Real-Time Interactive Shape Editing Design Using OpenGL

Hla Myo Tun

Abstract:

Geometric design often requires the capability to manipulate flexible objects, including bending, twisting, compressing or stretching parts of the model or its entire geometry. The proposed approach comprises also shape deformations with multiple (real, auxiliary, and virtual) control points and constrained, directional, and anisotropic deformations. It allows a user to edit a given shape interactively and spontaneously.Our approach successfully suggests a sense of rigidity of the lattice, which is difficult in interactive shape editing approaches.Mathematically, the problem being addressed in this research work involves a Free-Form Deformation (FFD) of the features of the affine three dimensional geometry of the image. The algorithms primarily develop are projective geometry techniques, which form the basis for accurate deformation. Furthermore, these algorithms generally assume standardized images, i.e. no prior knowledge of either the shape, or its pose (position and orientation) with respect to the transformed coordinates. This project aims to develop a general framework for perform mesh deformation and editing in real-time. The prerequisites include C/C++, OpenGL and basic knowledge in differential geometry.

Index Terms: Interactive Shape Editing, Free-Form Deformation, OpenGL, Digital Geometric Modeling, Visual C++.

I. Introduction

In the early days of computer graphics, the shape editing techniques were very simple, ranging from static texture to simple particle systems using free form deformation. At that time, the only goal was to tell the researcher that the specific part was solid geometric model and did not prioritize much on the realism of the deformation techniques due to the restricted capabilities. With the fast development of free form deformation in recent years, realistic shape editing is possible and thus research in shape editing became more vigorous [1].

The general interactive shape editing design using free-form deformation is the simulation in geometric modeling in which each geometry value on every point in the region such as shape into a uniform value associated with the region. As a consequence, the simulation does not support very well geometric modeling aspects such as mesh boundary [2]. However, geometric modeling simulation will not be able to be handled by current computer as the computer has to evaluate every deform shape. The relatively new Free-Form Deformation Method (FFDM) is the interactive shape editing simulation which can handle shape editing simulation correctly as it stands on the middle between geometric modeling and interactive shape editing. FFDM is the extension of Sederberg Engine which originated from solid geometry modeling. In the Sederberg Engine, the simulation is done on discrete time step over discrete region, in which the set of deform image [3], [4], [5] and [6].

Dr. HlaMyoTun, Associate Professor and Head, Department of Electronic Engineering, Mandalay Technological University, Mandalay Region, Republic of the Union of Myanmar. hmyotun@myanmar.com.mm

II. Mathematical Model of Interactive Shape Editing

Mathematically, the FFD is defined in terms of a tensor product trivariate Bernstein polynomial. We instigate by arresting a local coordinate system on a parallel piped region, as shown in Fig.1. The red colour points are control points. Any point **ISD** has (s,t,u) coordinates in this system such that

$$\mathbf{ISD}= \mathbf{ISD}_0+s\mathbf{S}^* t\mathbf{T} * u\mathbf{U}. \quad (1)$$

The (s.t,u) coordinates of **ISD** can easily be originated using linear algebra. A vector solution is

$$s=\frac{\mathbf{T}\times\mathbf{U}\ (\mathbf{ISD}-\mathbf{ISD}_0)}{\mathbf{T}\times\mathbf{U}.\mathbf{S}} \quad (2)$$

$$t=\frac{\mathbf{S}\times\mathbf{U}\ (\mathbf{ISD}-\mathbf{ISD}_0)}{\mathbf{S}\times\mathbf{U}.\mathbf{T}} \quad (3)$$

$$u=\frac{\mathbf{S}\times\mathbf{T}\ (\mathbf{ISD}-\mathbf{ISD}_0)}{\mathbf{S}\times\mathbf{T}.\mathbf{U}} \quad (4)$$

Note that for any point interior to the parallel piped that $0<s<1$, $0<t<l$ and $0<u<l$. We next compel a grid of control points $\mathbf{CP}_{ijk}$ on the parallelpiped. These form t+l planes in the s direction, m+l planes in the **T** direction, and n+l planes in the u direction. The control points are designated by small white diamonds, and the red bars indicate the neighbouring control points. These points lie on a lattice, and their locations are defined

$$\mathbf{CP}_{ijk}=\mathbf{ISD}_0+\frac{i}{l}\mathbf{S}+\frac{j}{m}\mathbf{T}+\frac{k}{n}\mathbf{U} \quad (5)$$

