$D = \frac{E}{9} \begin{bmatrix} 4 & -2 & -2 & 0 & 0 & 0 \\ -2 & 4 & -2 & 0 & 0 & 0 \\ -2 & -2 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix},$$

$m^T = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$.and N_p is original shape function.

Rest of the terms are same as in derivation of compressible elastic material.

Appendix 4 Derivation of Schur Compliment

Consider the following set of linear equations in which K^{11} is fully summed and is ready to be eliminated. The second row containing K^{22} is not fully assembled yet.

$$\begin{bmatrix} K^{11} & K^{12} \\ K^{21} & K^{22} \end{bmatrix} \begin{bmatrix} d^1 \\ d^2 \end{bmatrix} = \begin{bmatrix} f^1 \\ f^2 \end{bmatrix} \tag{A 4.1}$$

$$\begin{aligned} K^{11}d^1 + K^{12}d^2 &= f^1 \\ K^{21}d^1 + K^{22}d^2 &= f^2 \end{aligned} \tag{A 4.2}$$

Multiplying () by () and subtracting from () $\left[K^{21}\right]\left[K^{11}\right]^{-1}$,

$$\begin{aligned} K^{21}d^1 + K^{21}\left[K^{11}\right]^{-1}K^{12}d^2 &= K^{21}\left[K^{11}\right]^{-1}f^1 \\ \left(K^{22} - K^{21}\left[K^{11}\right]^{-1}K^{12}\right)d^2 &= f^2 - K^{21}\left[K^{11}\right]^{-1}f^1 \end{aligned} \tag{A 4.3}$$

$\left[K^{11}\right]^{-1}$ *can be obtained by LU decomposition*

We use Gauss elimination to eliminate the fully summed nodes in K^{11}, storing the dependency of d^1 on d^2. That means,

$$d^1 + \left[K^{11}\right]^{-1}K^{12}d^2 = \left[K^{11}\right]^{-1}f^1 \tag{A 4.4}$$

After Gauss elimination we directly obtain, $\left[K^{11}\right]^{-1}K^{12}$ and $\left[K^{11}\right]^{-1}f^1$. Let's call them $K^{12}{}'$ *and* $f^{1}{}'$ respectively.

Equation (A 4.3) becomes,

$$\left(K^{22} - K^{21}K^{12}{}'\right)d^2 = f^2 - K^{21}f^{1}{}' \tag{A 4.6}$$

Appendix 5 Maple Codes for Evaluating Explicit Stiffness Matrices

Derivation of Stiffness Matrix for Elastic Material

Cubic elements.
Order 2

```
> restart:
Order:=2:
with(LinearAlgebra):
VM:=VandermondeMatrix([x,y,z]):
genEqn:=expand(sum('VM[1,i]',i=1..Order)*sum('VM[2,i]',i=1..Order)*sum(
'VM[3,i]',i=1..Order)):
genEqn:=sort(%,[z,y,x],tdeg,ascending):
fields:=op(genEqn);
nodalEqnU:=0:
nodalEqnV:=0:
nodalEqnW:=0:
counter:=1:
for i in genEqn do
nodalEqnU:=nodalEqnU+aU[counter]*i:
nodalEqnV:=nodalEqnV+aV[counter]*i:
nodalEqnW:=nodalEqnW+aW[counter]*i:
counter:=counter+1:
end do:
nodalEqnU;
nodalEqnV;
nodalEqnW;
counter:=counter-1;

> counter:=1:
for i from 1 to Order do
        for j from 1 to Order do
                for k from 1 to Order do
                        if i=1 then x1:=0;
                        elif i=Order then x1:=L[1];
                        else x1:=L[1]/2;
                        end if;
                        if j=1 then y1:=0;
                        elif j=Order then y1:=L[1];
                        else y1:=L[1]/2;
                        end if;
                        if k=1 then z1:=0;
                        elif k=Order then z1:=L[1];
                        else z1:=L[1]/2;
                        end if;
cornerEqnU[counter]:=eval(nodalEqnU,{x=x1,y=y1,z=z1})=u[counter];
cornerEqnV[counter]:=eval(nodalEqnV,{x=x1,y=y1,z=z1})=v[counter];
cornerEqnW[counter]:=eval(nodalEqnW,{x=x1,y=y1,z=z1})=w[counter];
                        counter:=counter+1:
                end do;
        end do;
end do;
counter:=counter-1:
varsU:=[seq(aU[i],i=1..counter)]:
eqnsU:={seq(cornerEqnU[i],i=1..counter)}:
varsV:=[seq(aV[i],i=1..counter)]:
```

```
eqnsV:={seq(cornerEqnV[i],i=1..counter)}:
varsW:=[seq(aW[i],i=1..counter)]:
eqnsW:={seq(cornerEqnW[i],i=1..counter)}:
eqnMatU:=GenerateMatrix(eqnsU,varsU):
eqnMatV:=GenerateMatrix(eqnsV,varsV):
eqnMatW:=GenerateMatrix(eqnsW,varsW):
aSolnU:=LinearSolve(eqnMatU):
aSolnV:=LinearSolve(eqnMatV):
aSolnW:=LinearSolve(eqnMatW):
uSoln:=eval(nodalEqnU,{seq(aU[i]=aSolnU[i],i=1..counter)});
vSoln:=eval(nodalEqnV,{seq(aV[i]=aSolnV[i],i=1..counter)});
wSoln:=eval(nodalEqnW,{seq(aW[i]=aSolnW[i],i=1..counter)});

> du_dx:=diff(uSoln,x):
du_dy:=diff(uSoln,y):
du_dz:=diff(uSoln,z):
dv_dx:=diff(vSoln,x):
dv_dy:=diff(vSoln,y):
dv_dz:=diff(vSoln,z):
dw_dx:=diff(wSoln,x):
dw_dy:=diff(wSoln,y):
dw_dz:=diff(wSoln,z):
GaussianMat:=Matrix([du_dx,dv_dy,dw_dz,du_dy+dv_dx,dv_dz+dw_dy,du_dz+dw
_dx]):
```

$\varepsilon_x, \varepsilon_y, \varepsilon_z, \gamma_{xy}, \gamma_{yz}, \gamma_{zx}$

```
> interMat:=Matrix(3*counter,6):
for i from 1 to counter do
for j from 1 to 6 do
interMat[3*i-2,j]:=coeff(GaussianMat,u[i])[1,j]:
interMat[3*i-1,j]:=coeff(GaussianMat,v[i])[1,j]:
interMat[3*i,j]:=coeff(GaussianMat,w[i])[1,j]:
end do:
end do:
mat_E:=Matrix([[2,0,0,0,0,0],[0,2,0,0,0,0],[0,0,2,0,0,0],[0, 0, 0, 1,
0, 0],[0, 0, 0, 0, 1, 0],[0, 0, 0, 0, 0, 1]]):
ProdMat:=simplify(interMat.mat_E.Transpose(interMat)):

> finalMat:=Matrix(3*counter,3*counter);
for i from 1 to 3*counter do
for j from 1 to 3*counter do
finalMat[i,j]:=simplify(int(int(int(ProdMat[i,j],x=0..L[1]),y=0..L[1]),
z=0..L[1])/L[1]);
end do;
end do;

>with(codegen,C):
C(finalMat,filename = "f8.c"):
```

Derivation of Stiffness Matrix for Incompressible Elastic Material

Cubic elements.

