Virtual Sculpture

E. Ferley, Marie-Paule Cani and Jean-Dominique Gascuel

iMAGIS/GRAVIR-IMAG, 220 rue de la chimie BP53, 38041 Grenoble Cedex 9, France

Abstract
We propose here a sculpture metaphor for rapid shape prototyping. This metaphor makes the underlying surface description transparent for the user, enabling him to focus only on the shape being modeled. Our approach is based on implicit surfaces defined as iso-surfaces over a discrete field.

1. Introduction

We are seeking an *intuitive* modeling tool for designers or sculptors creating digital 3D models. A huge amount of work has been done in Computer Graphics to provide an intuitive design metaphor. In practice, there are still a lot of parameters to tune and limitations on the object's topology and geometry due to the underlying mathematical description. The user's attention then focuses on the model decomposition into patches and the continuity relationship between them, rather than on the shape itself.

With this in mind, a *sculpting* approach where a user could deposit material wherever he desires in space, and then iteratively deform, carve or paint it with a tool, without any consideration on its underlying description, seems more adequate to us.

Bill and Lodha[3] developed this *sculpting* metaphor using polygonal models. However, the user still had to worry about the mesh representation, since he had to control its resolution, delete some polygons to create holes or possibly handle its reconnections.
A better approach, in our opinion, was initiated by Galyean and Hughes[4]: the object is defined as the iso-surface of a discrete field sampled on a grid. The tool alters the surface by locally modifying the samples values. The surface is then computed using an *incremental marching cubes* algorithm, which means that the marching process is only performed in the modified region. This representation inherits the facilities of implicit surfaces to handle arbitrary topologies. Avila[2] and Massie[5] added some kinesthetic feedback to similar sculpting systems[†].

We consider the grid representation as a major limitation in these approaches as it imposes limits in space to the model extent in addition to its storage cost. Moreover, it seems useless to store a whole cubical-grid if we only want to model a thin shape, such as a branch.
We present here an implementation based on balanced binary search trees to store the volume data only where necessary.

2. Our Sculpting Metaphor

Figure 1 shows a typical screen snapshot of our application. The user can move each object either with a mouse or with a 3D input device such as the Spacemouse. The tool's color and shape can be modified. The tool's action is selected either by (function) keys, or Spacemouse buttons, or radio button in the interface. Pressing "space" applies the tool.

2.1. Classical tools

The tool can **create** some material, just as the *toothpaste* described by Galyean[4], by cumulating the associated field value into the grid samples. This added material is consequently blended with the (possibly) existing one. The tool can also **remove** some material, either progressively (*softEraser*) or not (*eraser*). Similarly, the tool can alter the color (*softPainter* and *painter*).

[†] The former uses ray-cast rendering which differs from the marching cubes used in the other approaches, but the principle is the same.

Published by the Eurographics Association, ISSN 1017-4656.

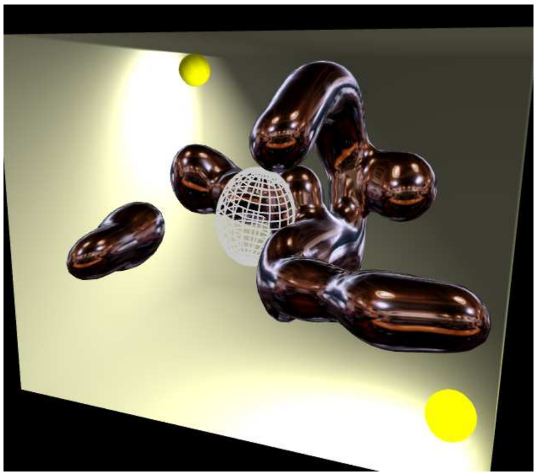

Figure 1: *Sample screen snapshot of the application: we can see a box representing the workspace, two lights (the yellow spheres), a tool displayed in wireframe mode, and a sculpted object environment-mapped.*

These actions were both present in the papers from Galyean[4] and Avila[2], but they were restricted to sphere-shaped tools. Here the tool shape is a general ellipsoïd. Instead of computing the image of each grid sample in the tool local frame, we compute this image once for one point (the lower-left-bottom corner of the axis-aligned tool bounding box), and the three world axes. We then *walk* through this bounding box both in world and local coordinates by simply adding the corresponding displacement vector to join the next grid point. This speeds up the update of the grid samples covered by the tool, and enables any tool's affine transformation at no extra cost.

2.2. Local Deformation Tool

We are experimenting with new kinds of tools that **deform** the existing material, just as if the user was pushing the clay with his finger, or with a sculpting tool such as a gouge. We want to avoid the computation cost of material displacement simulation when a collision between the tool and the object occurs. Consequently, we use a purely geometric approach, adapted from the Opalach/Cani-Gascuel method[6] which was dedicated to skeleton based implicit surfaces.

We use a deformation function (see Figure 2) to compute the ratio of the existing field that will be used to modify the shape: at the center of the tool we add -1.0 times its own value to the field; at its frontier we do not modify the field: thus, the iso-surface won't be altered; in a bounded region around the tool, we increase the field, so that a bump will appear. The zero-crossing of the function is imposed by the tool shape, but both the slope at that point and the *exterior* bump's height, location and extent can be tuned. These parameters are not yet automatically deduced from the characteristics of a given material. However, it's possible to achieve *visually realistic* deformations (see Figure 3).