where S, T and U represent the orthogonal axes of a bounding box (of size l, m and n) used to contain the object to be deformed. i, j and k are the evenly-spaced denominations of l, m and n used to denote the i-th Control Point along the l-axis, j-th Point along the m-axis, and so on. The deformation is stipulated by moving the $\mathbf{CP}_{ijk}$ from their undisplaced, lattieial positions. The deformation function is defined by a trivariate tensor product Bernstein polynomial. The deformed position $\mathbf{ISD}_{ffd}$ of an arbitrary point **ISD** is established by first computing its (s,t,u) coordinates from equation (1), and then evaluating the vector valued trivariate Bernstein polynomial:

$$\mathbf{ISD}_{ffd}=\sum_{i=0}^{l}\binom{l}{i}(1-s)^{l-i}s^{i}\left|\sum_{j=0}^{m}\binom{m}{j}(1-t)^{m-j}s^{j}\left|\sum_{k=0}^{n}\binom{n}{k}(1-s)^{n-k}s^{k}\right|\right| \quad (6)$$

where $\mathbf{ISD}_{ffd}$ is a vector containing the Cartesian coordinates of the displaced point, and where $\mathbf{CP}_{ijk}$ is a vector containing the Cartesian coordinates of the control point [1], and [7].

The principle assumptions, for simplification, in this implementation are:

- that it only allows movement of one Control Point at a time, and
- that it assumes the axes of the initial bounding box to be parallel to the three principal axes.

From $\mathbf{ISD}_{ffd}$, for each v calculated, the ijk-values are constant, as well as the lmn-values. The stu-values belong to each point and so are saved with it. The Control Points will vary along with their movement. However, from $\mathbf{CP}_{ijk}$, the three recursive summations are expensive to compute, with several repeated calculations, especially when applied over all vertices. Therefore, a work-around was devised. By assuming that one control point can only be manipulated at a time, the equation is simplified to become

$$v'=v+\left(\mathbf{CP}'_{ijk}-\mathbf{CP}_{ijk}\right)*\mathbf{ISD}_{ffd} \quad (7)$$

The control points $\mathbf{CP}_{ijk}$ are actually the coefficients of the Bernstein polynomial. As in the ease of Bezier curves and surface patches, there are momentous relationships between the deformation and the control point placement. Note that the 12 edges of the parallel piped are actually mapped into Bezier curves, defined by the control points which initially lie on the respective edges. Also, the six planar faces map into tensor product Bezier surface patches, defined by the control points which initially lie on the respective faces. This deformation could be originated in terms of other polynomial bases, such as tensor product D-splines or non-tensor product Bernstein polynomials

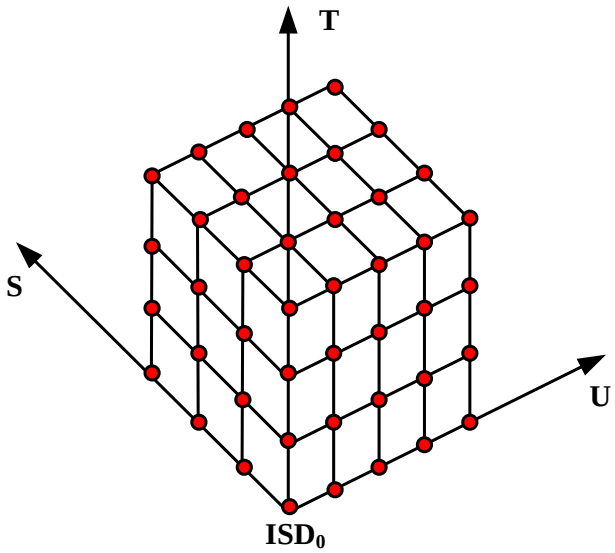


Fig.1. The s,t,uCoordinate System for Shape Editing

III. Shape Editing Domain

Although the purpose of this work is to establish FFD as a viable tool for solid modelling, we note that it can be applied to virtually any geometric model. Only the polygon vertices are transformed by the FFD, while maintaining the polygon connectivity. Deformation of polygonal data is conferred more thoroughly in [1]. The sphere and the plane could each be articulated in parametric equations, or in implicit equations. The FFD can be applied with equal validity to either representation. A very important characteristic of FFD is that a deformed parametric surface remnants a parametric surface. If the parametric surface is given by $x=f(\alpha,\beta)$, $y=g(\alpha,\beta)$ and $z=h(\alpha,\beta)$ and the FFD is given by $\mathbf{ISD}_{ffd} = \mathbf{ISD}(x, y, z)$), then the deformed parametric surface patch is given by $\mathbf{ISD}_{ffd}(\alpha,\beta) = x(f(\alpha,\beta), g(\alpha,\beta) , h(\alpha,\beta))$.