Order 2

```
> restart:
Order:=2:
with(LinearAlgebra):
```

```
VM:=VandermondeMatrix([x,y,z]):
genEqn:=expand(sum('VM[1,i]',i=1..Order)*sum('VM[2,i]',i=1..Order)*sum(
'VM[3,i]',i=1..Order)):
genEqn:=sort(%,[z,y,x],tdeg,ascending):
fields:=op(genEqn);
nodalEqnU:=0:
nodalEqnV:=0:
nodalEqnW:=0:
counter:=1:
for i in genEqn do
nodalEqnU:=nodalEqnU+aU[counter]*i:
nodalEqnV:=nodalEqnV+aV[counter]*i:
nodalEqnW:=nodalEqnW+aW[counter]*i:
counter:=counter+1:
end do:
nodalEqnU;
nodalEqnV;
nodalEqnW;
counter:=counter-1;

> counter:=1:
for i from 1 to Order do
        for j from 1 to Order do
                for k from 1 to Order do
                        if i=1 then x1:=0;
                        elif i=Order then x1:=L[1];
                        else x1:=L[1]/2;
                        end if;
                        if j=1 then y1:=0;
                        elif j=Order then y1:=L[1];
                        else y1:=L[1]/2;
                        end if;
                        if k=1 then z1:=0;
                        elif k=Order then z1:=L[1];
                        else z1:=L[1]/2;
                        end if;

cornerEqnU[counter]:=eval(nodalEqnU,{x=x1,y=y1,z=z1})=u[counter];

cornerEqnV[counter]:=eval(nodalEqnV,{x=x1,y=y1,z=z1})=v[counter];

cornerEqnW[counter]:=eval(nodalEqnW,{x=x1,y=y1,z=z1})=w[counter];
                        counter:=counter+1:
                end do;
        end do;
end do;
counter:=counter-1:
varsU:=[seq(aU[i],i=1..counter)]:
eqnsU:={seq(cornerEqnU[i],i=1..counter)}:
varsV:=[seq(aV[i],i=1..counter)]:
eqnsV:={seq(cornerEqnV[i],i=1..counter)}:
varsW:=[seq(aW[i],i=1..counter)]:
eqnsW:={seq(cornerEqnW[i],i=1..counter)}:
eqnMatU:=GenerateMatrix(eqnsU,varsU):
eqnMatV:=GenerateMatrix(eqnsV,varsV):
eqnMatW:=GenerateMatrix(eqnsW,varsW):
aSolnU:=LinearSolve(eqnMatU):
```

```
aSolnV:=LinearSolve(eqnMatV):
aSolnW:=LinearSolve(eqnMatW):
uSoln:=eval(nodalEqnU,{seq(aU[i]=aSolnU[i],i=1..counter)});
vSoln:=eval(nodalEqnV,{seq(aV[i]=aSolnV[i],i=1..counter)});
wSoln:=eval(nodalEqnW,{seq(aW[i]=aSolnW[i],i=1..counter)});

> du_dx:=diff(uSoln,x):
du_dy:=diff(uSoln,y):
du_dz:=diff(uSoln,z):
dv_dx:=diff(vSoln,x):
dv_dy:=diff(vSoln,y):
dv_dz:=diff(vSoln,z):
dw_dx:=diff(wSoln,x):
dw_dy:=diff(wSoln,y):
dw_dz:=diff(wSoln,z):
GaussianMat:=Matrix([du_dx,dv_dy,dw_dz,du_dy+dv_dx,dv_dz+dw_dy,du_dz+dw
_dx]): ε_x, ε_y, ε_z, γ_xy, γ_yz, γ_zx

> interMat:=Matrix(3*counter,6):
for i from 1 to counter do
for j from 1 to 6 do
interMat[3*i-2,j]:=coeff(GaussianMat,u[i])[1,j]:
interMat[3*i-1,j]:=coeff(GaussianMat,v[i])[1,j]:
interMat[3*i,j]:=coeff(GaussianMat,w[i])[1,j]:
end do:
end do:
mat_E:=Matrix([[4/3,-2/3,-2/3,0,0,0],[-2/3,4/3,-2/3,0,0,0],[-2/3,-
2/3,4/3,0,0,0],[0, 0, 0, 1, 0, 0],[0, 0, 0, 0, 1, 0],[0, 0, 0, 0, 0,
1]]):
ProdMat:=simplify(interMat.mat_E.Transpose(interMat)):

> finalMat:=Matrix(3*counter,3*counter);
for i from 1 to 3*counter do
for j from 1 to 3*counter do
finalMat[i,j]:=simplify(int(int(int(ProdMat[i,j],x=0..L[1]),y=0..L[1]),
z=0..L[1])/L[1]);
end do;
end do;

> with(codegen,C):
C(finalMat,filename = "f8.c"):
(eigVal,eigVec):=Eigenvectors(finalMat);

> C(eigVec,filename = "f8eig.c"):
with(linalg):
augMat:=augment(transpose(eigVec),eigVal):

> evalMat:=simplify(eval(finalMat,v=1/2)):

> simplify((interMat));
GaussianMat1:=simplify(Matrix([du_dx+dv_dy+dw_dz])):
eqtn[1]:=simplify(L[1]^2*eval(diff(GaussianMat1[1,1],x),{y=0,z=0}));
eqtn[2]:=simplify(L[1]^2*eval(diff(GaussianMat1[1,1],y),{z=0,x=0}));
eqtn[3]:=simplify(L[1]^2*eval(diff(GaussianMat1[1,1],z),{x=0,y=0}));
eqtn[4]:=simplify(L[1]^3*eval(diff(diff(GaussianMat1[1,1],x),y),z=0));
```

```
eqtn[5]:=simplify(L[1]^3*eval(diff(diff(GaussianMat1[1,1],y),z),x=0));
eqtn[6]:=simplify(L[1]^3*eval(diff(diff(GaussianMat1[1,1],z),x),y=0));
eqtn[7]:=simplify(L[1]*eval(GaussianMat1[1,1],{x=0,y=0,z=0}));
interMat1:=Matrix(3*counter,7):
for i from 1 to 7 do
for j from 1 to counter do
interMat1[3*j-2,i]:=coeff(eqtn[i],u[j]):
interMat1[3*j-1,i]:=coeff(eqtn[i],v[j]):
interMat1[3*j,i]:=coeff(eqtn[i],w[j]):
end do:
end do:
Transpose(interMat1);

> interMat;

> CMat:=Transpose(Matrix([[(L[1]-x)*(L[1]-y)*(L[1]-z),(L[1]-x)*(L[1]-
y)*(L[1]-z),(L[1]-x)*(L[1]-y)*(L[1]-z),0,0,0],[(L[1]-x)*(L[1]-
y)*z,(L[1]-x)*(L[1]-y)*z,(L[1]-x)*(L[1]-y)*z,0,0,0],[(L[1]-x)*y*(L[1]-
z),(L[1]-x)*y*(L[1]-z),(L[1]-x)*y*(L[1]-z),0,0,0],[(L[1]-x)*y*z,(L[1]-
x)*y*z,(L[1]-x)*y*z,0,0,0],[x*(L[1]-y)*(L[1]-z),x*(L[1]-y)*(L[1]-
z),x*(L[1]-y)*(L[1]-z),0,0,0],[x*(L[1]-y)*z,x*(L[1]-y)*z,x*(L[1]-
y)*z,0,0,0],[x*y*(L[1]-z),x*y*(L[1]-z),x*y*(L[1]-
z),0,0,0],[x*y*z,x*y*z,x*y*z,0,0,0]]))/L[1]^3;
Transpose(interMat);
Cm:=Transpose(Vector([c1,c2,c3,c4,c5,c6,c7,c8]));
CMatrix:=interMat.CMat;
finalCMat:=Matrix(3*counter,8):
for i from 1 to 3*counter do
for j from 1 to 8 do
finalCMat[i,j]:=simplify(int(int(int(CMatrix[i,j],x=0..L[1]),y=0..L[1])
,z=0..L[1])/L[1]^2):
end do:
end do:

> Mmatrix:=Transpose(Matrix([1,1,1,0,0,0]));
Mmatrix.Transpose(Mmatrix);

> augmentedMat1:=augment(finalMat, finalCMat):
augmentedMat2:=augment(Transpose(finalCMat), Matrix(8,8)):
augmentedMat3:=Matrix(stackmatrix(augmentedMat1,augmentedMat2)):
Fvec:=Vector(32):
for i from 1 to 24 do
Fvec[i]:=F[i]:
end do:
for i from 25 to 32 do
Fvec[i]:=0:
end do:
augmentedMat:=Matrix(augment(augmentedMat3,Fvec));
```

$$\begin{bmatrix} \text{32 x 33 Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

Appendix 6 Binary Subdivision of B-Spline Curve and Surface Patch

We will start from generic equation of a B-Spline Curve:

$$P(u)=\sum_{i=0}^{n}P_i N_{i,k}(u) \quad (t_{k-1}\le u\le t_{n+1})$$

where

$$N_{i,k}(u)=\frac{(u-t_i)N_{i,k-1}(u)}{t_{i+k-1}-t_i}+\frac{(t_{i+k}-u)N_{i+1,k-1}(u)}{t_{i+k}-t_{i+1}}$$

$$N_{i,1}(u)=\begin{cases}1 & t_i\le u\le t_{i+1}\\ 0 & otherwise\end{cases} \qquad \text{(A6.1)}$$

A single B-Spline segment can be separated out which is influence by four control points (order = 4). The parametric equation of this segment is,

$$P_k(t)=\begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix}\begin{bmatrix}1 & 4 & 1 & 0\\ -3 & 0 & 3 & 0\\ 3 & -6 & 3 & 0\\ -1 & 3 & -3 & 1\end{bmatrix}\begin{bmatrix}Q_k\\ Q_{k+1}\\ Q_{k+2}\\ Q_{k+3}\end{bmatrix} \qquad \text{(A6.2)}$$

$$For\ k=0\ \ to\ \ n-3\ \ \ and\ \ \ 0\le t\le 1$$

This describes a single uniform cubic B-spline blending function. Binary division of this generic segment yields,

$$\begin{bmatrix}P_0^1\\ P_1^1\\ P_2^1\\ P_3^1\\ P_4^1\end{bmatrix}=\frac{1}{8}\begin{bmatrix}4 & 4 & 0 & 0\\ 1 & 6 & 1 & 0\\ 0 & 4 & 4 & 0\\ 0 & 1 & 6 & 1\\ 0 & 0 & 4 & 4\end{bmatrix}\begin{bmatrix}P_0^0\\ P_1^0\\ P_2^0\\ P_3^0\end{bmatrix} \qquad \text{(A6.3)}$$