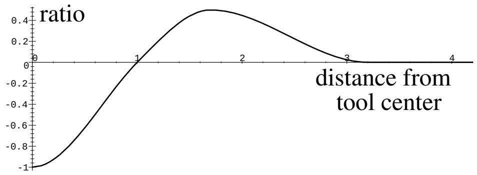

Figure 2: *Shape of the deforming function. Horizontal axis is the distance from the tool center. Vertical axis is the ratio.*

Known limitations: Since this deformation function is used only as a ratio to modify the existing field, there will be no expansion of this existing field in space. Problems may occur when a huge bump appears, and is limited only by the field extent. This can be seen as a field definition problem, and handled as a field conditioning step.

2.3. Undo-Redo Handling

Each time a tool is applied, the covered *grid points* are marked `dirty`, and their previous state is dumped into an *undo-file* in a temporary directory. We arbitrarily limited the number of these undo-files to 200, but in fact the real limitation to this is disk space. These files allow us to browse among the UNDO/REDO history.

3. Tree Structure

We only store in the tree *field values* that are greater than an arbitrary *minVal*. The values are symmetrically clamped to a *maxVal*, and the displayed surface corresponds to a given *iso*. Since we store fewer field samples than in a classical grid, we can store more information per sample. This avoids useless re-computations (such as world coordinates, field gradients, ...) and speeds up surface update.

We call the points where the field is sampled `corners`. `Corners` store their value as one float, their location and gradient as three floats, and their color as four unsigned bytes. The `cornersTree` balanced binary search tree we build from these `corners` is ordered according to their `key`, i.e. their *grid indices* (i, j, k). Two keys are compared in lexicographical order, that is: firstly on their i component, then (if $i_1 = i_2$), on their j component, and finally (if $i_1 = i_2$ and $j_1 = j_2$) on their k component. After each insertion/deletion, the tree equilibrium is checked to preserve the $O(n \log n)$ operation time on it.

To polygonize the surface, we then need to examine the

cubes. A `cube` is made up of eight `corners`, twelve `edges` and an `index` representing the surface configuration inside it. We store another tree for the cubes, also ordered with an (i, j, k) key. This `cubesTree` stores all the existing cubes. A cube is removed when all its corners are deleted, i.e. their value has become lower than *minVal*.
When a cube appears to cross the iso-surface, it is inserted into another simplified tree storing only pointers to these *crossing cubes*. At the time of this insertion, the cube's `index` is computed from the values of its eight `corners`. The edges intersected by the surface are created and inserted into another binary search tree `EdgesTree`. An `edge` stores this intersection point, i.e. its location and normal, as three floats, and its color as four unsigned bytes. This intersection is actually computed by linearly interpolating the corresponding fields of the two `corners` involved, according to their respective field value.

Surface update is interactive, since we only examine the corners (and related cubes) that were *dirtied* by the tool.

4. Visual Quality

With this test platform, we realized how crucial the visual quality is for the user's comfort, but also for the tool's position perception, and for the object's shape estimation.

One advantage of the *Infinite Reality* graphics card we use is its ability to antialias OpenGL primitives at *no cost* thanks to its hardware support of the multisample extension. We also tried some stereo rendering using some *Stereographics* shutter-glasses (*Crystal Eyes* model), and a *virtual-IO* HMD using interlaced rendering (*i-Glasses* model). Both are still in an early stage of development since we do not correctly handle the convergence/zero-parallax problem, and we do not track the head position. Even in this simple configuration, this proved very helpful for the tool placement in space.
Another feature which greatly enhances the shape perception is the use of environment textures that are *sphere-mapped* onto the object. This looked particularly useful if the surface had degenerated triangles, which is a typical drawback of the Marching Cubes algorithm. We used classical sphere-mapping with adjustable transparency to be able to see the surface color under the texture layer. We also implemented simplified *ClearCoat*[‡] like effect, i.e. simulating a paint layer, using a texture transparency varying accordingly to the incidence angle between the viewer and the surface.

[‡] information concerning SGI's *ClearCoat* product may be found at `http://www.sgi.com/newsroom/press_releases/1998/september/clearcoat.html`

5. Results

We obtain interactive response times without the need of any dedicated/specific volume rendering hardware. At the expense of reduced performances and visual quality (no *multisampling* antialiasing, and slower frame rates), our application also runs on a standard PC using OpenGL under WindowsNT.

Galyean[4] used grids from 30^3 up to 60^3, while Avila[2] reports the use of grids up to 256^3. Pfister[7] uses special purpose hardware based on his *Cube-4* ASIC to render a 256^3 volume with ray-casting up to 30 frames per second. Here the user is free to resize the workspace's box at any moment, and extend his model wherever he wants, providing *virtually unlimited* grid size. Since the field sampling is regular, two kinds of limitation appear in the current implementation:

- small tool: the sampling points become too distant relatively to the tool size. The tool isn't correctly sampled, and artifacts due to noise appear.
- large tool: the tool covers so many sample points that their update is no longer possible at interactive rates.

