Resolution Adaptive Volume Sculpting

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

iMAGIS/GRAVIR-IMAG,
a joint research project of CNRS/INRIA/UJF/INPG,
INRIA Rhône-Alpes ZIRST, 655 avenue de l'Europe,
Montbonnot, 38334 Saint Ismier Cedex, France.

E-mail: Eric.Ferley@imag.fr, Marie-Paule.Cani@imag.fr, Jean-Dominique.Gascuel@imag.fr

We propose a sculpture metaphor based on a multiresolution volumetric representation. It allows the user to model both precise and coarse features while maintaining interactive updates and display rates.

The modelled surface is an iso-surface of a scalar-field, which is sampled on an adaptive hierarchical grid that dynamically subdivides or undivides itself. Field modifications are transparent to the user: The user feels as if he were directly interacting with the surface via a tool that either adds or removes "material". Meanwhile, the tool modifies the scalar field around the surface, its size and shape automatically guiding the underlying grid subdivision.

In order to give an interactive feedback whatever the tool's size, tools are applied in an adaptive way, the grid being always updated from coarse to fine levels. This maintains interactive rates even for large tool-sizes. It also enables the user to continuously apply a tool, with an immediate coarse-scale feedback of the multiple actions being provided. A dynamic Level-Of-Detail (LOD) mechanism ensures that the iso-surface is displayed at interactive rates regardeless of the zoom value; surface elements, generated and stored at each level of resolution, are displayed depending on their size on the screen. The system may switch to a coarser surface display during user actions, thus always insuring interactive visual feedback.

Two applications illustrate the use of this system: Firstly, complex shapes with both coarse and fine features can be sculpted from scratch. Secondly, we show that the system can be used to edit models that have been converted from a mesh representation.

Key Words: volumetric sculpting, implicit surfaces, multiresolution, hierarchy of uniform grid, multiprocess, multithread

1. INTRODUCTION

Sculpting with direct 3D interaction has attracted much attention in the past few years. These approaches provide the user with direct interaction with a 3D model via a tool linked to a 3D input device; this may range from a virtual trackball attached to a 2D-mouse to a force-feedback articulated arm (see Figure 1). Mid-range input device include 3D-mice, such as a Spacemouse or a Spaceball. The user perceives the sculpted object on a classical screen with or without stereovision, or with semi-transparent stereo-glasses. The targeted user of these systems should already be familiar with 3D interaction such as clay-modelling or real-sculpting. This is contrary to users of 2D metaphors such as Teddy [10], where 2D curves are used to infer a 3D shape.

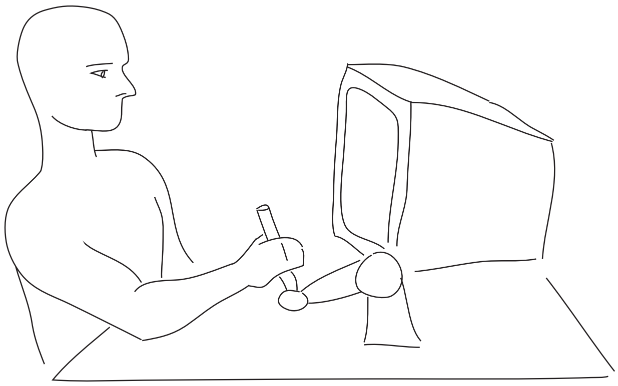


FIG. 1. Principle of operation

Hiding the underlying representation is essential in a 3D sculpting system. The user should be able to focus his attention on the shape being modelled rather than on its mathematical, internal representation. This is not the case in traditional, parametric approaches used in CAGD (e.g. the use of a control mesh to edit NURBS or subdivision surface representations). In addition, one of the key features we are seeking is the ability to transparently handle connection and dis-connection of model-parts. To this end, volumetric representations such as implicit surfaces[1] appear very well suited.