Therefore we have the mask,

$$P_{edge}^1=\begin{pmatrix}P_{end\ point1}^0 & P_{end\ point2}^0\\ \frac{1}{2} & \frac{1}{2}\end{pmatrix} \qquad \text{(A6.4)}$$

$$P^1_{vertex} = \begin{pmatrix} P^0_{prev.vertex} & P^0_{vertex} & P^0_{next\,vertex} \\ \frac{1}{8} & \frac{6}{8} & \frac{1}{8} \end{pmatrix} \quad \text{(A6.5)}$$

A mask $P = \begin{pmatrix} P_0 & \dots & P_l \\ w_0 & \dots & w_l \end{pmatrix}$ with l terms is to be interpreted as,

$$P = \begin{bmatrix} P_0 & \dots & P_l \end{bmatrix} \begin{bmatrix} w_0 \\ \dots \\ w_l \end{bmatrix}, w_0 + \dots + w_l = 1$$

Similarly bicubic uniform B-Spline surface can be defined by:

$$P(u,v) = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} M\, P\, M^T \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix} where, M = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \quad \text{(A6.6)}$$

$0 \le u, v \le 1$

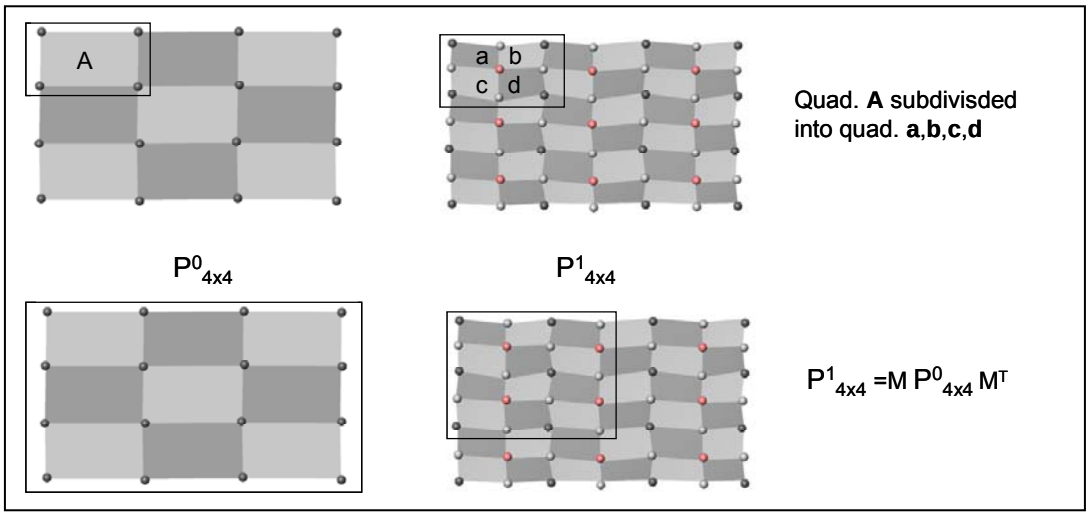


Figure A6.1: Catmull-Clark Subdivision

The bicubic uniform B-spline patch with 16 points when subdivided into subpatches can be used to express a similar set of 16 points in the subdivided surface (figure A6.1). If we focus on one of the four subpatches, say subpatch $a(0 \le u, v \le 1/2)$,

we can reparameterize this subpatch with $u' = u/2 \; and \; v' = v/2$. The subdivided patch *P'(u',v')* can be rewritten by substituting *u'* and *v'* in terms of *u* and *v*, resulting in,

$$\begin{bmatrix} P^1_{0,0} & P^1_{0,1} & P^1_{0,2} & P^1_{0,3} \\ P^1_{1,0} & P^1_{1,1} & P^1_{1,2} & P^1_{1,3} \\ P^1_{2,0} & P^1_{2,1} & P^1_{2,2} & P^1_{2,3} \\ P^1_{3,0} & P^1_{3,1} & P^1_{3,2} & P^1_{3,3} \end{bmatrix} = M_S \begin{bmatrix} P^0_{0,0} & P^0_{0,1} & P^0_{0,2} & P^0_{0,3} \\ P^0_{1,0} & P^0_{1,1} & P^0_{1,2} & P^0_{1,3} \\ P^0_{2,0} & P^0_{2,1} & P^0_{2,2} & P^0_{2,3} \\ P^0_{3,0} & P^0_{3,1} & P^0_{3,2} & P^0_{3,3} \end{bmatrix} M_S^T \quad \text{(A6.7)}$$

$$where, M_S = \frac{1}{8}\begin{bmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \end{bmatrix}$$

Appendix 7 Observations

Std. Order	Run Order	Object	Voxels	Nodes	Elements	Pre Process	FEM	Surf Gen	Smoothing	Regen	Total
		Input: Object type, number of Voxels, Nodes and Elements				*Output: Time in ms.*					*(ms)*
1	9	Slab	10000	300	186	0	980	20	10	40	1050
2	30	Slab	10000	500	328	10	3380	30	20	40	3480
3	18	Slab	100000	300	189	10	1010	100	90	200	1410
4	31	Slab	100000	500	326	10	7170	180	170	360	7890
5	35	Slab	1000000	300	188	150	1980	780	750	1630	5290
6	5	Slab	1000000	500	331	80	3710	450	380	1710	6330
7	26	Torus	10000	300	133	0	90	70	30	60	250
8	23	Torus	10000	500	250	0	270	110	40	70	490
9	22	Torus	100000	300	135	40	180	470	270	430	1390
10	12	Torus	100000	500	232	50	230	540	210	360	1390
11	34	Torus	1000000	300	134	120	100	1200	770	1550	3740
12	16	Torus	1000000	500	239	120	190	1710	930	1450	4400
13	29	Sphere	10000	300	239	0	280	20	10	20	330
14	1	Sphere	10000	500	399	0	650	30	20	30	730
15	33	Sphere	100000	300	253	10	130	100	80	120	440
16	17	Sphere	100000	500	418	10	510	110	70	100	800
17	6	Sphere	1000000	300	241	140	220	740	700	1160	2960
18	11	Sphere	1000000	500	401	150	910	900	590	960	3510
19	25	Slab	10000	300	186	0	980	20	20	40	1060
20	20	Slab	10000	500	328	0	3210	10	10	40	3270
21	4	Slab	100000	300	189	10	1010	100	90	210	1420
22	10	Slab	100000	500	326	10	6200	190	160	370	6930
23	8	Slab	1000000	300	188	150	1990	780	750	1640	5310
24	14	Slab	1000000	500	331	100	5540	880	800	1770	9090
25	7	Torus	10000	300	133	10	100	70	30	60	270
26	32	Torus	10000	500	250	10	200	80	30	60	380
27	21	Torus	100000	300	135	20	180	380	210	340	1130
28	28	Torus	100000	500	232	20	230	580	280	450	1560
29	24	Torus	1000000	300	134	90	90	990	590	1650	3410
30	2	Torus	1000000	500	239	120	190	1600	880	1440	4230
31	13	Sphere	10000	300	239	10	220	20	10	30	290
32	3	Sphere	10000	500	399	0	470	30	10	30	540
33	27	Sphere	100000	300	253	10	130	100	80	130	450
34	19	Sphere	100000	500	418	10	510	110	70	100	800
35	36	Sphere	1000000	300	232	110	230	640	520	1230	2730
36	15	Sphere	1000000	500	402	120	910	920	620	1000	3570

Table A7.1: Observations for Full Factorial Design

Frame Rate (frames per second, fps)

Std. Order	Object	Voxels	Nodes	Elements	fps
1	Slab	10000	300	186	3
2	Slab	10000	500	328	1
3	Slab	100000	300	189	6
4	Slab	100000	500	326	1
5	Slab	1000000	300	188	3
6	Slab	1000000	500	331	2
7	Torus	10000	300	133	12
8	Torus	10000	500	250	8
9	Torus	100000	300	135	6
10	Torus	100000	500	232	6
11	Torus	1000000	300	134	5
12	Torus	1000000	500	239	4
13	Sphere	10000	300	239	9
14	Sphere	10000	500	399	5
15	Sphere	100000	300	253	16
16	Sphere	100000	500	418	9
17	Sphere	1000000	300	241	6
18	Sphere	1000000	500	401	4
19	Slab	10000	300	186	3
20	Slab	10000	500	328	1
21	Slab	100000	300	189	6
22	Slab	100000	500	326	1
23	Slab	1000000	300	188	3
24	Slab	1000000	500	331	2
25	Torus	10000	300	133	11
26	Torus	10000	500	250	11
27	Torus	100000	300	135	7
28	Torus	100000	500	232	5
29	Torus	1000000	300	134	5
30	Torus	1000000	500	239	4
31	Sphere	10000	300	239	10
32	Sphere	10000	500	399	7
33	Sphere	100000	300	253	16
34	Sphere	100000	500	418	9
35	Sphere	1000000	300	232	6
36	Sphere	1000000	500	402	4

