A Real-time 3D Virtual Sculpting Tool Based on Modified Marching Cubes

Kuo-Luen Perng, Wei-Teh Wang, Mary Flanagan*, Ming Ouhyoung

Communication and Multimedia Laboratory
Dept. of Computer Science and Information Engineering, National Taiwan University
*University of Oregon
e-mail: {perng, bearw, ming}@cmlab.csie.ntu.edu.tw , mary@maryflanagan.com
http://www.cmlab.csie.ntu.edu.tw/~perng/sculpture

Fig. 1 Some of the results made by our system.

Abstract

We present a real-time interactive modeling system for users to create sculpture in a virtual world. This system uses voxels as the smallest editable unit, and uses a 3D tracker system to simulate virtual sculpting tools. A modified Marching Cubes is proposed for cutting edge anti-aliasing, and we apply octree and boundary division to accelerate the system and reduce the memory usage up to 20 times. Users can create virtual sculpture art pieces or 3D prototyping models easily through the intuitive user interface and immersing display device such as a HMD or a Vision Station. This system can run on a PC (PIII-500) supplied with any texture and lighting accelerated 3D graphics card in real-time. Fig.1 shows some of the models made by our system.

Key words: Modified Marching Cubes, Anti-aliasing, Fast Prototyping, Volume Rendering

1. Introduction

Although 3D modeling software has made much progress over the years, it has mainly emphasized precise modeling for CAD and 3D animations. Using those CAD programs to model a 3D object is a professional skill, and it requires such a long training time that some users find it difficult to use. We propose a modeling system that is highly intuitive. By using the system, an artist can carve an art piece with the 3D user interface, and a novice user can quickly and fluidly create objects.

The central contributions of this paper are:

- We use well-designed data structures and apply space partition techniques to reduce the memory usage and speed up the system. It requires 632MB of memory to record 80x80x80 marching cubes raw data. By using our optimization method, the memory usage requirement is under 32 MB.

- This system uses 3D tracker as input device, which works like the extension of users' hands. It is the most organic way to do sculpting. We also uses HMD and Vision Station as display devices, from which users may experience stereo and immersing environment.

- A modified Marching Cubes algorithm is proposed that can provide cutting edge anti-aliasing, and is partly based on the "Feature Sensitive Surface Extraction"[17] approach. The key point is that the surface normal at the cutting edge between the cutting tool and the object to be cut is calculated and recorded for later anti-aliasing use. Edge flipping techniques [17] are used to reduce the aliasing effect.

We will start with a brief overview of related work in the following section.

2. Related Works

2.1 Traditional Modeling Method

There are numerous famous 3D software packages such as 3D Studio Max, Maya, Lightwave, etc. They all include numerous powerful features to model 3D objects. Users have to learn and memorize significant function keys and parameters first, and try to combine those functions to make their desired results.

Those professional software packages have some drawbacks. (1) They are complicated and necessitate a long training time to master. (2) They all use the mouse

as the input device. The mouse, however, is a 2D device, but users have utilize it to edit an object in 3D space, and have to switch windows (view ports) to see different sides of the object. Our system tries to make the editing more organic, so we choose a 3D tracker as our input device.

Other modeling tools include digitizers and 3D scanners. These tools are required to convert a model made by a modeler with clay or other materials into digital data. However, digitizers need to sample numerous points sequentially by hand and that is time consuming; 3D scanners are not suitable to scan an object that is too reflective. In addition, if there is an obstruction between the object and the camera, a 3D scanner will generate data with unrecoverable gaps.

2.2 Previous Works

The basic concept of our system is to organize and edit a large amount of voxels in 3D space with memory efficiency, and tried to present those volumetric data in real-time.

Volume rendering has been researched for decades. Levoy et. al. proposed [2] and [3] when they tried to display the scanned data of Michelangelo's statues. However, their method needed preprocessing and could not dynamically add or remove voxels.

3D textures are another way for volume rendering [4]. But only high-end graphics workstations support this function. Memory requirements by 3D textures are incredible; for example, a 256x256x256 3D texture requires 192MB.

