Practical Volumetric Sculpting

Eric Ferley, Marie-Paule Cani, Jean-Dominique Gascuel

Practical Volumetric Sculpting

Eric Ferley, Marie-Paule Cani and Jean-Dominique Gascuel

iMAGIS[†]-GRAVIR/IMAG-INRIA

Abstract

We present here a *sculpture metaphor* for rapid shape-prototyping. The sculpted shape is the iso-surface of a scalar field spatially sampled. The user can deposit material wherever he desires in space and then iteratively refine it using a freeform tool. The tool shape can be designed inside the application. The tool can add, remove, paint or smooth material, or it can be used as a stamp to make prints on an existing shape. The user can move the scene and/or the tool with a Spacemouse (6*D* input device), or a 2*D* mouse using virtual trackball. We also focussed on the rendering quality, exploiting lighting variations, and environment textures that simulate high quality highlights on the surface. Both greatly enhance the shape estimation, which is in our opinion a crucial step during the design process. We also explored some stereo configurations to improve the user's perception of the spatial relationships between the elements of the scene.
Our current implementation based on GLUT allows the application to run both on UNIX-based systems, such as IRIX and Linux, and on Windows systems. We obtain interactive response times, strongly related to the size of the tool. The performances issues and limitations are discussed.

Categories and subject descriptors: I.3.5 [Computer Graphics]: Object Modeling – Solid and object representations, Modeling packages; I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Virtual reality

Additional Key Words and Phrases: Interactive sculpture, Local deformations, Discrete implicit surfaces, Volumetric representation, Balanced binary trees

1 Introduction

The general context of our work is 3*D* freeform design. We deliberately chose a *sculpture metaphor* based on iso-surfaces over a sampled scalar-field.
We propose here a simplified classification based on some major/most recent contributions that came to our knowledge. This classification helps us to present the choices we made, and places our approach in the context of freeform modeling.

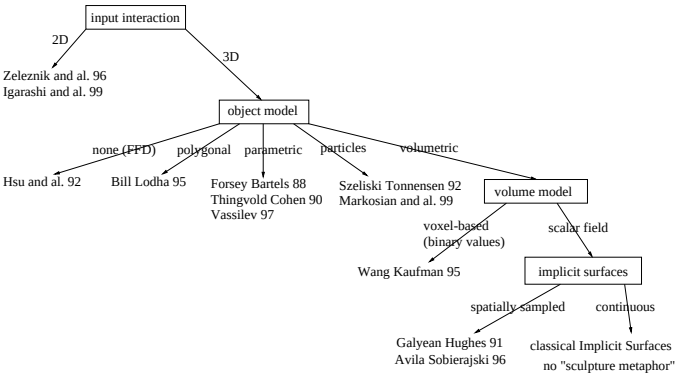


Figure 1: Proposed simplified classification of approaches to rapid prototyping of 3*D*-shapes

The first distinction we make concerns the input device used. One possible approach to designing 3*D* shapes is to use 2*D* input devices, such as a classical mouse or a pen to draw strokes on a screen, and then deduce plausible 3*D* shapes fitting these silhouettes strokes. Some interesting approaches have been conducted such as the SKETCH system[20] that enabled rapid shape construction using simple primitives (boxes, spheres, cones) that are combined with CSG-like operations and positioned with gestures. This was recently extended with the Teddy system[10] which uses freeform strokes to deduce 3*D* polygonal model and then simple gestures to extrude, smooth, cut or deform (bend) it. These approaches are very appealing both because of their close relation to the pen and paper interaction, which is natural when roughly sketching ideas in early stages of design, and because of their simplicity in software interface (no sliders, buttons, confusing options) and in hardware interface (simply a pen).
Nevertheless, we preferred 3*D* input devices (also called *direct* or 6*D* input devices), as we believe it allows a more precise control over the 3*D* shape. When considering more complicated devices, such as the Phantom force-feedback articulated arm, the device also allows to *feel* the surface being modeled, which greatly enhances the surface perception, and consequently its valuation.

The second distinction deals with the underlying model description, which also strongly conditions the range of possibly achievable objects. We only consider here approaches based on direct surface interaction. Free Form Deformation[9] is a powerful method to deform any kind of object. Mixed with a force feedback device[6], it leads to a convincing *clay modeling metaphor*. However, the serious limitation of this space deformation technique is that the user can not modify the topology of the deformed object. In the polygonal sculpture system proposed in [3], the user may alter the topology, but he has to specify the new connectivity at a triangle level. Moreover, when the user deforms the shape, detecting and handling its self-collisions become very intricate. Another inter-

[†]iMAGIS is a joint research project of CNRS, INRIA, Institut National Polytechnique de Grenoble and Université Joseph Fourier.
Contact: {Eric.Ferley, Marie-Paule.Cani, Jean-Dominique.Gascuel}@imag.fr or `http://www-imagis.imag.fr`

esting approach was initiated with Hierarchical B-Splines[7], that allowed to add details via *overlays* locally refining the shape. Intuitive spline manipulation was extended with physical simulation processes such as [16, 17]. But here again, topology changes, especially when auto-intersections occur, are very difficult to handle (and are not handled, to our knowledge). Volumetric models enable one to easily represent 3D shapes of any topology. However, building a sculpting metaphor from classical implicit surfaces seems difficult. Indeed, these surfaces are usually defined as the blending of elementary potential fields generated by individual primitives. The number of these primitives greatly affects the field evaluation cost. Since iterative shape refinement is a key aspect of sculpting, a classical implicit representation would lead to a complexity explosion due to the increasing number of primitives.