Table A7.2: Frame Rates

Appendix 8 ANOVA of Measured Time Periods

ANOVA: FEM Preprocessing ... versus Objects, Voxels, ...

```
Factor  Type   Levels  Values
Object  fixed       3  1, 2, 3
Voxels  fixed       3  10000,  100000, 1000000
Nodes   fixed       2  300, 500
```

Analysis of Variance for FEM Preprocessing

```
Source  DF      SS     MS       F      P
Object   2     206    103    0.38  0.686
Voxels   2   98739  49369  183.35  0.000
Nodes    1     100    100    0.37  0.547
Error   30    8078    269
Total   35  107122

S = 16.4091   R-Sq = 92.46%   R-Sq(adj) = 91.20%
```

Analysis of Variance for FEM Execution

```
Source  DF         SS        MS      F      P
Object   2   62939072  31469536  27.08  0.000
Voxels   2    2048706   1024353   0.88  0.425
Nodes    1   16782678  16782678  14.44  0.001
Error   30   34867067   1162236
Total   35  116637522

S = 1078.07   R-Sq = 70.11%   R-Sq(adj) = 65.12%
```

Analysis of Variance for Surface Generation

```
Source  DF       SS       MS      F      P
Object   2   967400   483700  16.86  0.000
Voxels   2  5645717  2822858  98.37  0.000
Nodes    1    96100    96100   3.35  0.077
Error   30   860883    28696
Total   35  7570100

S = 169.399   R-Sq = 88.63%   R-Sq(adj) = 86.73%
```

Analysis of Variance for Smoothing

```
Source  DF       SS       MS       F      P
Object   2    96706    48353    6.03  0.006
Voxels   2  3035089  1517544  189.31  0.000
Nodes    1     2178     2178    0.27  0.606
Error   30   240483     8016
Total   35  3374456

S = 89.5327   R-Sq = 92.87%   R-Sq(adj) = 91.69%
```

Analysis of Variance for Reconstruction

Source	DF	SS	MS	F	P
Object	2	526017	263008	14.81	0.000
Voxels	2	13374217	6687108	376.63	0.000
Nodes	1	1111	1111	0.06	0.804
Error	30	532656	17755		
Total	35	14434000			

S = 133.249 R-Sq = 96.31% R-Sq(adj) = 95.69%

Analysis of Variance for Total

Source	DF	SS	MS	F	P
Object	2	60424906	30212453	23.75	0.000
Voxels	2	78345206	39172603	30.80	0.000
Nodes	1	19448100	19448100	15.29	0.000
Error	30	38156678	1271889		
Total	35	196374889			