Our system is basically an extension from [1], which uses Marching Cubes algorithm [7] to present the volumetric data. We combine this algorithm with various techniques, such as the octree algorithm to accelerate the system, the edge flipping techniques [17] to reduce the edge aliasing, and boundary division to reduce the amount of memory usage.

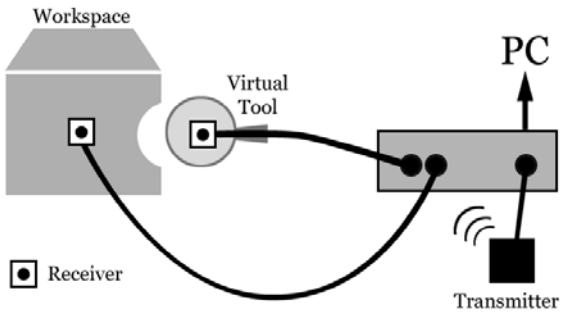

Fig.2 Setup of Virtual Sculpting system.

3. System Overview

Fig.2 is the setup of our system. Two receivers are used. One represents workspace and the other represent the virtual tool. Receivers will send its positions and orientations to our program for further calculations. Fig.3 is a screenshot from our system.

The workspace is filled with voxels, the basic editable unit in our system. Each voxel has two states, either "Stuffed" or "Cleared". Users can use different kinds of sculpting tools to modify the states. Only those "Stuffed" voxels that are also at the outmost surface of the object will be rendered by the modified Marching Cubes algorithm.

Fig.3 Screen shot from carving a vast.

3.1 Hardware Requirement

- **3D Tracker**: The most intuitive way to do sculpture is doing it in 3D space, so we chose 3D tracker as our user interface. This system requires two receivers: the user can hold a receiver in one hand, which controls the movement and the orientation of the virtual object that will be carved, like a wooden block. That means the object has 6 DOF and the user can transform it to see all its faces with only one input device. The user can hold the other pen-like sensor in the other hand, which simulates traditional carving tools. By using these two simple sensors, users can start to do the modeling without any further training.

- **RAM**: Our system typically requires about 32MB of RAM. It will take 632MB to record the raw data of a 16x16x16(cm3) workspace with the accuracy at 0.2 cm. We will use various techniques to reduce the memory usage under 32MB.

- **CPU & GPU**: Our program works well at a PIII 500 PC with a Geforce2 MX video card. The bottleneck is not on the CPU but rather on the GPU. Therefore, with a better video card, we can expand the workspace and make the voxels denser, which will create better, more precise results.

3.2 System Feature

Our system provides four main features: (1) Carving (2) Stuffing (3) Pottery Making (4) Painting. The user has

five virtual tools to choose from: (1) Sphere (2) Cube (3) Plane (4) Cylinder (5) Cone.

3.2.1 Carving

The user may initialize the workspace as a solid cube, wherein all voxel states are "Stuffed"-- except the outmost ones; we need them to be "Cleared". The reason will be explained later. The user may use virtual sculpting tools to carve this cube. Those voxels which collide with the virtual tools will be labeled as "Cleared", and the rendering list will be updated. In the carving mode, the user carves and sculpts as freely as would be done in real space, except for the force feedback.

3.2.2 Stuffing

The user may choose to start with an empty workspace with all the voxels in the "Cleared" state. In the "Stuffing" mode, voxels which collide with the virtual sculpting tools will be relabeled as "Stuffed", and the rendering list will be updated. This mode works like adding clay onto an object, and it can run cooperatively with the carving mode. If the user makes a mistake in the carving, he may use this mode to repair the object and carve again.

3.2.3 Pottery Making

We can turn our block as though it were on a virtual spinning table; this will drive the workspace to spin around an axis. It is like making pottery and it is easy to "throw" a pot of "clay" as it spins.

3.2.4 Painting

After the model has been made, the user can paint it. The painting will assign color to each vertex of the triangle list and the color will be interpolated and blended with the texture.