In this context a representation of the field function as values stored in a grid seems much more appropriate. Then, whatever the number of editing operations, tri-linear interpolation always evaluates the field value in constant time.

We adopted here this representation, which was already studied in [8, 2].

This paper extends the tools shapes to freeform models that can be designed inside the application. It also proposes a new tool action which allows to use the tool as a stamp printing its shape on the virtual clay.

We also pay attention to the rendering quality, which greatly enhances the perception of the 3D shape, which is a crucial task in the design process.

We firstly describe our application from a user point of view. Then we describe our current implementation through a couple of *how does it work* sections: the first one dedicated to the potential field storage and update, and the second one dedicated to the tools shapes and actions. Finally, we detail our visual feedback, estimate our system performances on sample cases, and discuss future works.

2 Main features of the system

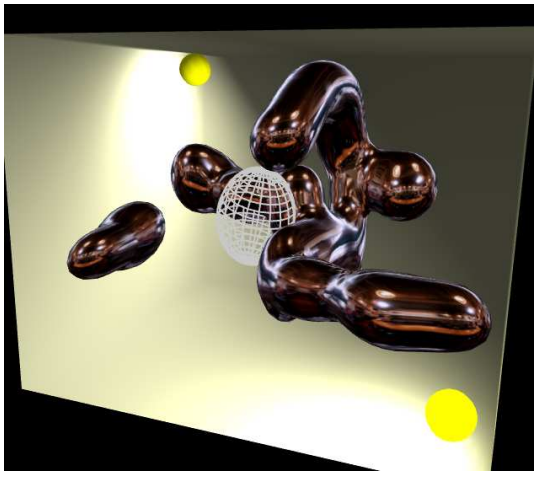


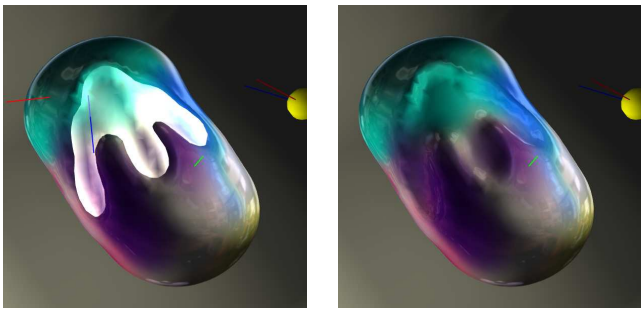
Figure 2: Sample snapshot of our sculpting application. The object being modeled is environment mapped. The tool is displayed in wireframe mode. The two yellow spheres represent the lights and a box representing the workspace is displayed.

Figure 2 shows a typical screen snapshot of our application. We display a box to delimit the workspace. This may also serve as a landmark for the user. This box can be modified to surround the sculpture, or hidden.

During a standard sculpting session the user selects which of the scene, tool or light he wants to move, and the device, either the mouse or the Spacemouse in our current implementation, he wants to use. It's possible for instance to use the mouse to rotate the scene, while controlling the tool's location with the Spacemouse, or the opposite.

The default tool is an ellipsoid which can be resized. It may be hidden, or displayed in transparent or wireframe mode. Its default action is *toothpaste*, which progressively deposits material in space. The user can edit its color, and the speed of the material deposit.

Other possible tool actions are *eraser*, or its progressive counterpart *soft eraser* that remove material in space. Identically, a *painter* and *soft painter* apply the tool color to the existing material. It's also possible to smooth an existing surface. Our particular implementation of these *classical* actions will be discussed in section 4.2. The



(a) (b)

Figure 3: Using a tool designed inside the application as a stamp to make a print on a shape. (a) the tool is displayed in transparent mode. (b) the same view without the tool

tool can also be used as a stamp to locally deform a shape. As the user can also design his own tool shapes inside the application, he's able with this tool action to make a print on an existing shape simply by pushing the tool on the surface (see Figure 3) or by shifting the tool onto the surface (see Figure 4). Our implementation of this *printing* action will be discussed in section 4.3.