S = 1127.78 R-Sq = 80.57% R-Sq(adj) = 77.33%

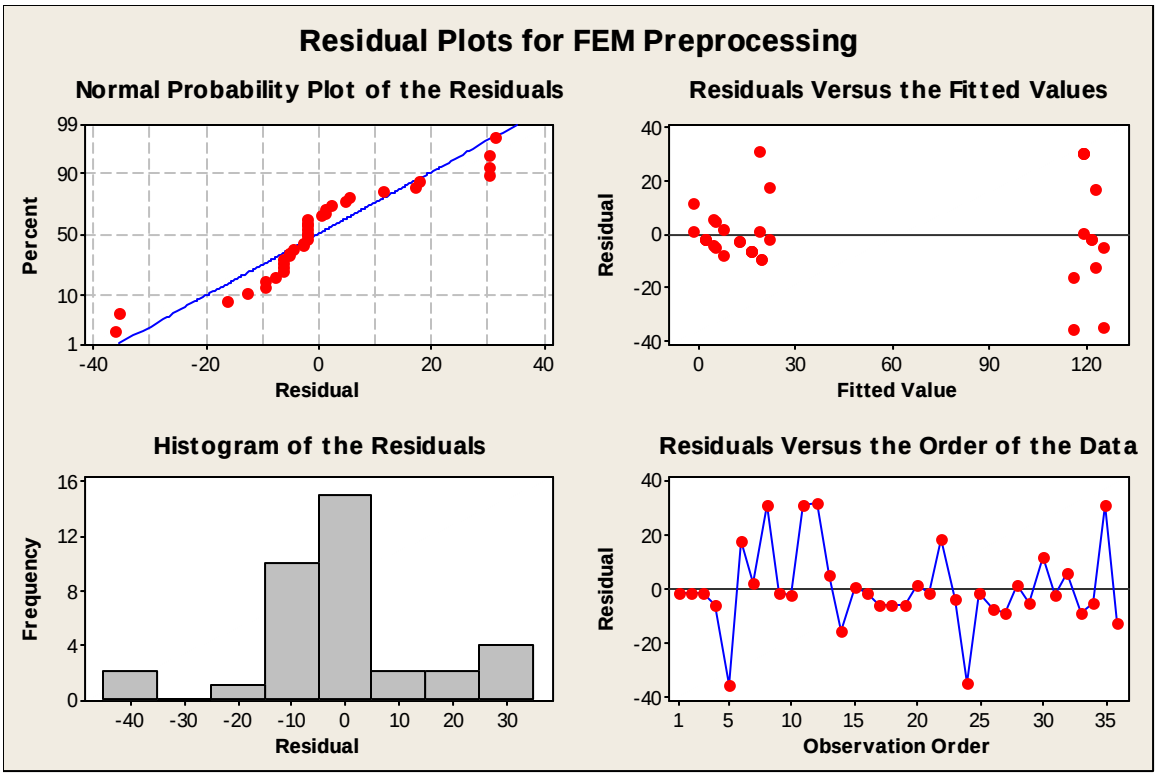


Figure A8.1: Residual Plots for FEM Preprocessing

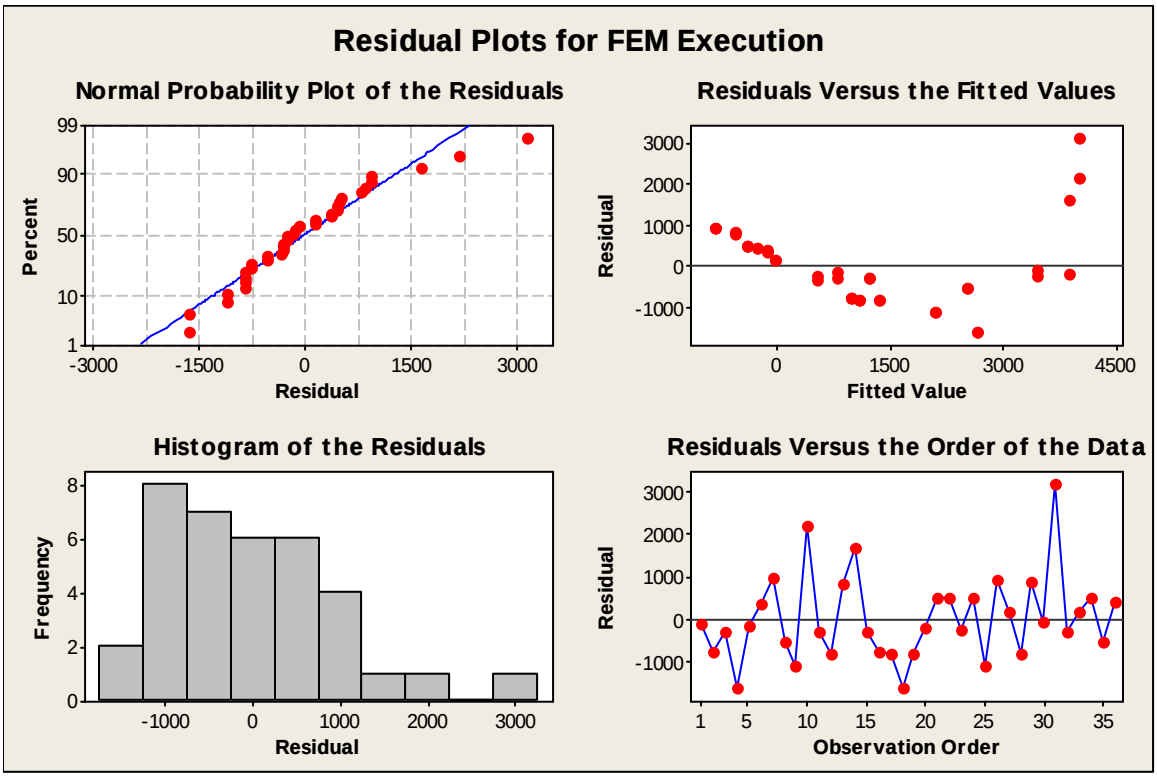


Figure A8.2: Residual Plots for FEM Execution

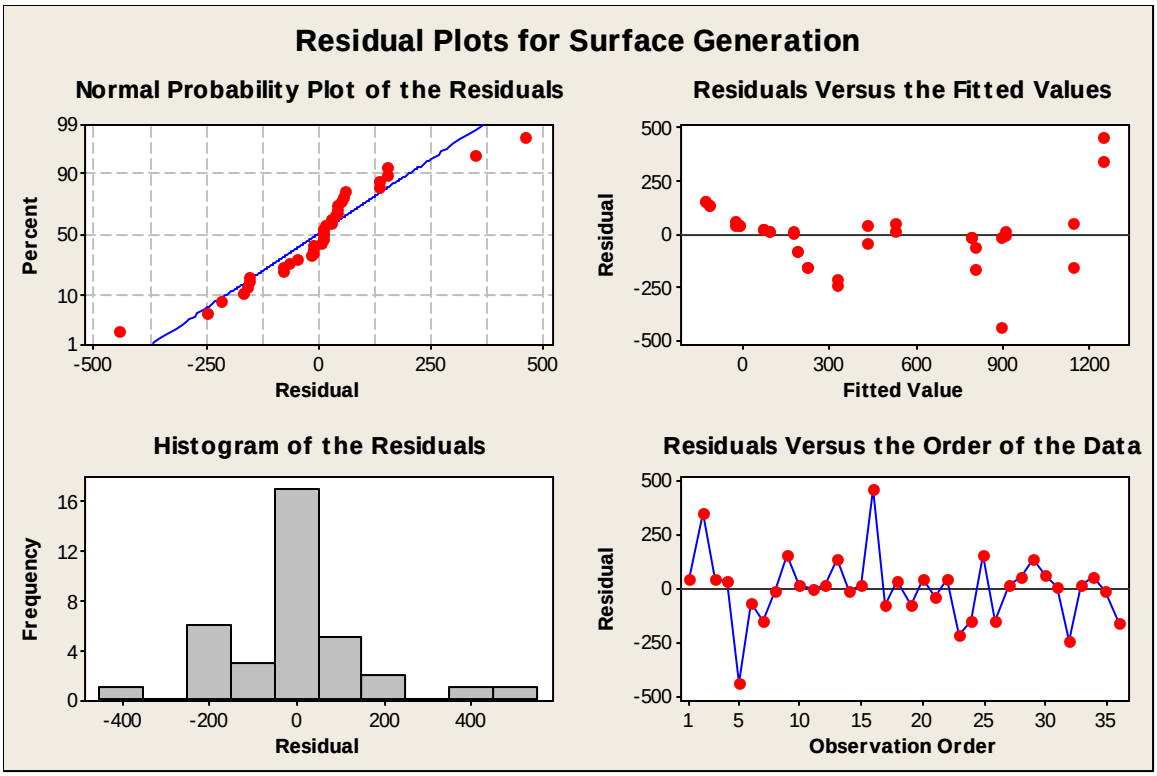


Figure A8.3: Residual Plots for Surface Generation

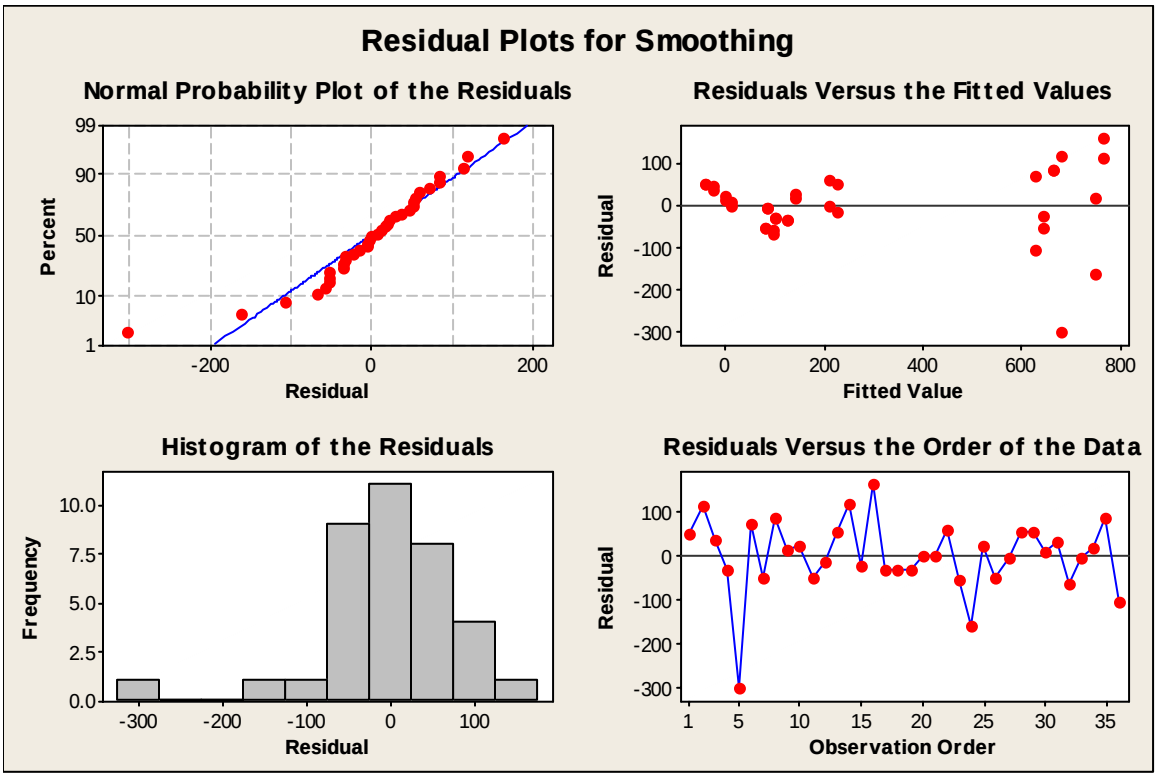


Figure A8.4: Residual Plots for Smoothing

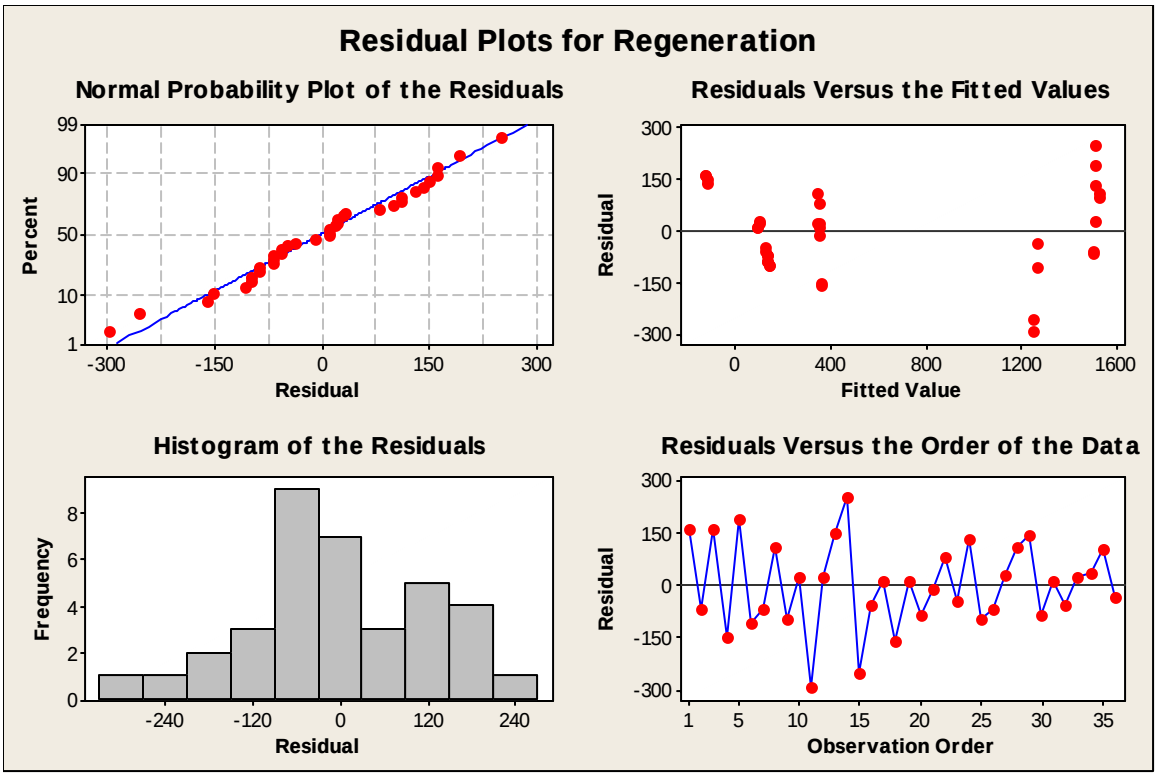


Figure A8.5: Residual Plots for Regeneration

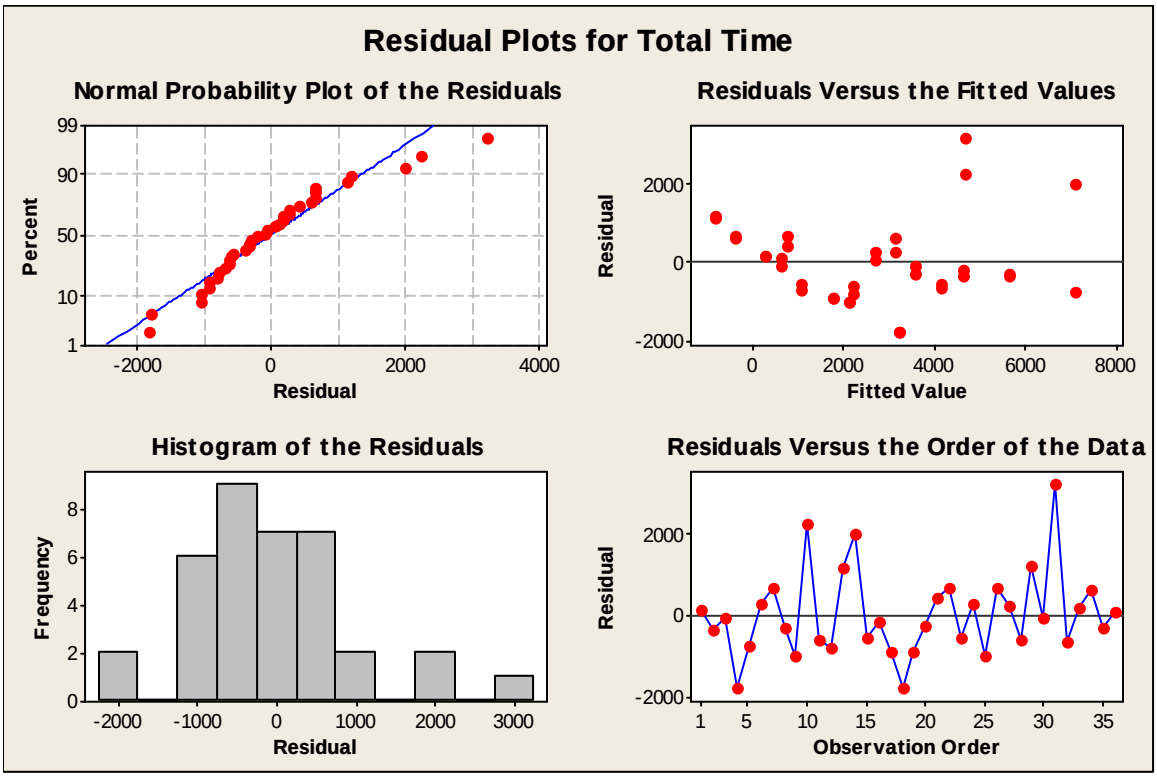


Figure A8.6: Residual Plots for Total Tome

Appendix 9: Analysis of Full Factorial Design

General Linear Model: FEM Preprocessing ... versus Object, Voxels, ..

```
Factor  Type   Levels  Values
Object  fixed       3  1, 2, 3
Voxels  fixed       3  10000, 100000, 1000000
Nodes   fixed       2  300, 500
```

Analysis of Variance for FEM Preprocessing, using Adjusted SS for Tests

```
Source               DF    Seq SS   Adj SS   Adj MS       F      P
Object                2     205.6    205.6    102.8    0.77  0.477
Voxels                2   98738.9  98738.9  49369.4  370.27  0.000
Nodes                 1     100.0    100.0    100.0    0.75  0.398
Object*Voxels         4    1777.8   1777.8    444.4    3.33  0.033
Object*Nodes          2    1050.0   1050.0    525.0    3.94  0.038
Voxels*Nodes          2     316.7    316.7    158.3    1.19  0.328
Object*Voxels*Nodes   4    2533.3   2533.3    633.3    4.75  0.009
Error                18    2400.0   2400.0    133.3
Total                35  107122.2
```

```
S = 11.5470   R-Sq = 97.76%   R-Sq(adj) = 95.64%
```

Analysis of Variance for FEM Execution, using Adjusted SS for Tests

```
Source               DF     Seq SS    Adj SS    Adj MS       F      P
Object                2   62939072  62939072  31469536  259.84  0.000
Voxels                2    2048706   2048706   1024353    8.46  0.003
Nodes                 1   16782678  16782678  16782678  138.57  0.000
Object*Voxels         4    4194444   4194444   1048611    8.66  0.000
Object*Nodes          2   21540239  21540239  10770119   88.93  0.000
Voxels*Nodes          2    2088206   2088206   1044103    8.62  0.002
Object*Voxels*Nodes   4    4864178   4864178   1216044   10.04  0.000
Error                18    2180000   2180000    121111
Total                35  116637522
```

```
S = 348.010   R-Sq = 98.13%   R-Sq(adj) = 96.37%
```

```
Unusual Observations for FEM Execution

Obs  FEM Execution      Fit  SE Fit  Residual  St Resid
  5        3710.00  4625.00  246.08   -915.00     -3.72 R
 14        5540.00  4625.00  246.08    915.00      3.72 R

R denotes an observation with a large standardized residual.
```

Analysis of Variance for Surface Generation, using Adjusted SS for Tests

```
Source               DF   Seq SS   Adj SS   Adj MS       F      P