4. Sculpting and Display Pipeline

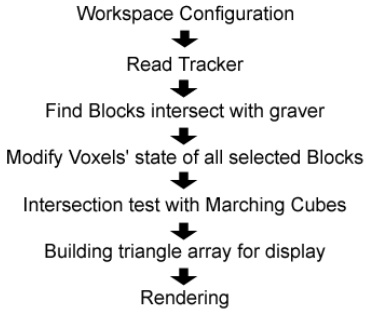

Fig.4 Flaw chart of sculpting pipeline

4.1 Workspace Configuration

First, the user has to set up the size of the workspace (the space containing editable voxels) and the density. These two parameters will define how many voxels will be used. The more voxels, the better the rendering result. We use the default values in the following discussion. Each axis is divided equally into 16 segments (totally 4096 blocks) and each block contains 5x5x5 Marching Cubes (totally 512000 Marching Cubes). Therefore there will be up to 81x81x81 voxels displayed with an accuracy of 0.2cm. Each voxel has two states, either "Stuffed" or "Cleared". We restrict the outmost voxels to a "Cleared" state, or the outmost marching cubes will not show any triangles (when all voxels of a marching cube are "Stuffed", we consider it inside the object and will not show any triangles.) Therefore the editable voxels will be 79x79x79.

4.2 Read Tracker

We continuously retrieve the tracker' s positions and orientations. One receiver controls the workspace and the other controls the virtual tool. We convert the coordinate of the tool into the coordinate of the workspace and do all the calculations in that coordinate. This simplifies the further calculation.

4.3 Blocks Intersection With Graver

Afterwards, we need to find out whether the tools have collisions with the blocks, and only those influenced blocks need updating. This step will take O (n) to check all the blocks by using brute force and will be reduced to O (log (n)) by using the octree traversal.

To calculate the collision detection in real time, the virtual sculpting tools provided are constructed of basic geometries that can be represented in simple mathematics.

If the virtual sculpting tool is considerably large, it may influence many blocks and the octree traversal will stop at a non-leaf node and retrieve the information of those blocks affected. Each block has a counter which records how many voxels are "Stuffed" within this block. We can skip the marching cubes test if a block has all its voxels "Cleared" in the carving mode, as well as all when it has all of its voxels "Stuffed" in the stuffing mode.

4.4 Modify Voxels' States

We now restrict our actions at certain blocks selected from the previous step. We would like to discuss this in the carving mode; the stuffing mode is using the similar concept and will be skipped. To carve the block means to change the voxel's state from "Stuffed" to "Cleared", and we will change only those voxels within the boundaries of the sculpting tool. As mentioned earlier, we do not record the voxels' coordinates in order to optimize memory use. We use the bounding box information of the block and the density parameter to

calculate the voxel position in real time. The pseudcode of these actions are as follows:

```
for (i = 0 ; i < max_voxels ; i ++)
{
    if( Filled(i) )
    {
      v = GetPositionOfVoxel(i);
      if ( InsideGraver(v) )
      {
        CutVoxel(i);
        }
    }
}
```

4.4 IntersectionTest With Modified Marching Cubes

A marching cube is composed of eight neighboring voxels. We sequentially check whether the marching cubes of a block intersect the tools. If so, we will check which edge of the marching cube intersects the tool chosen and then calculate that intersection point. Furthermore, the surface normal of the intersection point is calculated based on the contact surface between the cutting tool and the object to be cut.

Those marching cubes with their entire corner voxels "Stuffed" (within the object) or "Cleared" (outside the object) will not be checked. We only check those with part of their corner voxels "Stuffed" (at the surface of the object). We will calculate those edge-tool intersection points under two conditions: (1) The intersection point hasn't been calculated on that edge. (2) The edge already has an intersection point, but it is within the tool itself. We will update the triangle list afterwards.

The marching cubes are tightly connected. Neighboring marching cubes share some edges with each other. If we process each marching cube independently, calculating the intersection of the shared edge would be redundant. The result of the shared edge could be calculated only once if we don't process each marching cube independently. In order to save the redundant CPU time, we categorize marching cubes into eight different types according to their position.