A crucial feature during the design process is surface appreciation,

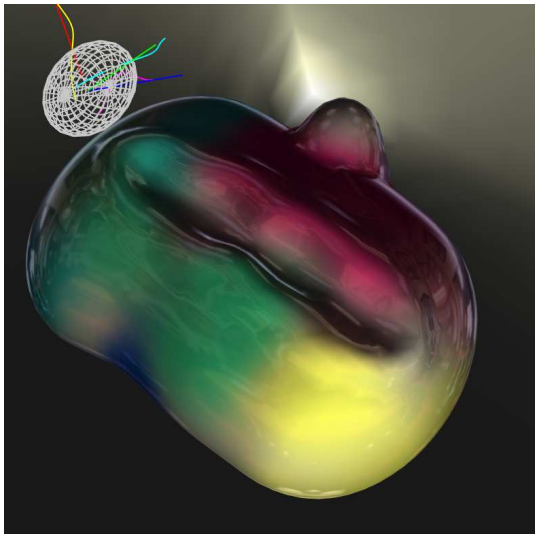


Figure 4: Print made by shifting an ellipsoidal tool

to verify at each modeling step that the resulting shape fits our inten-

tion. One possibility to perceive the surface shape is to use different lighting conditions. Here, the user can shift the lights to emphasize some shape features. He can make these light sources either move with the scene, i.e. remain fixed relatively to the sculpture, or move with the viewpoint. The user can also edit the lights color and the sculpture's specular properties. Though this *direct shading* may suffer from artifacts due to the surface triangulation (a well-known drawback of the marching-cubes process used to extract the iso-surface), it enhances the perception of the sculpted shape. Another feature that helps the user to understand the surface's shape is environment mapping. The user can cycle through different modes: no environment texture, opaque texture, semi-transparent texture, and angle dependent transparent texture. An opaque texture completely masks the object colors, and gives it a metallic aspect. The transparent texture imitates a paint layer and gives the object a plastic effect. Both will be discussed in section 5.
The user can also exploit stereo rendering to further enhance his perception of the scene. Our current implementation enables him to use either a head-mounted display (iGlasses from Virtual-IO) or shutter glasses (Crystal Eyes from Stereographics). Our stereo settings will also be detailed in section 5. Though they are very simple at the moment, it greatly improves the scene comprehension.
Finally, it's possible during the sculpting process to save a screen snapshot, or export the triangulated surface as an Inventor object, or save the scene in an internal format which writes the whole field in a file.

3 Discrete potential field storage

We considered the physical size of the sampling grid directly implemented as an array[8, 18, 2] as a limitation: not only because it encloses the model and limits its extension, but also because it wastes memory storing irrelevant sample points where no potential field is defined. We decided to use here a dynamical structure to only store these *relevant* sampling points.
Hash-tables would have been very difficult to use because we have no *a-priori* knowledge of the spatial distribution of the points to store. So it would have been very difficult to design an efficient hash-key: getting closer to the worst case point retrieval would tend to a list-search, and would become very inefficient.
We decided to use balanced binary search trees which are less efficient than hash-table in best case, but have better operation time in worst case configuration.

We firstly describe our data structures, giving the aim of each structure. In section 3.2 we explain how we use these structure, i.e. how we update them and how they interact together.

3.1 Data structures: static description

We call these regularly spaced points that sample the potential field `Corners`. Each of them stores 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). We call the balanced binary search tree we build from them `CornersTree`. `Corners` possess a key, so that they can be compared to build the tree. The key we use is simply made up of the indices (i, j, k) of the `Corner` in the virtual grid implicitly defined by the regular space sampling. Two keys (i_1, j_1, k_1) and (i_2, j_2, k_2) are compared in lexicographical order, which means that we first compare i_1 and i_2. If they are equal, we then compare j_1 and j_2, and finally if j_1 equals j_2, we compare k_1 and k_2.
Each `Corner` having a value between an arbitrary `minVal` and `maxVal` (arbitrarily set to 0 and 3 in our case) is stored in the `CornersTree`. A corner whose value becomes less than `minVal` is removed from the tree and deleted. Values above `maxVal` are clamped to `maxVal`. When requesting the value of a `Corner` not present in the tree, the returned value is `minVal`.

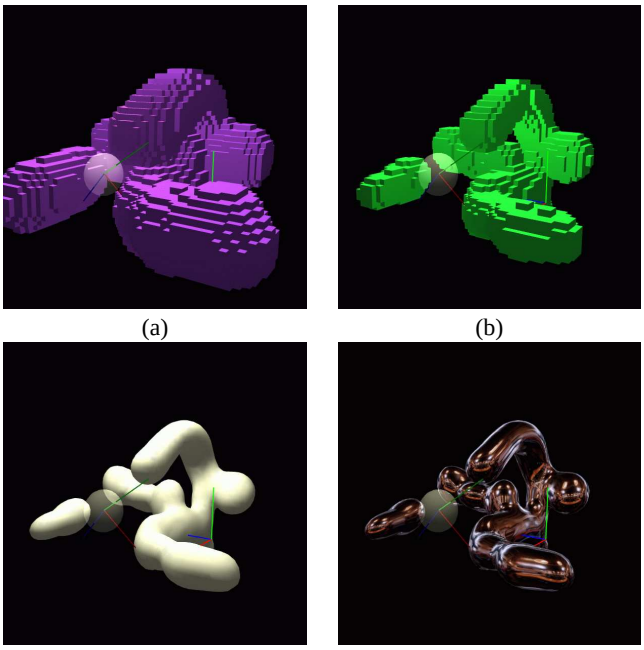


Figure 5: Visualization of some data structures elements: (a) shows all the cubes having at least one corner defined. (b) shows all the cubes that cross the iso-surface. (c) shows the extracted iso-surface. (d) uses environment texture to improve the shape perception

This regular space sampling divides the space in cubical elements we name `Cubes`. In a similar manner, each `Cube` having at least one `Corner` defined is stored in a `CubesTree`. The key associated with each `Cube` is a triplet (i, j, k) that corresponds to the smallest `Corner`-key of the eight `Corners` defining it. A `Cube` is made up of:

- eight pointers to its `Corners`, one of which is non-null, at least.
- an index deduced from the value of its eight `Corners` relative to the iso-value, which encodes the `Cube`/iso-surface intersection configuration. This is a classical decomposition step from the Marching-Cubes algorithm[4, 11, 5].
- twelve pointers to edges.