Handling topological changes is not the only advantage of implicit surfaces in the context of an interactive sculpting system : they also ensure a *correct* definition of a closed surface which always possesses a well defined interior and exterior. Classical implicit surfaces modelling mostly uses primitives (*skeletons*) that generate a scalar field from which an iso-surface is extracted. These scalar fields can then be combined in various ways, the most basic one being summation. A drawback of this constructive approach is that the cost of field evaluation grows with the number of primitives. If each user action results in a primitive creation, the field evaluation rapidly becomes prohibitive and forbids interactivity. Primitive sorting

[1] An implicit surface modelling review is beyond the scope of this paper. The interested reader may find a good introduction in [4].

optimization, such as those proposed by A. Sourin [14] in the case of metal embossing (which requires less primitives than modeling a 3D shape), or other techniques to merge and simplify the primitives could be explored to solve this problem. We rather use a straightforward solution that consists in directly storing a discrete, sampled representation of the scalar field. Various papers have already proposed similar *Discrete Scalar Field* approaches. We discuss them in the next section, before describing our contribution.

1.1. Modelling with discrete scalar fields: Previous work

Interactive modelling based on discrete scalar field representation was early introduced in 1991 by T. Galyean and J. Hughes [9]. The field was stored on a regular 3d grid (*voxmap*). The tool used to edit the field was also discretized and particular attention was paid to prevent aliasing when the discrete tool was re-sampled into the field grid. Available tool actions included adding or removing *material*, and smoothing the surface through a convolution applied to the 3D field.

In 1995, S. Wang and A. Kaufman [15] extended the interaction to carving using tools deduced from a pre-generated 20^3 volume raster or sawing (extruding) via curves drawn onto the screen.

The following year, R. Avila and L. Sobierajski [1] used a force feedback articulated arm to command the tool in a similar context. The very rapid update rate required limited the tool size to 3-5 voxels.

We also developed a sculpting systems based on a similar methodology [6, 7]. We used a hashing structure to store a regular virtual grid, in order to allow the user to add material without any limitation in space. Other available operations were material removal, and local deformations modelling contact with a rigid tool. We paid very little attention to aliasing, as we considered it as a natural sampling limitation due to the fixed grid resolution. We suggested multi-resolution or adaptive sampling of the field as the adequate solution to the problem. This paper describes this solution.

Surprisingly, a huge amount of contributing papers deal with multiresolution volumetric data or adaptive discretization of implicit surfaces but few multiresolution approaches have been applied in the context of volumetric modelling.

In 1998, J. Bærentzen [2] proposed an *octree-based* volume sculpting system. Used to accelerate ray-casting rendering, the octree is unfortunately static: the subdivision is limited to, and always reaches a fixed *leaf* level, yielding a regular sampling solution very close to the grid used in [9]. Dynamic *leaf-node* management to preserve memory in regions of low details or to increase resolution in highly detailed regions was left as a future work.

More recently, A. Raviv and G. Elber [13] proposed a different hierarchical approach based on scalar tensor product uniform trivariate B-Spline function. A collection of B-Spline patches with arbitrary position, orientation and size is used to represent the scalar field. The user can create patches and select an active patch to edit with a tool that modifies its scalar coefficients. An additional octree structure is used to sample the collection of patches and conduct a Marching Cubes to extract the iso-surface. The octree resolution is guided by the underlying patches size.

K. McDonnell, H. Qin and R. Wlodarczyk [11] recently presented another interesting approach, based on subdivision solids and surfaces. Three distinct kinds of tools are presented: haptic, geometric and physically-based; each addressing different features of the model. The paper claims direct physics-based interaction with the sculpted surface to transparently handle the internal subdivision schemes. However, the user still has to explicitly edit control cells to change the topology of the model and to edit mass-points to change its physical behavior. The haptic-based deformation also requires to set a spring connection between the selected mass-point to the tool's cursor.

More recently, Frisken and al. [8] proposed a resolution adaptive volumetric approach, named ADF which stands for Adaptively Sampled Distance Field. The basic idea is to use the euclidean distance to a given surface and adaptively sample into a discrete scalar field. Field recomputations due to surface editing are performed either by starting at the bottom of the hierarchy and then performing a simplification pass (bottom-up strategy) or by refining the discrete field where necessary (top-down strategy). Their last paper [12] addressed some limitations of the method such as improving the update rate. In order to maintain interactivity, the update is performed in priority at the neighborhood of the surface. Then distance modifications propagate during idle moments.