Assuming there are M*M*M marching cubes in each block, we use a 3D vector, (n,n,n) for 0<=n<M, to represent the position of each marching cube in each block. The eight different types are:

1.Inside: when (n,n,n), 0<=n<M-1

2.In Top Face: when (n,M-1,n), 0<=n<M-1

3.In Front Face: when (n, n, M-1), 0<=n<M-1

4.In Right Face: when (M-1, n, n), 0<=n<M-1

5.In Top Front Edge: when (n, M-1, M-1), 0<=n<M-1

6.In Top Right Edge: when (M-1, M-1, n), 0<=n<M-1

7.In Front Right Edge: when (M-1, n, M-1), 0<=n<M-1

8.Corner: when (M-1, M-1, M-1)

Please refer to Fig.5 for the following rules.

For all the marching cubes labeled as "Inside", we only have to calculate edges 0, 3, 8. These "Inside" marching cubes share the intersection test results of edges 0, 3, 8 to the neighboring marching cubes which share the same edges.

For all the marching cubes labeled as "In Top Face", we only have to calculate edges 0, 3, 4, 7, 8 and apply this information to the neighboring marching cubes. Other faces are processed similarly.

There would be only one marching cube labeled as "Corner", and we have to calculate all the edges of that particular cube. Because it is located in corner, no edge could be shared with others.

Assuming Cube A,1,2 and 3 are all "inside", we only have to process edge 0, 3, 8 of each marching cube.

The result of edge 4 on Cube A comes from edge 0 on Cube 1. The result of edge 5 on Cube A comes from edge 3 on Cube 2.

The result of edge 9 on Cube A comes from edge 8 on Cube 3.

The result of edge 1 on Cube A comes from edge 3 on Cube 3.

To simplify it, sharing edges could be done only when marching cubes are within the same block. Of course we can share edges of the neighboring marching cubes between different blocks. The only thing we have to do is to position marching cubes in global, but not local space. Instead of positioning each marching cube in its block, we could position each marching cube in the whole workspace and use the same definition above. But in order to simplify the system, we implement our program to share marching cubes only within the same block.

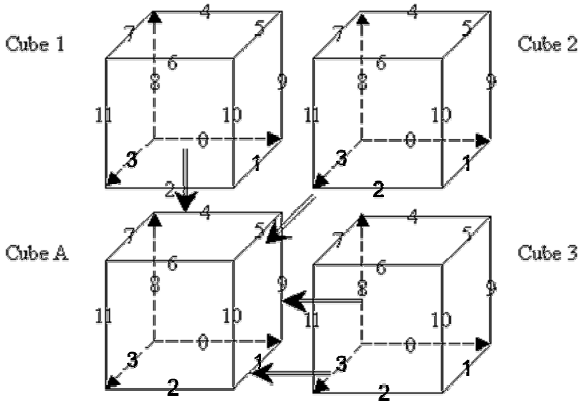

Fig.5 Blocks sharing edges.

4.5 Building Anit-aliased Triangle Array

After all the intersection points have been calculated, the

triangle array is generated by the proposed modified Marching Cubes algorithm. Our system records not only corners' filling states but also the intersection points and associated surface normals on each edge. The surface normal at the intersection point between the cutting tool and the object to be cut is exactly calculated and recorded for later anti-aliasing use. Therefore the rendering result is better than the original Marching Cubes algorithm. Fig.6 and 7 shows the difference. Note that the "distance field" representation used by Kolbelt et al [17] aims to find the surface normals at the intersection point. However, in our virtual sculpting tool, the surface normals can be calculated directly, and so distance field representation is not necessary for us. However, the feature sensitive sampling and edge flipping techniques proposed by Kobelt et al [17] is very useful and is used in our system accordingly. Anti-aliasing is therefore done in this modified approach.