An important corollary to this is that parametric curves remain parametric under FFD. If one performs FFD in a CSG modelling environment only after all Boolean operations are executed, and the primitive surfaces are planes or quadrics, then all intersection curves would be parametric, concerning rational polynomials and possibly square roots. Quadrics and planes make exceptional primitives because they possess both implicit and parametric equations. The parametric equation enables rapid computation of points on the surface, and the implicit equation provides a simple point classification test - is a point inside, outside, or on the surface. To organize a point on a deformed quadric, one must first compute its s,t,u coordinates and substitute them into the implicit equation. The s,t,u coordinates can be created by subdividing the control point lattice, or by trivariate Newton iteration [8]. This inverse mapping necessitates significant computation, and can be a foundation of robustness problems, especially if the Jacobian of the FFD changes sign.

IV. Algorithm Pipeline of Interactive Shape Editing Design

In this project, the Sederberg Algorithm is used to simulate interactive shape editing design in 3D domain with an object is deformed to the Free-Form sense. The project consists of three main parts, Sederberg Engine, the M-file Reader, and the user input. The Sederberg Engine which is used to compute the control point on the image comprises the transformation from x,y,z, to s,t,u domain and reverse transformation step, and the output is the transformation image by Free-Form sense which is used in the real-time interactive shape editing design. The real-time interactive shape editing design simulation consists of Sederberg Engine and M-file reader step, which in every time step the image will deform to its neighboring region in the Sederberg Engine step and will be advocated according to the control point allocation in the M-file reader step. The user can interact both with the Sederberg Engine and the real-time interactive shape editing simulation. For example, the user can add external images which are used as input to the control point allocation step in the Sederberg Engine step. Another source of the external images is the movement of the object as the object movement can disturb

the shape editing. When the user transforms the image, the specified location and orientation of the image boundary will be updated and will be used as input to the control point allocation step in Sederberg Engine and to the real-time interactive shape editing simulation. The information from the image boundary is necessary since the real-time interactive shape editing cannot move past through the image boundary. The information from updating the Free-Form sense such as the MATLAB m file and the boundary information can be obtained by doing the deformation process since the simulation is done on the grid consisting of geometry domain. The overview of the program flow is shown on the Fig.2 on the same page.The simulation is implemented in C++ language by using Microsoft Visual C++ 2008 Express Edition. As for the graphics API we use OpenGL together with Cg for the GPU programming.

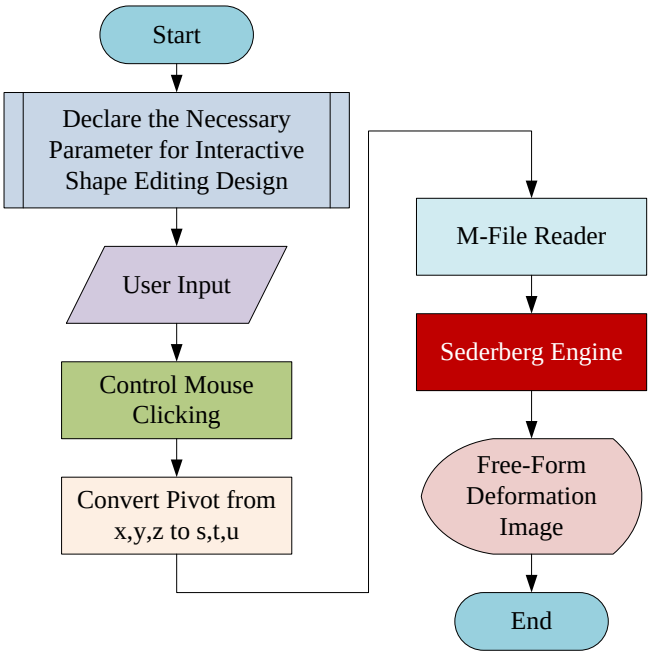


Figure.2. Program Flow

V. Interactive Shape Editing Method

The Sederberg Engine we choose to implement is the Free-Form Deformation which is the three dimensional shape editing image. The reason we chose this Engine was that the Sederberg has too high deformation for solid modeling which may cause the simulation to be a high actuate. On the other hand, the Sederberg might give a very accurate result; however, due to large number of deformation will make the implementation become very high. The Sederberg Engine section as shown on the above Fig.2 consists of convert pivot x,y,z. The result of the Sederberg Engine is the evolution of the shape editing which will be used to advent the Free-Form Deformation in the sections following the shape editing method.

A. M-File Reader

In the M-file reader step the three dimensional image is read by MATLAB m file to the same original image. Before the simulation is started with the M-filer reader, this 3D image is initialized with the graphic plane, which is the deformation image for Sederberg Engine. As the result, the total deformed images are evaluated when read from M file reader. In the M-file reader the program visits each original 3D image and transforms the deformation images. The M-file reader is given on the following Algorithm.1.