We report in the following table some statistics concerning three differently sized *toothPaste* tools adding some material to the object represented in Figure 1. This object corresponds to $15,573$ values stored and `cornersTree`, `cubesTree` and `edgesTree` of respective depth of 14, 16 and 13. The iso-surface displayed has $4,200$ vertices and $8,392$ triangles. The application runs on an SGI *Onyx2/IR* with 1Gb RAM and a 195MHz R10k processor. For each tool size, we report:

- an average frame rate (wall-clock time).
- some results concerning the number of corners and cubes covered in a *best* and *worst* case, depending of the tool's local bounding box orientation.
  - the corners **visited** are the corners covered by the axis-aligned tool's bounding box.
  - the corners **computed** are the corners covered by the oriented tool's bounding box but lying outside the tool, i.e. having a *null* contribution from the tool.
  - the corners **dirtied** are the corners having a *non-null* contribution from the tool.
  - the cubes **treated** are the cubes which had at least one corner dirtied, and hence were updated. This means that we recomputed their configuration `index`, and the `edges` intersected where updated, if needed.

frames/s	#corners visited	#corners computed	#corners dirtied	#cubes treated
19-23	216	125	93	184
small	1331	203	110	209
7-8	1452	887	501	751
medium	4352	975	508	771
3-4	2744	2197	1021	1424
large	8316	1919	1003	1407

This shows that as we are able to rapidly reject the corners lying outside of the local tool's bounding box, the tool orientation isn't affecting the field update performance.

6. Future work

There are still some improvements to conduct concerning the visual quality, such as enhancing the stereo display, or adding some visual cues such as shadows (either with textures, or volumes) or depth of field.
Another key feature that will definitely improve the immersion of the application into reality is force feedback: a first idea was proposed by Avila[2,1], but we foresee that it will add more constraints on the field conditioning (such as mentioned in 2.2).
An important limitation in our current implementation is the fixed sampling resolution. At the moment, we are planning to use octrees instead of binary search trees to store field values, but a multigrid approach also looks promising.

Acknowledgments

This work is supported by Renault and CNRS. We would like to thank Andras Kemeny for making the whole project possible. Many thanks to Frédo Durand for valuable discussions and intuitions concerning the one-pass rendering of angular dependant environment textures. This work is done in a great atmosphere thanks to the iMAGIS team (many thanks to the many people who did reread this paper!). We would also like to thank the many people who contributed to develop GLUT, and Paul Rademacher for providing GLUI[§]. Both are really helpful to develop cross-platform OpenGL-based application.

References

1. R.S. Avila. Volume haptics. *Computer Graphics*, pages 103–123, July 1998. SIGGRAPH'98 Course Notes #01.
2. R.S. Avila and L.M. Sobierajski. A haptic interaction method for volume visualization. *Computer Graphics*, pages 197–204, October 1996. Proceedings of Visualization'96 (San Francisco).
3. J.-R. Bill and S.K. Lodha. Sculpting polygonal models using virtual tools. In *Graphics Interface'95*, pages 272–278, 1995.
4. T.A. Galyean and J.F. Hughes. Sculpting: An interactive volumetric modeling technique. *Computer Graphics*, 25(4):267–274, July 1991. Proceedings of SIGGRAPH'91 (Las Vegas, Nevada, July 1991).
5. T. Massie. A tangible goal for 3d modeling. *IEEE Computer Graphics and Applications*, 3:62–65, May 1998.
6. A. Opalach and M.-P. Cani-Gascuel. Local deformation for animation of implicit surfaces. *Proceedings of SCCG'97 (Bratislava, Slovakia)*, June 1997. can be found at `http://www-imagis.imag.fr/`.
7. H.P. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. *Computer Graphics*, pages 47–55, October 1996. Proceedings of Visualization'96 (San Francisco).

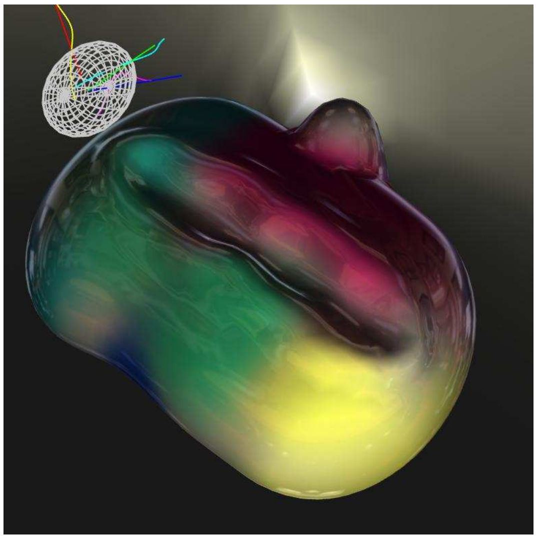

Figure 3: *Sample screen snapshot of an object deformed with our local deformation tool.*

§ The GLUI User Interface to GLUT can be downloaded from `http://www.cs.unc.edu/~rademach/glui`