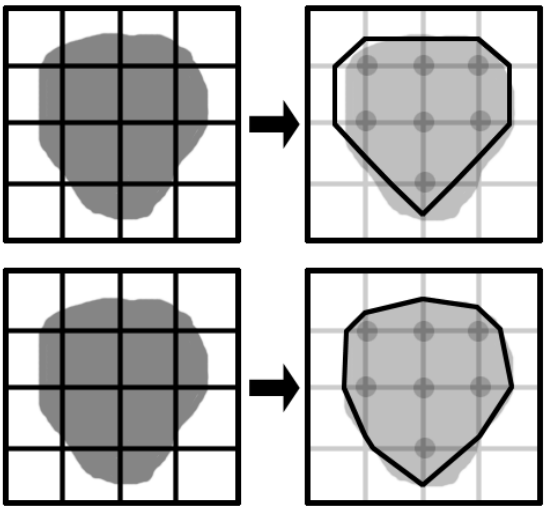

Fig.6 The top one is original Marching Cubes algorithm which only record corners' filling states. The bottom one is our method. We record intersection points and make triangles from these points, which will have better result.

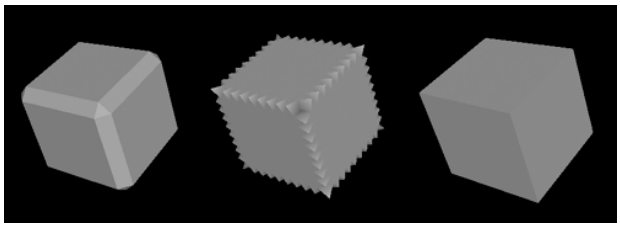

Fig.7 shows the edge flipping techniques (following Kobelt[17]) being used for anti-aliasing in intersection edge.

5 Rendering

5.1 Building Vertex array on VRAM

We have experimented with many methods to accelerate the rendering process. Our system uses OpenGL and will focus on this approach.

1. Put all vertices in an array

We found that giving the OpenGL API an array of vertex can render three times faster than immediate mode. The vertex information stored in the main memory needs to be downloaded to the VRAM of a T&L graphics card to be able to render them. Each API call will be followed by a download operation, which will become a burden if it runs too many times. The fewer the API calls, the better the performance.

2. Put vertex array on the VRAM

When rendering a lot of triangles, the bottleneck is not the computing power of the GPU but the memory transfer bandwidth. The GPU has to download the information of vertices from system memory. To render a scene containing 300,000 triangles at 30 fps with sufficient information about each, including vertex coordinates, normal, color and texture coordinates, the bandwidth needed to transfer such information is 377MB/sec. It takes a long time to download and at the mean time, the GPU will idle and waste its computing power. Therefore a graphics card which claims to render 10 Million/sec can only render 2 Million/sec when all the information is stored on the main memory.

Our system stores the triangle information directly on the VRAM and will almost double the rendering speed.

5.2 Texture

We can apply different texture maps to the model, so that users may feel as though they are carving a range of materials such as a wooden block or marble.

We use Decal method to calculate the texture coordinates. We only take x and y value (normalized) from the vertex's coordinates as the texture coordinates.

Assume (xmax, ymax, zmax) are max coordinates of workspace. (xmin, ymin, zmin) are min coordinates of workspace.

Width = xmax – xmin;

Height = ymax – ymin;

Tx = (x-xmin)/width;

Ty = (y-ymin)/width;

This method could be performed automatically by GPU with proper configuration. So there is no need to record the texture coordinates and the system thus saves memory.

5.3 Coloring

The painting methods use a similar pipeline as does carving, which traverse the octree to find the block that is about to be affected.

At first, check all the edge points of the marching cubes in the affected block to see whether these points are lying in the tool. If so, change the color of the points to the selected color. Afterwards we will update the triangle list with the new color.

6 Memory Management

First, we will show the memory usage of the raw data, and then we will discuss how to use memory efficiently with the data structure we propose and some dynamical allocation techniques.

6.1 Raw Data Storage

Let's take a look at the raw data storage requirement in our system.

BlockManager: two 3D vectors for the boundary box.

Block: two 3D vectors for the boundary box.

Marching Cube: two vectors for the boundary box, states of 8 voxels and intersection points of 12 edges.