The `Cubes` that intersect the iso-surface (depending on their index value) are also inserted into another tree which we call `crossList`. Figure 5.a shows all the `Cubes` contained in `CubesTree`; Figure 5.b shows the cubes crossing the surface (stored in `crossList`); Figures 5.c and 5.d shows the corresponding iso-surface with different rendering modes.
An `Edge` is created only to compute and store an intersection with the iso-surface. `Edges` are stored in another balanced binary tree, ordered with a key which is the concatenation of the two `Corners` keys forming the edge (with the smallest `Corner` key first to ensure uniqueness of `Edge` keys). The `Edge` keys comparison is again achieved in a lexicographical order.

3.2 Applying a tool: data structures update

When a tool is applied, we have to flush its modification in the `CornersTree`. To this end, we first compute the axis-aligned (grid-aligned) bounding box surrounding the local tool bounding box. Then, we have to walk through this box by:

1. transforming from world to local (tool) coordinate only the two extremum points of the box (P_{min} and P_{max}) and the three displacement vectors (that move from one `Corner` to the next in each axis direction).

2. starting from the P_{min} point, we can reach the next point simply by adding its displacement vector to its current location, and similarly adding its counterpart displacement vector to its counterpart location in local (tool) frame coordinate (see Figure 6). Note that any scaling can be applied to the tool by applying the inverse scaling to the local location we just obtained.

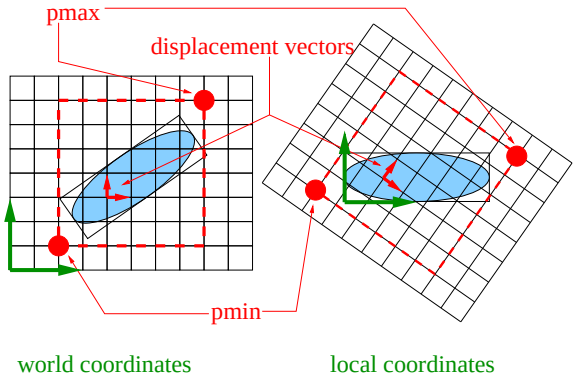


Figure 6: Parallel bounding box walkthrough is conducted both in world and local (tool) coordinates. Only the two points P_{min} and P_{max}, and the three axis-aligned displacement vectors are transformed from world to local coordinates.

For each `Corner` examined during this walkthrough, we distinguish three cases (see Figure 7.a):

1. the `Corner` is in the world bounding box, but outside of the local bounding box. It can be very quickly rejected, since the bounding box containment is a very rapid test in the local frame coordinate. We call these `Corners` the **visited** `Corners`. They appear light grey in Figure 7.

2. the `Corner` is inside the local bounding box, but outside the tool's influence (i.e. the tool's potential field has a null contribution at this point). To identify this case, we must compute the tool's potential value for that point; we call them the **computed** `Corners` (meaning that we computed the tool's potential field, but finally the `Corner` wasn't modified). They appear in grey on Figure 7.

3. the last category concerns the `Corners` whose value was effectively modified. We call them the **dirtied** `Corners` because they have to be updated (cleaned). They appear in dark grey on Figure 7.

All these **dirtied** `Corners` are inserted into a temporary tree called `modified`.

Each time a redisplay is needed, we successively extract (pop) every `Corner` from the `modified` tree. For each `Corner`, we

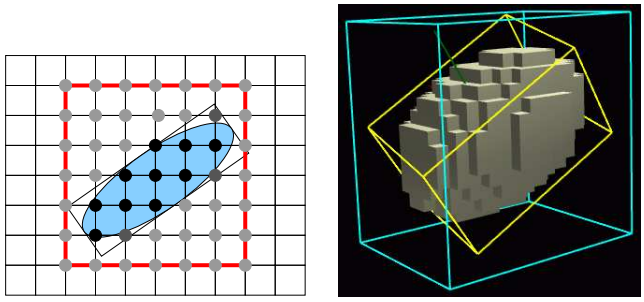


Figure 7: Applying the tool. (a) represents in $2D$ the virtual grid. Light-grey points represent the `Corners` **visited**. Grey points represents the `Corners` **computed**. Dark-grey points represent the `corners` **dirtied**. (b) shows the axis-aligned (blue) and oriented (yellow) tool bounding boxes. The cubes displayed are the `Cubes` which possess at least one `Corner` that has non-null contribution from the tool.

then have to update the eight `Cubes` that share it. To avoid multiple `Cube` examinations, we used either a timestamp-mechanism or another temporary tree to store only once each *dirtied* `Cube`.
To examine a `Cube` we compute the index (i.e. a bitmask deduced from the `Corners` value relative to iso). If the `Cube` doesn't cross the iso-surface, we're finished with it. If it crosses the iso-surface, its index corresponds to a surface crossing configuration in a precomputed table (classical part of the Marching Cubes process[4, 11, 5]). This configuration tells us which `Edges` of the `Cube` are intersected. The corresponding `Edges` are then updated.

When an `Edge` is updated (or created), the field gradients of its two extremity `Corners` are first (re-)computed (with a central difference scheme in our current implementation). Then, the intersection point is obtained by linearly interpolating each `Corner` attribute (such as the location, gradient and color) weighted by the corresponding potential field value stored. The interpolated gradient serves as surface normal (and is consequently sent to the graphic hardware).