Algorithm.1 M-File Reader.

```
:MFileReader = MFileReader() //using namespace standard
:MFileReader = ObtainNextVertex()//Obtains next vertex
: indexCh1←indexCh1+ 7 //Move 7 character right to the 44
: vector's number.
:  indexCh2←find(indexCh1)
:  array1← 0 // Empty the array
:  array1←new char[array Size]
:// copy the value out
: vertexNum←atoi(array1)
:VertexPt→vertPt
: vertPt.ptNumber= vertexNum//assign number of e vertex
:  indexCh1←indexCh2 + 1
:// Obtain Coords
: for i = 0 to 2
:indexCh1←indexCh1+ 1//Move 1 character right to the
:// next coordinate (skips spaces)
:   array1← 0 // Empty the array
:   array Size←indexCh2 - indexCh1
:// copy the value out
:   switch (i)
: case 0→ vertex point position x = vertex Coordinate
: case 1→ vertex point position y = vertex Coordinate
:case 2→ vertex point position z = vertex Coordinate
:indexCh1 = indexCh2; // Update start pointer with end
: indexCh1←indexCh1+9 // Obtain normal
: end
:for i = 0 to 2
:indexCh1←indexCh1+ 1//Move1 character right to the next
:array1← 0// Empty the array
:array Size←indexCh2 - indexCh1
:// copy the value out
:switch (i)
:case 0→ vertex point normal x = vertex Coordinate
:case 1→ vertex point normal y = vertex Coordinate
:case 2→ vertex point normal z = vertex Coordinate
:   indexCh1←indexCh2; // Update start pointer with end
:// Save the vertex (Account for non-zero start)
:// Obtains next face in the file
: end
:   MFileReader=ObtainNextFace()
:   indexCh1←find(cstrFace)
: if indexCh1≠ npos string // Found a face
:   indexCh1←indexCh1+5//Move 5 character right to the
:// face's number
:   indexCh2←find(indexCh1)
:array1← 0// Empty the array
:array Size←indexCh2 - indexCh1
: end // copy the value out
: thisFace.faceNumber=vertexNum//assign number of e
:// vertex
:   indexCh1←indexCh2 + 1 // "Face X  _<-"
:   for i = 0 to 2 // Obtain coordinates
:   indexCh2←find(indexCh1)
:   if indexCh2= npos string // Finished
:   indexCh2 = line.length()
:array1← 0// Empty the array
:array Size = indexCh2 - indexCh1
: end // copy the value out
:   vertexPts + vertexNum - 1
:  indexCh1←indexCh2 // Update start pnter with end
:// Executes the read and obtaining procedure
:   MFileReader=Read(string filename, MObject *mObj)
:   ifstream=readFile //MObject mObj;
:   throw 0 //readFile.close();
:   if indexCh1≠ npos string // Found a vertex
:   mObj.ObjFaces = faces
:   mObj.ObjVertexPts = vertexPts
:   vertexPts.clear()
:   readFile close()//return mObj
```

B. Sederberg Engine

In the Sederberg Engine step in every vertex is transferred to the same direction vertex in the neighboring control point

which is in the direction of the vertex. Before the simulation is started with the Sederberg Engine, every vertex is initialized with the equilibrium value, which is the domain of s,t,u corresponding to the domain of x,y,z. As the result, the total particle distribution of each vertex in the beginning of the vertex is $\mathbf{ISD_0}$. In the Sederberg Engine the program visits each vertex and copies the value of distribution function of every vertex. The Sederberg Engine is given on the following Algorithm .2.