We assume there are BxBxB Blocks in BlockManager and MxMxM marching cubes in each block.

We need information of position, normal, and color for each vertex on each edge.

MarchingCubes = B*B*B*M*M*M.

Vertices = MarchingCubes * 12 (edges).

Bytes_Per_Vertex = (3+3+3)*(3*4) = 108.

Bytes = Vertices * Bytes_Per_Vertex.

Take our default configuration as an example, B=16, M=5. We need 632MB RAM to store all the required information, not even including the bounding box of each marching cube and the state of each voxel.

Programs always run slower when they use more memory because of the cache miss. In addition, most consumer PCs don't have more than 128MB of memory.

6.2 System Memory Management

To save the memory usage, we use some mechanisms to reduce the memory requirement to 32MB-64MB.

1. Only BlockManager and block have to remember their bounding boxes. The bounding boxes for marching cubes are always calculated in run time.

2. Positions of each voxel are calculated in run time.

3. Only allocate memory for marching cubes in each block when necessary. If a block is fully "Stuffed" or all "Cleared", the system doesn't have to do anything about marching cubes inside the block. So there is no reason for us to allocate memory for those marching cubes.

4. Only allocate memory for the edges of the marching cube when it has intersections with the virtual sculpting tools, and release the memory when the intersection no longer exists.

Using this mechanism, a marching cube doesn't always require memory for each edge. A block doesn't always require memory for each marching cube

In the typical case, it is impossible for each marching cube to have 12 intersections all of its edges. It is also almost impossible that every block has to allocate memory for their marching cubes. In our observation, this system seldom uses more than 50MB of memory.

6.3 Video Memory Management

If we could put our display vertex information in the video memory, the FPS could be doubled compared with putting them in the system memory. More FPS means you can work with a higher resolution within our system. Currently, every display card has more than 32 MB memory. But the memory must be reserved for the Frame buffer, Zbuffer and textures. We can use no more than 20MB of the memory. When each block has to rebuild its triangle array, the most efficient way to this is to build it in the video memory. But this is not practical because we have to allocate more memory for the blocks when they have more triangles and have to release memory when they don't have any triangles. This could cause memory fragments. So we cannot allow each block to allocate video memory on its own. A brute-force algorithm is used to allocate one very large array in video memory, and equally divide the array for each block. But it only works if you have a lot of video memory. For the reasons above, our system builds the triangle array in system memory. The array will be copied to video memory after it's constructed.

This is the way in which we use video memory:

1. Allocate some video memory when initialized.

2. Rebuild triangle array of each block in system memory when necessary.

3. Calculate memory requirement for display for all blocks.

4. Reallocate video memory if necessary.

5. Copy triangle array from the smallest modified block index to video memory.

6. Go back to step 2.

Copying the triangle array from the smallest modified block index means that, for example, if a user modified the marching cube in the 20th block in BlockManager, the system doesn't have to update the video memory which stores the triangle array of 1-19 blocks.

7 Display Devices

Normal CRT and LCD monitor lacks perspective. Users cannot get depth information from those display devices. Therefore it is hard to create the model very accurately. Two kinds of 3D display device are adapted to our system; they are HMD and Vision Station. By using HMD, users may have stereo view and can carve the model more accurately. The Vision Station is new a 3D display device. Users are actually surrounded by a dorm like screen and may experience immersing environment (Fig.8).

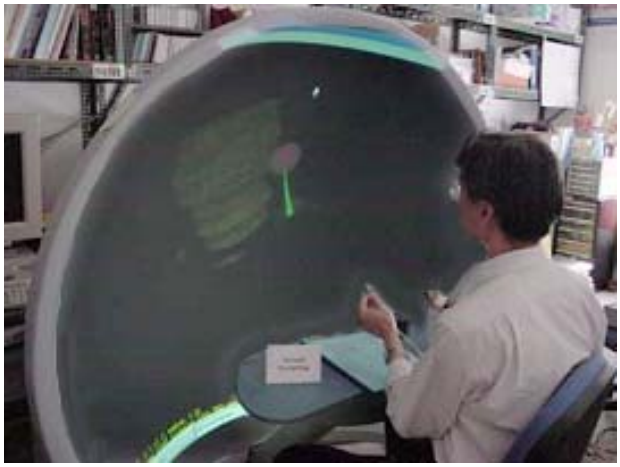

Fig.8 Our system runs on a Vision Station.