3.3 Undo/redo handling

One key feature to encourage creative explorations is to allow multiple successive tries: the user can experiment whatever he desires without any consequence because he can always return to an earlier configuration.
We achieve the undo/redo process via temporary *undo-files*: each time a tool is applied, we dump all the modified `Corners` into a new *undo-file*.
In our current implementation, dumping a `Corner` corresponds to:

1. writing its indices in the virtual grid (i.e. the triplet (i, j, k) relative to an arbitrary origin, and an arbitrary step size, arbitrary meaning here semantically empty, as it has no physical meaning/size/dimension).

2. writing its previous value and attributes (color only in our case, the other attributes such as the location and gradient are simply caches, and can be computed).

3. writing its new value and color after modification.

We arbitrarily limited the number of undo-files to 200. When more than 200 modifications are conducted, we cycle through the existing undo-files (restarting from the beginning). As the filesystems'

caching are sufficiently efficient to achieve these dumps at interactive rates (both under IRIX6 and WindowsNT4), the real limitation to this undo/redo process is only disk space.

4 Sculpting tools

A tool is defined by:

- a **potential field**, that defines what we called the *tool contribution* in space. The tool bounding box is in fact the bounding box of this field. This potential field also indirectly (implicitly!) defines the tool's shape, which corresponds to an iso-value of the field.
- an **action**, that defines the way the tool's potential field is combined with the (possibly) existant object's potential field.

We will in the next section explain what are the possible shapes/potential fields for our tools, focusing also on a particular kind of tool we firstly used. In the next section, we go through the possible tool's actions. We describe how we implemented the classical tool (i.e. the actions already proposed in the previous approaches[8, 18, 2]). Then, we present our method to achieve local deformations that make prints on an existing object.

4.1 Tool shape

4.1.1 Ellipsoidal tools

During our implementation process, we first developed ellipsoidal tools. We simply used either a classical *Wyvill*[19] potential function or a box-shaped function to generate an isotropic (spherical) continuous field around the tool center (see Figure 8). A general ellipsoid is obtained by scaling the tool along its three axes. In

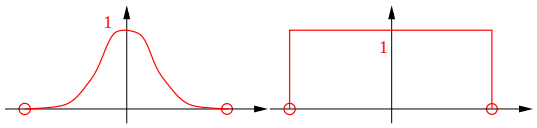


Figure 8: Continous field functions used for the ellipsoidal tool

this particular case, the shape displayed to represent the tool in the workspace is the limit of the influence region.

4.1.2 Freeform tools

We also used the freeform shapes generated inside our application as generalized tools. To achieve this, we duplicate the trees storing the corresponding discrete potential field. The surface displayed corresponds to the iso-surface, which is the same as the one visualized during its design process. The tool can also be scaled along the three axes of its local frame coordinate.

Since applying the tool requires the evaluation of its potential field between its samples' location, we *reconstruct* a continuous potential field by tri-linear interpolation inside the `Cube` the point falls into: as the `Cube` stores pointers to its `Corners`, we save time to retrieve their value (as we avoid the whole `CornersTree` search for each of the eight values).

4.2 Classical tool actions

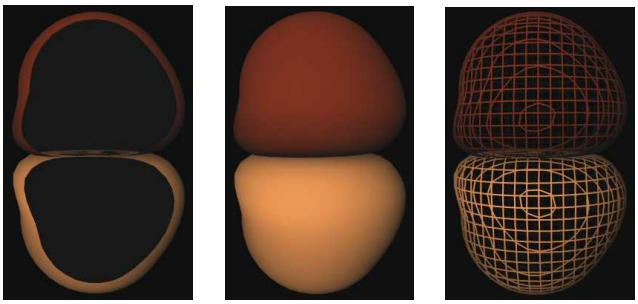
The classical tool's actions are, similar to the concepts presented in [8, 2]:

- deposit material, which means that we **add** the tool's contribution to the (possibly) existing sample point (a `Corner`, for instance).
- remove material, either smoothly (which means that we substract the tool's contribution to the `Corners`) or not (which means that we delete all the `Corners` where the tool has a non-null contribution).
- paint material, again either smoothly or not, depending on if we take into account the respective field value of the `Corner` being examined to the tool's contribution, or not.
- smooth the shape being modeled, which is indirectly conducted via the local field smoothing. This corresponds to a low-pass filtering operation.

Our implementation of these various tool's actions is very similar to the previously proposed ones[8, 2].

4.3 Local deformation tool: use the tool as a stamp

Our aim here is to produce a visually convincing deformation while avoiding the computation cost and stability problems of a physical simulation of the material displacements. Our method is inspired from an approach developed for classical (continuous) implicit surfaces[13]. It consist in applying a negative field to compress the object in the area where another object penetrates, while creating a bulges to imitate material displacement around this area. The images in Figure 9 were obtained with our implementation of this method inside the iMAGIS-team implicit modeler: *Fabule*[15].



(a) (b) (c)

Figure 9:

Though we keep the same underlying ideas:

1. direct use of a bulging potential field to achieve *geometric deformations* without any physical simulation.
2. use the composite of the potential field of the colliding object (the tool in our case) and an *ad-hoc* deformation function to produce the bulging potential field.

we use a slightly different deformation function, and our use of discrete potential field allows us to obtain plastic deformations which were impossible with the original approach.