Algorithm.2 Sederberg Engine.

```
:SederbergEngine
: Convert xyz To stu(Vector xyz)// Assign
: Results←xyz.x-origin.x,xyz.y-origin.y,xyz.z-origin.z
: Convert stu To xyz(Vector stu)// Assign
: Results←stu.x+origin.x,stu.y+origin.y,stu.z+origin.z
: Copy From Vertexes(vector<VertexPt>inputVectors)
: stu point ← 0
: for i = 0 to input Vectors size
:    stu Points push back(SederbergAlgorithm)
: end // end for
: end // Copy From Vertexes
: Copy To Vertexes(vector<VertexPt> inputVectors)
: vector<VertexPt> = result
: for i = 0 to input Vectors size
:    stu Points push back(SederbergAlgorithm)
: end // end for
: end // Copy To Vertexes
: Initialize(Vector stuOrigin, Vector lmnInput)
: // float sGran, float tGran, float uGran
: origin = stu_Origin
: lmn = lmn_Input
: UpdatePoint(Vector point,Vector originalCP,Vector
:   changedCP,Vector ijk)
: Vector stu
: stu.x = point.x - origin.x
: stu.y = point.y - origin.y
: stu.z = point.z - origin.z
: ComputeWeight(stu, lmn, ijk)
: point.x=point.x+(changedCP.x-originalCP.x)*weight
: point.y=point.y+(changedCP.y-originalCP.y)*weight
: point.z=point.z+(changedCP.z-originalCP.z)*weight
: end // end UpdatePoint()
: Factorial(int number)
: if number is less than or equal 1  then return 1
:    temp = number * Factorial(number - 1)
: end // end if
:  return
:   end // end of Factorial
: Factorial(int upper, int lower)
:    int temp = Factorial(upper)
:    temp = temp / Factorial(lower)
:    temp = temp / Factorial(upper - lower)
: return temp
:    end // end of Factorial
: ComputeWeight(Vector stu, Vector lmn, Vector ijk)
: result=Factorial((int)lmn.x(int)ijk.x)*pow(stu.x,ijk.x)*
: pow(1 - stu.x, lmn.x - ijk.x);
: result*=Factorial((int)lmn.y,(int)ijk.y)*pow(stu.y,
:ijk.y)*pow(1 - stu.y, lmn.y - ijk.y);
: result*= Factorial((int)lmn.z,(int)ijk.z)*pow(stu.z,
:ijk.z)*pow(1 - stu.z, lmn.z - ijk.z);
: end // end SederbergAlgorithm
```

Thus, the applied algorithm is as follows:

Step 1. The vertices are converted in bulk to a corresponding array of stu-space coordinates. For simplicity, this project assumes or forces the axes of the stu-space as the same as that of xyz-space, though the origin need not be similar. This eliminates costly rotational translation per vertex into the stu-space.

Step 2. The Control Points (CPs) are initialized from the object's stu-space bounding box (in this case, each boundary line divided into four sections of five CPs). They are saved in an array.

Step 3. The engine is initialized: The stu-vertices are passed into the engine for storage, as well as the co-ordinates for its origin.

Step 4. The image is displayed with the CPs, and the user selects and moves one point.

Step 5. The corresponding CP's new and initial point are passed into the engine, which calculates and re-saves the new stu-coordinates of each vertex. These vertices are then converted back *en masse* into the xyz-space.

Step 6. The newly-modified vertices and CP are refreshed onscreen.

Step 7. The step 4-6 is repeated as necessary.

VI. Implementation

In this section, we present some results of the FFD simulation using the implementation discussed in previous section.We conducted six simulations with different settings to in order test certain features of the simulation. The first simulation was the bunny simulation. The Free-Form Deformation essentially does not depend on the any m files. However, if the m file input is active, the FFD will not only shapes and sizes of image, but also get campaigner according to the deformation which will make it difficult to observe the interactive shape editing.The control points for deforming the image are specified by identifying the two main steps for point grid and point selection.

A. Point Grid

For our Interactive Shape Editing with Free Form Deformation program, we will use a control mesh derived from the bounding box of the model. This mesh will be a 3D grid of 4x4x4 which gives us 64 control points.

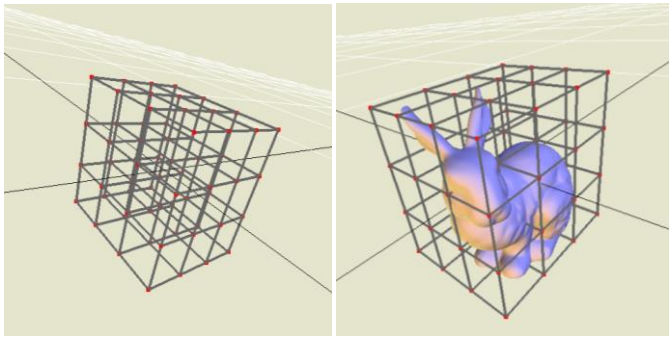


Fig.3.Point Grid

B. Point Selection

OpenGL provides a mechanism to select objects in a 3D scene. This is done by making use of the **Name Stack**. The idea is simple, we can enter *selection mode* (GL_SELECT) and render a small area around the mouse. For each selectable object that we render, we push and pop a unique name (the name is actually an integer number) into the name stack. When each selectable object is rendered, if it intersects the viewing volume a new *hit record* (with its corresponding depth information) is created. After we go back to the normal rendering mode, we can retrieve the *hit records* and select the hitted object that is closer to the camera. The function calls for the name stack are ignored if not in *selection mode*. This means that we could use a single rendering function with the name stack calls inside, but since only the control points are selectable in our program, we chose to enter selection mode and only render these points (using their position in the array

as the name) when the user clicks on the screen. The user will then be able to move the selected point around while the mouse button is still pressed. Since the normal rendering and the rendering for the selection function will be different, it is very important to use the same projection matrix, otherwise the selection would not work properly since the points would be projected on a different position in the projection plane.