8 Import and Export

Our system supports importing and exporting models.

8.1 Import

We can import models made by other CAD software and edit them in our system. Most of the 3D files contain the model's triangle information; thus we have to convert those triangles into voxels and marching cube information. The process of importing is as follows:

1. Import the model as triangles and put it into the working space. Each voxel has 6 outgoing rays along +x, -x, +y, -y, +z, and -z. We will check the triangle first hit by the ray. If the triangle is facing outward, then this voxel is considered inside the model ("Stuffed"), and vise versa.

2. Calculate the intersection points between the triangles and the edges of the marching cube to construct the rendering list.

We apply spatial divisions to the imported triangles. Marching cubes of each block would only check with spatially related triangles with their own blocks, which will speed up the process.

8.2 Export

Exporting is much simpler. We already have all of the triangle information of the model. Therefore we just need to write the data out in certain 3D file formats.

9 Results and Gallery

A few samples using our system are given below as results.

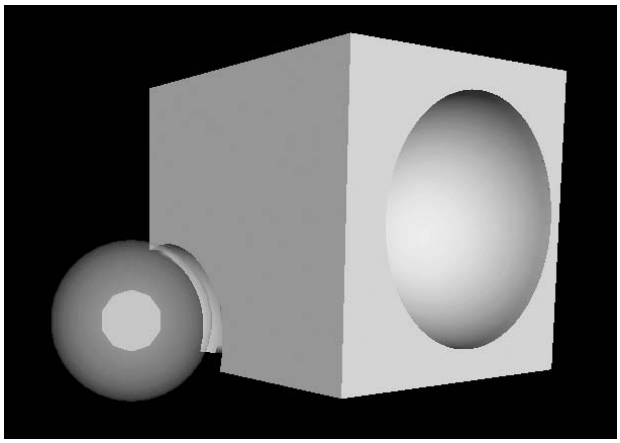

Fig.9 A cube with two holes being cut where the new proposed modified Marching Cubes approach is applied. Note that aliasing at the hole boundary is greatly reduced..

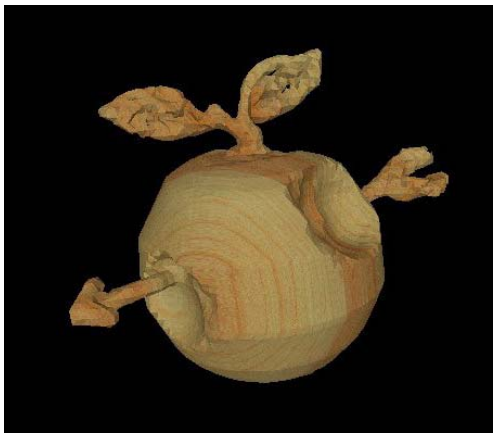

Fig.10: Hagihara Tadanori created this "An apple pierced by an arrow" in less than 15 minutes. He used both stuffing mode and carving mode.

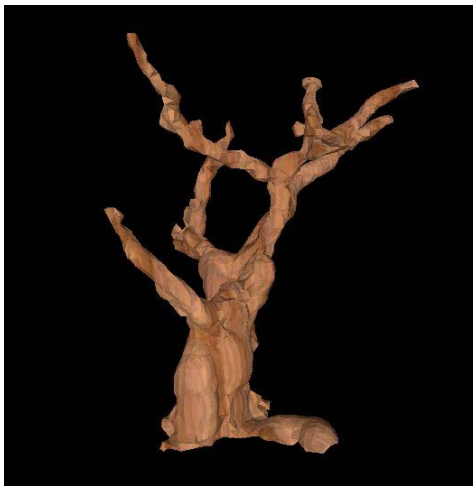

Fig.11 This "Gnarled Tree" was created by visiting professor Mary Flanagan in less then 10 minutes. She mostly used stuffing mode with a sphere graver to create this.