Concerning the first point, our deformation function comes from simple intuitions: inside the tool, we should remove some material; outside the tool, we should add some material in order to imitate the real material displacement that occurs when a real tool collides with a block of clay. As this inside/outside knowledge is already encoded in the potential field (value greater than iso meaning inside

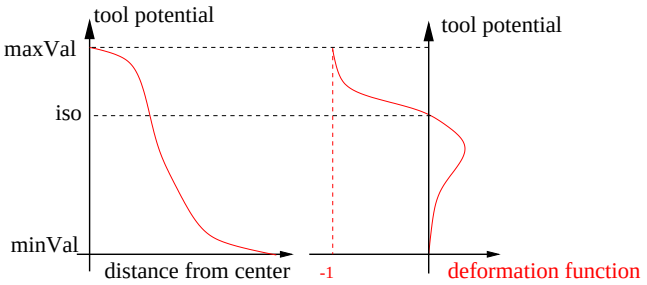


Figure 10: Deformation function principle. On the left, we see the tool potential generated by a spherical tool: this potential varies with the distance from the center of the tool. On the right, we see how the deformation function is mapped onto the tool's potential. Composing the two function relates the deformation function to the distance from the center of the tool.

in our case; the lower the potential value, the farther away from the iso-surface), we use the potential field of the tool as input parameter for our deformation function (see Figure 10).
The deformation function parameterization we are currently using appears in Figure 11 (see Figure 12 to identify parameters):

$$def(v) = \begin{cases} \text{if } (v<=s) & -\frac{(s.k+2)}{s^3}.v^3 + \frac{(s.k+3)}{s^2}.v^2 - 1 \\ \text{if } (v<=s+m) & -\frac{(k.m+2.n)}{m^3}.v_0^3 + \frac{(2.k.m+3.n)}{m^2}.v_0^2 - k.v_o \\ & \text{with: } v_0 \leftarrow v\text{-}s \\ \text{if } (v<=s+m+r) & -\frac{2.n}{r^3}.v_0^3 - \frac{3.n}{m^2}.v_0^2 + n \quad \text{with: } v_0 \leftarrow v\text{-}s\text{-}m \\ \text{if } (v>s+m+r) & 0 \end{cases}$$

Figure 11: Equation of our deformation function

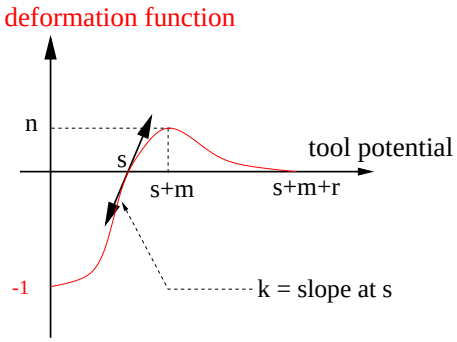


Figure 12: Tunable parameters of the deformation function

We then set parameter s to our iso-value, and control that the sum $s+m+r$ doesn't bypass $maxVal - minVal$, otherwise the domain of definition of the deformation function becomes greater than the potential value variation: the deformation function would be truncated.

To sum up, we evaluate the contribution of a tool at a point P by first computing the value $v_{tool}(P)$ of the tool's potential field at P. The final tool's contribution is the composite of v_{tool} and def:

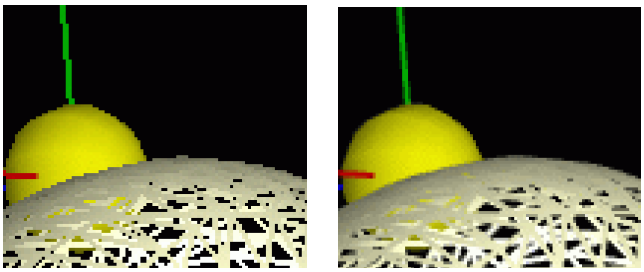
$$v_{deformTool} = def(v_{tool}(P))$$

Some results can be seen in Figures 3 and 4.

5 Visual feedback

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 (see Figure 13).
We also tried some stereo rendering using some *Stereographics*



(a) (b)

Figure 13: We see a close-up of a snapshot from our system. (a) without the multisample extension and (b) with.

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. We were first guided in that direction by some methods using textures to simulate high quality highlights. This reveals particularly useful if the surface had degenerated triangles(see Figure 14), which is a typical drawback of the Marching Cubes algorithm. We used classical sphere-mapping with adjustable

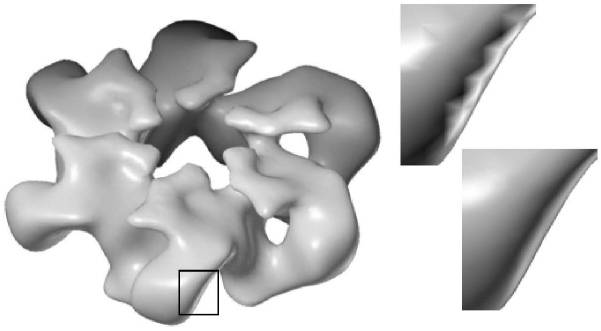


Figure 14: Sample shading artifacts due to the object triangulation, and a proposed solution based on textures to render high quality highlights of infinitely distant light sources (image from [12], pp.273)

transparency to be able to see the surface color under the texture layer. We also implemented simplified *ClearCoat*[1] like effect,

[1]information concerning SGI's *ClearCoat* product may be found at `http://www.sgi.com/software/clearcoat/`

i.e. simulating a paint layer, using a texture transparency varying accordingly to the incidence angle between the viewer and the surface.