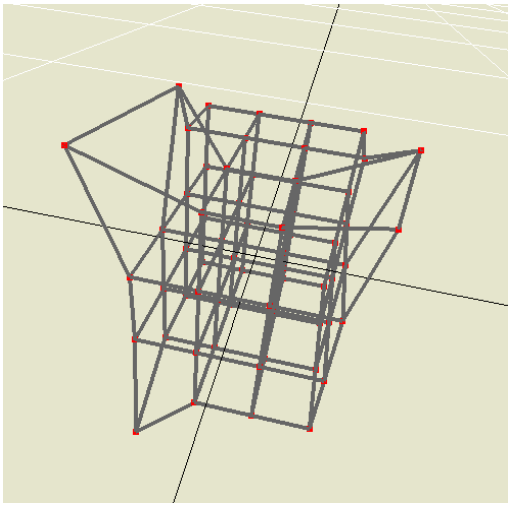
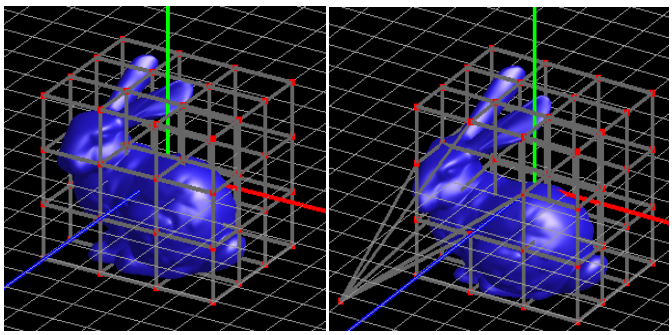


Fig.4. Point Selection

VII. Experimental Results

A. Experimental Results for Bunny Image

The following Fig.5(a) shows the screenshot of the simulation with the Pre-Deformed bunny image. The screenshots were taken in 4×4×4 frame. We can see from the bunny image that lies inside the Sederberg boundary for Free-Form Deformation process. The screenshot results of post-deformed bunny image with (s=2, t=1, u=0) is illustrated in Fig.5(b).

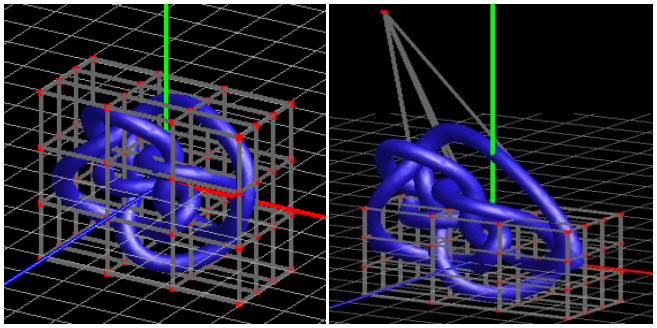


(a) (b)

Fig.5 (a) Pre-Deformed Bunny Image (b) Post-Deformed Bunny Image with (s=2, t=1, u=0)

B. Experimental Results for Knot Image

Fig.6 (a) shows the screenshot of the simulation with the Pre-Deformed knot image. The screenshots were also caught in 4×4×4 frame. We can perceive from the knot image that deceits inside the Sederberg boundary for Free-Form Deformation development. The screenshot results of post-deformed knot image with (s=2, t=3, u=2) is illustrated in Fig.6 (b).

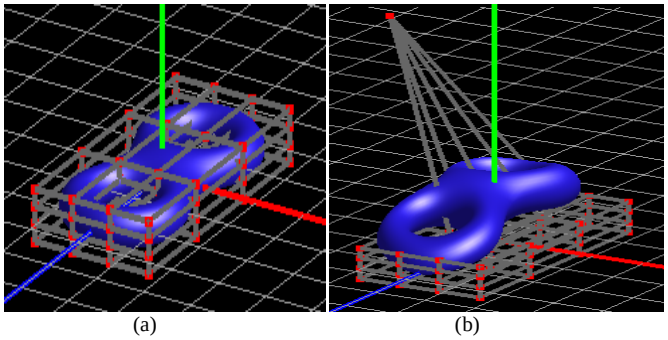


(a) (b)

Fig.6 (a) Pre-Deformed Knot Image (b) Post-Deformed Knot Image with (s=2, t=3, u=2)

C. Experimental Results for Eight Image

The following Fig.7 (a) demonstrates the screenshot of the simulation with the Pre-Deformed eight image. The screenshots were held in 4×4×4 frame. We can see from the eight image that deceptions within the Sederberg boundary for Free-Form Deformation procedure.The screenshot results of post-deformed eight image with (s=2, t=2, u=2) is illustrated in Fig.7 (b).