Object                2   967400   967400   483700   66.31  0.000
Voxels                2  5645717  5645717  2822858  386.99  0.000
Nodes                 1    96100    96100    96100   13.17  0.002
Object*Voxels         4   424183   424183   106046   14.54  0.000
Object*Nodes          2    96200    96200    48100    6.59  0.007
Voxels*Nodes          2    69350    69350    34675    4.75  0.022
```

```
Object*Voxels*Nodes   4   139850   139850    34963    4.79  0.008
Error                18   131300   131300     7294
Total                35  7570100


S = 85.4075   R-Sq = 98.27%   R-Sq(adj) = 96.63%


Unusual Observations for Surface Generation

        Surface
Obs  Generation     Fit  SE Fit  Residual  St Resid
  5      450.00  665.00   60.39   -215.00     -3.56 R
 14      880.00  665.00   60.39    215.00      3.56 R

R denotes an observation with a large standardized residual.
```

Analysis of Variance for Smoothing, using Adjusted SS for Tests

```
Source               DF   Seq SS   Adj SS   Adj MS       F      P
Object                2    96706    96706    48353    6.86  0.006
Voxels                2  3035089  3035089  1517544  215.42  0.000
Nodes                 1     2178     2178     2178    0.31  0.585
Object*Voxels         4    33811    33811     8453    1.20  0.345
Object*Nodes          2    18672    18672     9336    1.33  0.290
Voxels*Nodes          2      689      689      344    0.05  0.952
Object*Voxels*Nodes   4    60511    60511    15128    2.15  0.117
Error                18   126800   126800     7044
Total                35  3374456


S = 83.9312   R-Sq = 96.24%   R-Sq(adj) = 92.69%


Unusual Observations for Smoothing

Obs  Smoothing      Fit  SE Fit  Residual  St Resid
  5    380.000  590.000  59.348  -210.000     -3.54 R
 14    800.000  590.000  59.348   210.000      3.54 R

R denotes an observation with a large standardized residual.
```

Analysis of Variance for Reconstruction, using Adjusted SS for Tests

```
Source               DF    Seq SS    Adj SS   Adj MS        F      P
Object                2    526017    526017   263008   255.90  0.000
Voxels                2  13374217  13374217  6687108  6506.38  0.000
Nodes                 1      1111      1111     1111     1.08  0.312
Object*Voxels         4    407317    407317   101829    99.08  0.000
Object*Nodes          2     46339     46339    23169    22.54  0.000
Voxels*Nodes          2     30339     30339    15169    14.76  0.000
Object*Voxels*Nodes   4     30161     30161     7540     7.34  0.001
Error                18     18500     18500     1028
Total                35  14434000


S = 32.0590   R-Sq = 99.87%   R-Sq(adj) = 99.75%


Unusual Observations for Reconstruction
```

Obs	Reconstruction	Fit	SE Fit	Residual	St Resid
24	1650.00	1600.00	22.67	50.00	2.21 R
34	1550.00	1600.00	22.67	-50.00	-2.21 R

R denotes an observation with a large standardized residual.