10 Conclusions

A real-time virtual sculpturing tool is developed and a modified marching cubes algorithm is proposed. Traditional marching cubes algorithm has the problem of aliased edges, and so precision is always a problem. Our system records not only corners' filling states but also the intersection points and associated surface normals on each edge. The surface normal at the intersection point between the cutting tool and the object to be cut is exactly calculated and recorded for later anti-aliasing use. The edge flipping techniques proposed by Kobelt et al [17] is also implemented and aliasing at the intersection edge is greatly reduced.

In short, our design makes it possible to run this system on a PC with a texture and lighting accelerated graphic card. The system makes it possible to create a digitized sculpture intuitively and can be used as a fast prototyping tool. Besides, initial users note that the system provides artists with a brainstorming environment that can stimulate their imagination.

References

1. Galyean ,T. , Hughe ,J. "Sculpting: An Interactive Volumetric Modeling Techniques" . Proceedings of SIGGRAPH'91, 1991, pp267-274.

2. Levoy, M. and Whitted, T. "The Use of Points as a Display Primitive". Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

3. Rusinkiewicz, S. and Levoy, M. "Qsplat: A Multiresolution Point Rendering System for Large Meshes". SIGGRAPH 2000, pp. 343-352Westermann, R., Ertl, T. "Efficiently Using Graphics Hardware In Volume Rendering Applications". SIGGRAPH 1998, pp. 169-177.

4. Cabral, B., Cam, N., and Foran, J. "Accerlerating Volume Reconstruction with 3D Texture Hardware". ACM Symposium on Volume Visualization '94, pp. 91-98.

5. Danskin, J. and Hanrahan P. "Fast Algorithms for Volume Ray Tracing.". ACM Workshop on Volume Visualization '92, pp. 91-98.

6. Lacroute, P., and Levoy, M. "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation". Proceedings of SIGGRAPH '94, 1994, pp.451-458.

7. Lorensen, W. and Cline, H. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". SIGGRAPH 1987, pp. 163-169.

8. Pfister, H., Zwicker, M., van Baar, J. and Gross, M. "Surfels: Surface Elements as Rendering Primitives". SIGGRAPH 2000, pp. 335-342.

9. Nvidia OpenGL Extension Specs http://www.nvidia.com/ March 2, 2001.

10. Eric A. Bier. "Snap-Dragging in Three Dimensions". In 1990 Symposium on Interactive 3D Graphics, SIGGRAPH, 1990, pp. 131--138.

11. Barillot, C., Gibaud, B., Scarabin, J., and Coatrieux, J. "3D Reconstruction of Cerebral Blood Vessels". IEEE Computer Graphics and Applications 5,12 (December 1985), pp. 13-19.

12. R. Lewis and C.H. Sequin, "Generation of Three-Dimensional Building Models from Two-Dimensional Architectural plans," Computer-Aided Design, vol 30, no 10, 1998, pp 765-779.

13. Pei-Wen Liu, Lih-Shyang Chen, Su-Chou Chen, Jong-Ping Chen, Fang-Yi Lin, and Shy-Shang Hwang,"Distributed Computing: New Power for Scientific Visualization." IEEE Computer Graphics and Applications, Vol. 16, No. 3, May 1996, pp.42-51.

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

15. Janis Wong, Rynson Lau, and Lizhuang Ma, "Virtual 3D Hand Sculpturing with a Parameter Hand Surface," Computer Graphics International, IEEE Computer Society Press, June 1998, pp. 178-186.

16. Alan Watt "3D Computer Graphics Third Edition" ADDISON-WESLEY 2000.

17. Kobbelt, L.P. Botsch, M. Schwanecke, U. Seidel, H-

P. ”Feature Sensitive Surface Extraction fromVolume Data”. Proceedings of SIGGRAPH 2001, pp. 57-66.