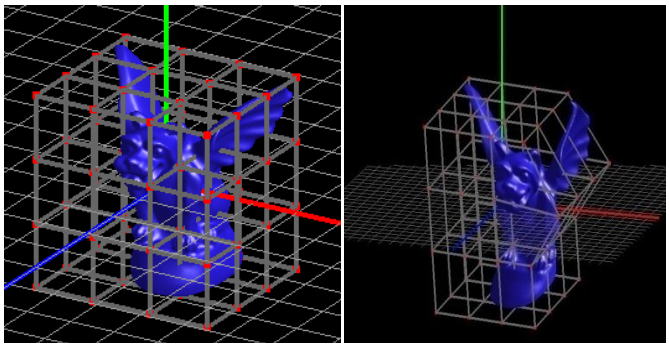


(a) (b)

Fig.7 (a) Pre-Deformed Eight Image (b) Post-Deformed Eight Image with (s=2, t=2, u=2)

D. Experimental Results for Gargoyle Image

Fig.8 (a) proves the screenshot of the simulation with the Pre-Deformed gargoyle image. The screenshots were also trapped in 4×4×4 frame. We can recognize from the gargoyle image that shams surrounded by the Sederberg boundary for Free-Form Deformation expansion.The screenshot results of post-deformed gargoyle image with (Movement of Four Points) is illustrated in Fig.8 (b).

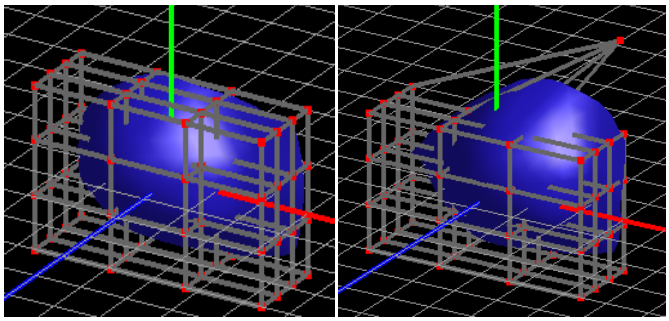


(a) (b)

Fig.8 (a) Pre-Deformed Gargoyle Image (b) Post-Deformed Gargoyle (Movement of Four Points)

E. Experimental Result for Cap Image

Fig.9 (a) confirms the screenshot of the simulation with the Pre-Deformed cap image. The screenshots were also fascinated in 4×4×4 frame. We can distinguish from the cap image that shams surrounded by the Sederberg boundary for Free-Form Deformation expansion. The screenshot results of post-deformed cap image with (s=2, t=2, u=2) is demonstrated in Fig.9 (b). In proportion to this simulation result, the cap image is distorted the shape by particular coordinate on the image.

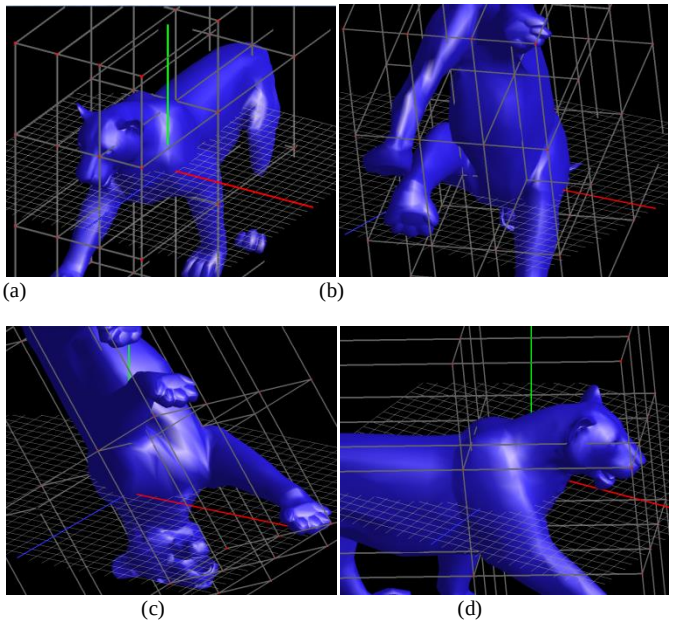


(a) (b)