Analysis of Variance for Total Time, using Adjusted SS for Tests

Source	DF	Seq SS	Adj SS	Adj MS	F	P
Object	2	60424906	60424906	30212453	121.87	0.000
Voxels	2	78345206	78345206	39172603	158.01	0.000
Nodes	1	19448100	19448100	19448100	78.45	0.000
Object*Voxels	4	4672628	4672628	1168157	4.71	0.009
Object*Nodes	2	19949617	19949617	9974808	40.23	0.000
Voxels*Nodes	2	2497117	2497117	1248558	5.04	0.018
Object*Voxels*Nodes	4	6574817	6574817	1643704	6.63	0.002
Error	18	4462500	4462500	247917		
Total	35	196374889				

S = 497.912   R-Sq = 97.73%   R-Sq(adj) = 95.58%

Unusual Observations for Total Time

Obs	Total Time	Fit	SE Fit	Residual	St Resid
5	6330.00	7710.00	352.08	-1380.00	-3.92 R
14	9090.00	7710.00	352.08	1380.00	3.92 R

R denotes an observation with a large standardized residual.

Appendix 10: Gauge R&R

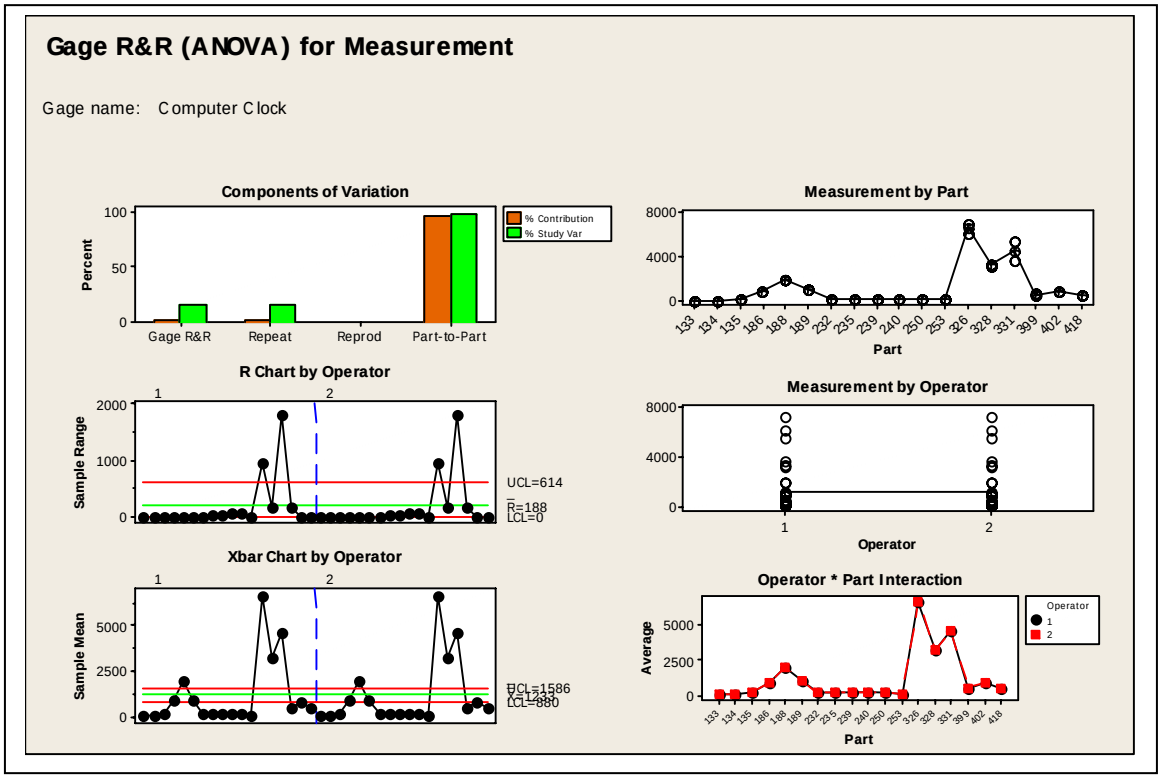


Figure A10.1 Gauge R&R for Time Measurement

Gage R&R Study - ANOVA Method

Two-Way ANOVA Table With Interaction

Source	DF	SS	MS	F	P
Part	17	228912644	13465450	*	*
Operator	1	0	0	*	*
Part * Operator	17	0	0	0	1.000
Repeatability	36	4362400	121178		
Total	71	233275044			

Alpha to remove interaction term = 0.25

Two-Way ANOVA Table Without Interaction

Source	DF	SS	MS	F	P
Part	17	228912644	13465450	163.595	0.000
Operator	1	0	0	0.000	1.000
Repeatability	53	4362400	82309		
Total	71	233275044			

Gage R&R

Source	VarComp	%Contribution (of VarComp)
Total Gage R&R	82309	2.40
Repeatability	82309	2.40
Reproducibility	0	0.00
Operator	0	0.00
Part-To-Part	3345785	97.60
Total Variation	3428094	100.00

Source	StdDev (SD)	Study Var (6 * SD)	%Study Var (%SV)
Total Gage R&R	286.90	1721.4	15.50
Repeatability	286.90	1721.4	15.50
Reproducibility	0.00	0.0	0.00
Operator	0.00	0.0	0.00
Part-To-Part	1829.15	10974.9	98.79
Total Variation	1851.51	11109.1	100.00

Number of Distinct Categories = 8

Gage R&R for Measurement

References

Baerentzen, A. (1998). "Octree-based volume sculpting", IEEE Visualization.

Barr, A. H. (1984). "Global and Local Deformations of Solid Primitives", Computer Graphics **17**(3): 21-30.

Bouma, W. and Vanecek, G. (1991). "Collision Detection and Analysis in a Physical Based Simulation", Proceedings of Eurographics Workshop on Animation and Simulation: 191-203.

Bro-Nielsen, M. (1995). "Modelling elasticity in solids using Active Cubes - Application to simulated operations", Proc. Computer Vision, Virtual Reality and Robotics in Medicine (CVRMed'95): 535-541.

Bro-Nielsen, M. and Cotin, S. (1996). "Real-time Volumetric Deformable Models for Surgery Simulation using Finite Elements and Condensation", Computer Graphics Forum, **15**(3): C57- C461.

Cabral, B., Cam, N. and Foran, J. (1994). "Accelerating Volume Reconstruction with 3D Texture Hardware", ACM Symposium on Volume Visualization: 91-98.

Catmull, E. and Clark, J. (1978). "Recursively generated B-spline surfaces on arbitrary topological meshes", Computer-Aided Design **10**(6): 350-355.

Chai, Y.H., Luecke, G.R. and Edwards, J.C. (1998). "Virtual Clay Modeling using the ISU Exoskeleton", Proceedings of IEEE Virtual Reality Annual Symposium (VRIAS'98), Los Alamitos California, IEEE Computer Society Press: 76–80.

Chen, H. and Fang, S. (1999a). "A volumetric approach to CSG modeling"*, Symposium on Solid Modeling and Applications, New York, ACM Press: 318–319.

Chen, H. and Fang, S. (1999b). "Fast voxelization of 3D synthetic objects", ACM Journal of Graphics Tools **3**(4):33–45.

Cline, H. E., Dumoulin, C. L., Lorensen, W. E., Hart, H. R. and Ludke, S. (1987). "3D Reconstruction of the Brain from Magnetic Resonance Images", Magnetic Resonance Imaging.

Cobb, E. (1984). Design of Sculptures Surfaces Using B-splines, PhD thesis, University of Utah.

Coquillart, S. (1990). "A Sculpting Tool for 3D Geometric Modeling", Proceedings of SIGGRAPH '90: 187–196.

Debunne, G., Desbrun, M., Cani, M., and Barr, A. H. (2001) "Dynamic real-time deformations using space & time adaptive sampling", Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01 ACM Press, New York, NY: 31-36.

Dewaele G. and Cani M.P. (2003). "Interactive global and local deformations for virtual clay", Pacific Graphics.

Doo, D. (1978). "A subdivision algorithm for smoothing down irregularly shaped polyhedrons", Proceedings of International Conference on Interactive Techniques in Computer Aided Design: 157-165.

Duff, I. S., Erisman, A. M., and Reid, J. K. (1989). Direct Methods for Sparse Matrices. Clarendon Press.

Duff, I. S. and Reid, J. K. (1983). "The Multifrontal Solution of Indefinite Sparse Symmetric Linear", ACM Trans. Math. Softw. **9**(3): 302-325.

Duff, I. S. and Scott, J. A. (1999). "A frontal code for the solution of sparse positive-definite symmetric systems arising from finite-element applications", ACM Trans. Math. Softw. **25**(4): 404-424.

Dyn, N., Gregory, J. A. and Levin D. A. (1990). "Butterfly Subdivision Scheme for Surface Interpolation with Tension Control", ACM Transactions on Graphics **9**(2): 160–169.

Ebert, S. David (1996). "Advanced Modeling Techniques for Computer Graphics", ACM Computing Surveys **28**(1).