6 Performances and Results

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

Galyean[8] used grids from 30^3 up to 60^3, while Avila[2] reports the use of grids up to 256^3. Pfister[14] 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 2. 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) and 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.

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.

7 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].
An important limitation in our current implementation is the fixed spatial sampling resolution of potential fields. 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.

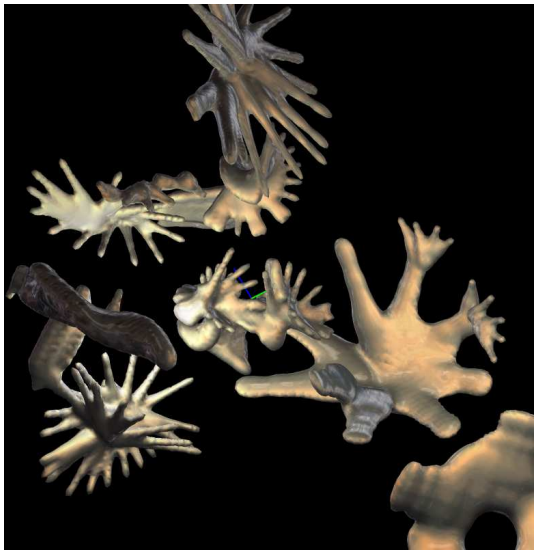


Figure 15: Sample sculpture produced with one finger-shaped tool we built during a previous session. This was produced in less than ten minutes, during a debugging stage.

Figure 16: A clown character. He was entirely built using the ellipsoidal tool on an SGI O2 workstation. This was achieved in around one hour and a half including lots of trial and errors.

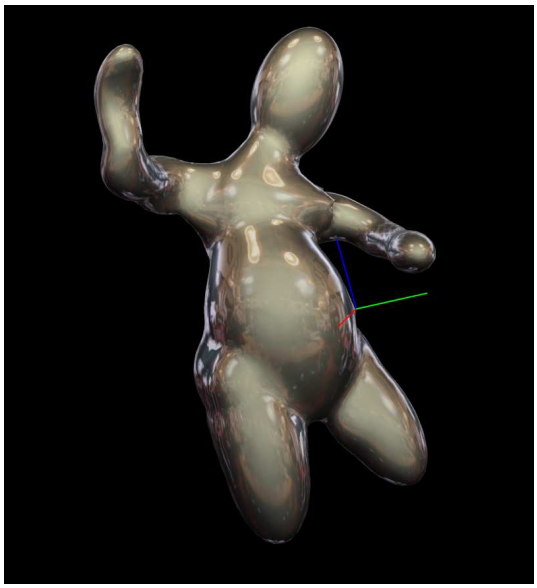


Figure 17: The same clown at an earlier stage

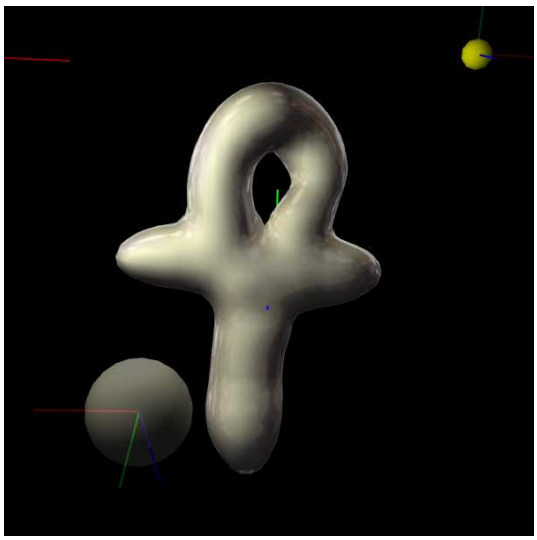


Figure 18: Design of a new tool using the ellipsoidal tool

Figure 19: A more complicated shape modeled in less than a quarter of hour using the previously designed tool at different scales.

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 dependent environment textures. We would also like to thank the many people who contributed to develop GLUT, and Paul Rademacher for providing GLUI[2]. Both are really helpful to develop cross-platform OpenGL-based application.

We would also like to thank the reviewers for their help in making this article looking more *english-english* than *french-english*, and for their valuable remarks.

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.

[3] J.-R. Bill and S.K. Lodha. Sculpting polygonal models using virtual tools. In *Graphics Interface'95*, pages 272–278, Quebec, Canada, May 1995.

[4] J. Bloomenthal. Polygonization of implicit surfaces. *Xerox Technical Report CSL-87-2*, pages 1–19, May 1987.

[5] J. Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[6] Y.-H. Chai, G.R. Luecke, and J.C. Edwards. Virtual clay modeling using the isu exoskeleton. *Proceedings of VRAIS'98*, March 1998.

[7] D.R. Forsey and R.H. Bartels. HierarchicalB-spline refinement. *Computer Graphics*, 22(4):205–212, August 1988. Proceedings of SIGGRAPH'88 (Atlanta).

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

[8] 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).

[9] W.H. Hsu, J.F. Hughes, and H. Kaufman. Direct manipulation of free-form deformations. *Computer Graphics*, 26(2):177–184, July 1992. Proceedings of SIGGRAPH'92 (Chicago).