Fig.9 (a) Pre-Deformed Cap Image(b) Post-Deformed Cap with (s=2, t=2, u=2)

F. Experimental Result for Lion Image

Fig.10 (a) corroborates the screenshot of the simulation with the Pre-Deformed lion image. The screenshots were also enthralled in 4×4×4 frame. We can discriminate from the lion image that shams surrounded by the Sederberg boundary for Free-Form Deformation expansion. The screenshot results of post-deformed lion image by rotating down position s,t,u axis is demonstrated in Fig.10 (b). In proportion to this simulation result, the lion image is rotated the shape by particular coordinate on the image. The screenshot results of post-deformed lion image by rotating up and right position s,t,u axis is demonstrated in Fig.10(c) and (d).



(a) (b)

(c) (d)

Fig.10 (a) Pre-Deformed Lion Image(b) Post-Deformed Lion with s,t,u Axis Rotation (Down)(c) Post-Deformed Lion with s,t,u Axis Rotation (Up) (d) Post-Deformed Lion with s,t,u Axis Rotation (Right)

VIII. Statistic Table for Simulation Results

The statistic table for the deformation of several models is given in Table.1. In this table, the deformation time for Eight image is the lowest deformation time than other models. The deformation time for Lion image is the highest deformation time with the same vertices. The deformation time of Knot image and Gargoyle image is approximately the same. According to this statistic table from the simulation results, the Sederberg Engine is applied to deform the several images and the several amounts of vertices.

TABLE I
STATISTIC TABLE

Model	Vertices	Frame Size	Algorithm	Deformation Times (Seconds)
Bunny Image	40002	4×4×4	Sederberg	0.1-0.2
Knot Image	5000	4×4×4	Sederberg	0.3-0.4
Eight Image	3070	4×4×4	Sederberg	0.05-0.15
Gargoyle Image	20002	4×4×4	Sederberg	0.3-0.5
Cap Image	186	4×4×4	Sederberg	0.2-0.4
Lion Image	5000	4×4×4	Sederberg	0.35-0.6

IX. Conclusion

We experienced clipping, especially when the picture was deformed in several unnatural ways e.g. trying to 'push' back a CP that had been 'pulled'. There are also cases where the object stretched out of the bounding box as in Fig.3, which was not expected. However, in casual cases e.g. game models that favor improved game performance over full accuracy of the models, the performance of these deformations, if applied over modest CP movement, is well worth the loss in accuracy, especially since the deformed transformations of objects are not truly intuitive processes.As the only value to change per calculation pass is the value of the single manipulated control point CP_{IJK}, to CP_{IJK}' at point I-J-K. All other ijk- and stu-values are unchanged per vertex. The rational here is that the weight of all other vertices and Control Points on the vertex v are the same as before CP_{IJK}'s shift: instead of a summation of all points each pass, a difference between the current and initial states is added or subtracted proportional to the Control Point IJK's shift. With this assumption in place, the calculation is greatly simplified. The six summations are eliminated, while the only values that need to be stored in memory are the co-ordinates of each vertex and Control Point in the stu-space.

Acknowledgment

I would wish to acknowledge the many colleagues at Mandalay Technological University who have contributed to the development of this paper. In particular, I would like to thank Myat Su Nwe, my wife, ThetHtarSwe, my daughter, and ZayYarTun, my son, for their complete support.

References

[1] Sederberg et al., "Free-Form Deformation of Solid Geometric Models", *Proceedings of SIGGRAPH 1986*, pp. 151-160, 1986.

[2] Battle et al., "Three-Dimensional Attenuation Map Reconstruction Using Geometrical Models and Free Form Deformations",IEEE Transactions on Medical Imaging, pp.404-411, 2000.

[3] Borrel, and Rappoport et al., "Simple constrained deformations for geometric modeling and interactive design", *ACM Transactions on Graphics, 13(2):*pp. 137–155, 1994.

[4] Chunyan, Jin and. Bin, et al, "Shape Edit Distance on Contour based Shapes", *Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications (ISDA'06)*, 2006.

[5] Difei, Ying, and Xiuzi, "A New Method of Interactive Marker-Driven Free form Mesh Deformation", *Proceedings of the Geometric Modeling and Imaging— New Trends (GMAI'06)*, 2006.

[6] Edward et al., "Interactive Computer Graphics A Top-Down Approach Using OpenGL", 2003.

[7] Masamichi et al., "Free-Form Deformation for Implicit Surfaces", *MSc Dissertation*, 2009.

[8] Petros et al ., "Dynamic Free-Form Deformations for Animation Synthesis", IEEE Transaction on Visualization and Computer Graphic, 1997.