Ferley, E., Cani, M.P. and Gascuel, J.D. (2000). "Practical Volumetric Sculpting", The Visual Computer, **16**(8):469–480.

Freeman, H. (1974). "Computer Processing of Line-Drawing Images", ACM Computing Survey **6**(1): 57-97.

Frisken, S.F., Perry, R.N., Rockwood, A.P. and Jones, T.R. (2000). "Adaptively sampled distance fields: A general representation of shape for computer graphics", Computer Graphics, Proceedings of SIGGRAPH'00, **34**(4): 249–254.

Galyean, T. and Hughe, J. (1991). "Sculpting: An Interactive Volumetric Modeling Techniques", Proceedings of SIGGRAPH'91: 267-274.

Gascuel, M. (1993). "An implicit formulation for precise contact modeling between flexible solids", Proceedings of the 20th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '93. ACM Press, New York, NY: 313-320.

Gibson, S. F. (1997). "3D chainmail: a fast algorithm for deforming volumetric objects", Proceedings of Symposium on interactive 3D Graphics (SI3D '97) ACM Press, New York, NY, 149-ff.

Gibson, S.F. (1998). "Using distance maps for smooth surface representation in sampled volumes", IEEE Symposium on Volume Visualization, Los Alamitos California, IEEE, IEEE Computer Society Press: 23–30.

Gourret, J. P., Magnenat-Thalmann, N. and Thalmann, D. (1989) "Simulation of object and human skin deformations in a grasping task", Proceedings of SIGGRAPH 1989, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings: 21-30.

Hirota, G., Maheshwari, R., and Lin, M. C. (1999). "Fast volume-preserving free form deformation using multi-level optimization", Proceedings of the Fifth ACM Symposium on Solid Modeling and Applications (SMA '99) ACM Press, New York, NY: 234-245.

Hsu, W.M., Hughes, J.F. and Kaufman, H. (1992). "Direct manipulation of free-form deformations", ACM Trans. Computer Graphics, **26**: 177–184.

Hughes, T. J. R. (1987). The finite element method: linear static and dynamic fnite element analysis. Prentice Hall.

Hui, Kin Chuen and Leung, H.C. (2002) "Virtual Sculpting and Deformable Volume Modelling", Proceedings of the Sixth International Conference on Information Visualisation (IV'02): 664-669.

Irons, B. (1970). "A Frontal Solution Program for Program for Finite Element Analysis", International Journal for Numerical Methods in Engineering, **2**(1): 5-32.

Kamerkar, A. and Kesavadas, T. (2004). "Touch Based Interactive NURBS Modeler using a Force/Position Input Glove", ASME 2004, Design and Technology Conferences, Salt Lake City, Utah.

Kameyama, K. (1997). “Virtual clay modeling system”, Symposium on Virtual Reality Software and Technology (VRST ’97), New York, ACM: 197–200.

Kaufman, Arie (1987) "Efficient algorithms for 3D scan-conversion of parametric curves, surfaces, and volumes", SIGGRAPH ’87, **21**: 171–179.

Kuester, F., Hamann, B., Joy, K.I. and Ma, K.L. (2002). "Virtual Clay Modeling using Adaptive Distance Fields", Proceedings of the 2002 International Conference on Imaging Science, Systems, and Technology (CISST 2002).

Kunii, T. (1994). “Research Issues in Modeling Complex Object Shapes”, IEEE Computer Graphics and Applications: 80-83.

Lee, Y.T. and Requicha, A.A.G. (1982). "Algorithms for computing the volume and other integral properties of solids", Communications of the ACM, 25(9): 635–650.

Loop, C. (1987) Smooth Subdivision Surfaces Based on Triangles. Master’s thesis, University of Utah, Department of Mathematics.

Loop, C. (2001). "Triangle mesh subdivision with bounded curvature and the convex hull property." Technical Report MSR-TR-2001-24, Microsoft Research.

Lorensen, W. and Cline, H. (1987). “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, SIGGRAPH'87: 163-169.

Mandal, C., Qin, Hong and Vemuri, Baba C. (1999). "A novel FEM-based dynamic framework for subdivision surfaces", Proceedings of the Fifth Symposium on Solid Modeling:191–202.

McDonnell, Kevin T. and Qin, Hong (2001). "FEM-based subdivision solids for dynamic and haptic interaction", Proceedings of the sixth ACM symposium on Solid modeling and applications: 312-313.

McDonnell, K.T., Qin, H. and Wlodarczyk, R.A. (2001). "Virtual Clay: A real-time sculpting system with haptic toolkits", ACM Symposium on Interactive 3D Graphics: 179–190.

Meagher, D. J. (1980). "Octree encoding: A new technique for the representation, manipulation, and display of arbitrary three-dimensional objects by computer", Technical Report IPL-TR-80-111. Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy NY.

Mizuno, S., Okada, M. and Toriwaki, J. (1998a). "Virtual Sculpting and Virtual Woodcut Printing", The Visual Computer **14**(2): 39–51.

Mizuno, S., Okouchi, T., Okada, M. and Toriwaki, J. (1998b). "Automatic Printing Block Generation from a 3D Model for Virtual Woodcut Printing", Proceedings of VSMM ’98: 134–139.

Mizuno, S., Okada, M. and Toriwaki, J. (1999a). "An interactive designing system with virtual sculpting and virtual woodcut printing", Proceedings of EUROGRAPHICS ’99: 183–193.

Mizuno, S., Okouchi, T., Okada, M. and Toriwaki, J. (1999b). "Multiworkpiece and multicolor virtual woodcut printing", Proceedings of VSMM ’99: 523–530.

Montani, C. and Scopigno, R. (1990). "Rendering Volumetric Data using the STICKS Representation Scheme", Computer Graphics **24**(5).

Montgomery, Douglas C. (2001). Design and analysis of experiments, 5th edition, John Wiley, New York.

Naylor, B. (1990). "Sculpt: An interactive solid modeling tool", Proceedings of Graphics Interface: 138–148.

Patra, A., Laszloffy, A. and Long, J. (2003). "Data Structures and Load Balancing for Parallel Adaptive hp Finite Element Methods", Computers and Mathematics with Applications **46**: 105-123.

Pentland, A. P. (1990), "Computational complexity versus simulated environments", Computer Graphics, **24**(2): 185-192.

Perng, Kuo-Luen, Wang, Wei-The, Flanagan, Mary (2001). "A Real-time 3D Virtual Sculpting Tool Based on Modified Marching Cubes", International Conference on Artificial Reality and Tele-Existence (ICAT): 64-72.

Perry, R. N. and Frisken, S. F. (2001). "Kizamu: A system for sculpting digital characters", Computer Graphics, Proceedings of SIGGRAPH' 01, **35**(4): 47–56.

Requicha, A.A.G. (1980). "Representations for Rigid Solids: Theory, Methods, and Systems", Computing Surveys **12**(4).

Sederberg, T. W. and Parry, S. R. (1986). "Free-Form Deformation of Solid Geometric Models", Proceedings of SIGGRAPH '86, Computer Graphics **20**(4): 151-159.

Shu, Renben, Zhou, Chen and Kankanhalli, Mohan S. (1995). "Adaptive marching cubes", The Visual Computer, **11**(4): 202-217.

Terzopoulos, D., And Fleischer, K. (1988a). "Deformable models", The Visual Computer **4**: 306-331.

Terzopoulos, D., And Fleischer, K. (1988b), "Modeling inelastic deformation: viscoelasticity, plasticity, fracture", Proceedings of SIGGRAPH 1988, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings: 269-278.

Terzopoulos, D., Platt, J., Barr, A., And Fleischer, K. (1987). "Elastically deformable models", SIGGRAPH 1987, ACMPress/ACMSIGGRAPH, Computer Graphics Proceedings: 205-214.

Wan, M., Kaufman, A. and Bryson, S. (1999). "High Performance Presence-Accelerated Ray Casting", Proceedings IEEE Visualization '99: 379-386.

Wang, S. and Kaufman, A.E. (1995). "Volume sculpting", Symposium on Interactive 3D Graphics, Monterey, CA, ACM: 151–156.

Wu, X., Downes, M. S., Goktekin, T. & Tendick, F. (2001). "Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes", Proceedings of EG 2001 Computer Graphics Forum **20**(3): 349–358.

[WWW_Exchange3d] http://www.exchange3d.com/new3d/gallery-album_3d-model-transport-vehicle-auto-luxury-american-car-4x4-GMC-Cadillac-Escalade-ESV-2004.html, 2005.

[WWW_Faeryforest] www.faeryforest.com/making_a_hand_using_an_armature.htm, 2005

[WWW_Lyonstudio] http://www.lyonstudio.com/Sculpting/Sculpting%20Textures.htm, 2004

[WWW_Nvidia] www.nvidia.com, 2005.

[WWW_Sensable_A] www.sensable.com, 2005

[WWW_Sensable_B] www.sensable.com/products/Rhino/index.asp, 2005