[10] T. Igarashi, S. Matsuoka, and H. Tanaka. A sketching interface for 3d freeform design. *Computer Graphics*, pages 409–416, August 1999. Proceedings of SIGGRAPH'99 (Los Angeles).

[11] W.E. Lorensen and H.E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *Computer Graphics*, pages 163–169, July 1987. Proceedings of SIGGRAPH'87 (Anaheim).

[12] T. McReynolds, D. Blythe, B. Grantham, and S. Nelson. Advanced graphics programming techniques using opengl. *Computer Graphics*, July 1998. SIGGRAPH'98 Course Notes #17.

[13] 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/`.

[14] 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).

[15] D. Rossin, M.-P. Cani-Gascuel, J.-D. Gascuel, A. Opalach, and M. Desbrun. Plateforme d'expérimentation pour la modélisation par surfaces implicites. September 1997. Proceedings of Modeleurs Géométriques'97 (Grenoble).

[16] J.-A. Thingvold and E. Cohen. Physical modeling withB-splines surfaces for interactive design and animation. *Computer Graphics*, 24(4):129–137, August 1990. Proceedings of SIGGRAPH'90 (Dallas).

[17] T.I. Vassilev. Interactive sculpting with deformable nonuniform b-splines. *Computer Graphics Forum*, 16(4):191–199, 1997.

[18] S.W. Wang and A.E. Kaufman. Volume sculpting. *Computer Graphics*, pages 151–156, 1995. Proceedings, Symposium on Interactive 3D graphics.

[19] B. Wyvill, C. McPheeters, and G. Wyvill. Animating soft objects. *Visual Computer*, 4(2):235–242, August 1986.

[20] R.C. Zeleznik, K.P. Herndon, and J.F. Hughes. Sketch: An interface for sketching 3d scenes. *Computer Graphics*, pages 163–170, August 1996. Proceedings of SIGGRAPH'96 (New Orleans).

Practical Volumetric Sculpting

Eric Ferley Marie-Paule Cani Jean-Dominique Gascuel
iMAGIS/GRAVIR-IMAG

Figure 3.(a)

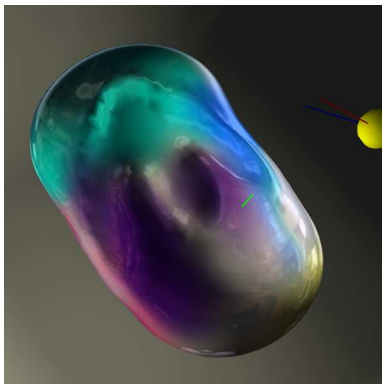


Figure 3.(b)

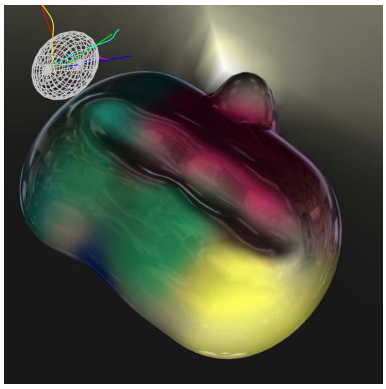


Figure 4

Figure 16

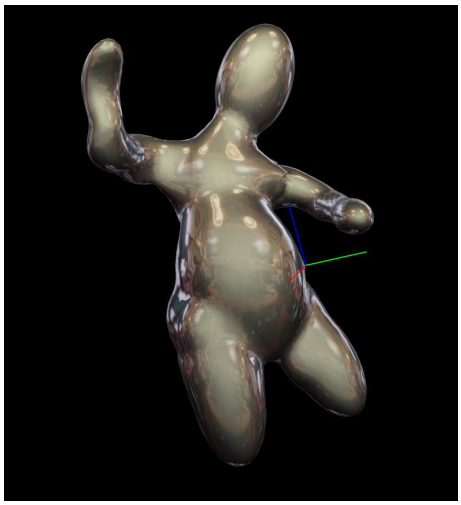


Figure 17

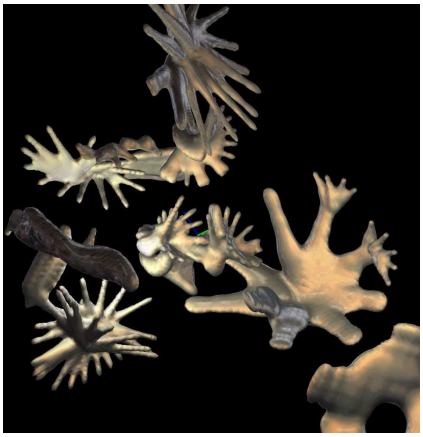


Figure 15

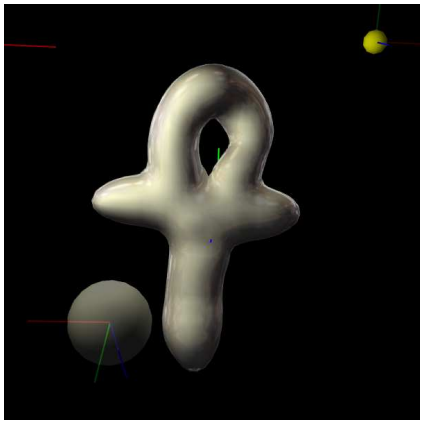


Figure 18

Figure 19