This approach presents some similarities with our method, though they were developped totally independently. Both of them exploit the idea of adaptive, multiresolution volume sculpting. However, major differences can be observed in the way the information is stored and sampled in the 3D field. Instead of storing the euclidean distance to the modelled surface, we store a *field function* (similar to the ones used in implicit surface modelling). This field function is the composition of the euclidean distance with a *potential function* that smoothly decreases to zero as the distance grows. This formulation presents several advantages: not only the potential acts as a filter that smoothes the distance, but, it also bounds the region of influence of editing actions. Consequently only local field updates are required. In contrast, the region where the euclidian distance changes after surface editing is unbounded. So even local editing may produce global field updates in Frisken's approach. Another difference is the subdivision strategy. Frisken stops subdividing when the difference between the distances computed by two consecutive levels of the hierarchy is under a given threshold. With this method, subdivision reaches the bottom of the hierarchy on every distance discontinuity. Such discontinuities occur near each surface concavity, and their spatial extend can be unbounded. Athough subdivision is restricted to cells that cross the surface during edition [12], subdivision in those discontinuity regions may propagate at idle moments. This can lead to over-sampled regions that can be located far away from the surface, severely impacting computational time and memory requirement.

1.2. Overview

This paper presents a 3D sculpture system that enables interaction with a sculpted object, at any modelling scale, without having to be concerned with the underlying mathematical representations. The modelled surface is an iso-surface of a scalar-field. The field is stored in a hierarchical grid with a given subdivision factor. The extent of the structure, in terms of space and resolution, is fully dynamic; it is driven by the actions of the user and has no size limitation of maximum depth. This system allows for precise, interactive direct modelling through addition and removal of material. As the underlying scalar field representation is completely hidden from the user, the use of the system is intuitive, and gives the feeling of direct interaction with the sculpted surface.

Our main contributions are: (1) an adaptive subdivision of the field guided by the tool; (2) a progressive update of the hierarchical field using priority queue mechanisms which allow the use of large tools at interactive rates and to continuously apply a tool; (3) an interactive rendering that directly exploits the hierarchical field representation to locally conduct a Marching Cubes surface extraction and limit the display complexity at each frame according to the camera position.

The remainder of the paper addresses these three points in Sections 2 to 4. Section 5 then shows our results in two different contexts: the direct creation of a complex sculpture and using an editable model that has been converted from a mesh representation. Section 6 presents conclusions and future research directions.

2. TOOL GUIDED ADAPTIVE SUBDIVISION

Our sculpting system lets the user interact with a tool that modifies a scalar field. Our convention is to represent the interior of the sculpted object using positive field values and to have a null field where no *material* exists. This enforces the analogy with the real world, the scalar field being similar to a material density. The displayed surface evolves with a given iso-value of the field, creating the illusion of direct interaction with the object's surface.

Clamping field values so that they lie between a minimum and a maximum value is necessary in this representation. Indeed, as the tools' contributions are applied in a cumulative manner, successive subtractive tool applications would result in a negative field which would artificially prevent new material creation in this region. Symmetrically, successive additive applications would artificially prevent material removal. In our particular implementation, the scalar field is clamped between 0 and 3, the object's surface being defined by the iso-value 1.5.

This section details the constraints we have on the field representation and discusses our choices.

2.1. Data structures

2.1.1. Hierarchical scalar field

The most straightforward, easily edited, scalar field representation consists in sampling the (supposed continuous) field on a *regular grid basis*. Samples of field values are stored, along with other attributes such as the field gradient and color, in a *Vertex* structure (see top-left of Figure 3). This can be done without spoiling the field's apparently unlimited spatial extent using a hashing structure to store the grid as in [7]. However, this fixed resolution framework limits the kind of shapes that can be modelled: when a fine grid is used to represent details, surface modification at a coarser scale requires editing too many samples, thus preventing interactivity.

We extend here this approach with the ability to locally refine the sampling rate, such as illustrated in Figure 2. In order to preserve the multiresolution representation, the left cell isn't replaced by the sub-cells set, but is rather enriched with it:

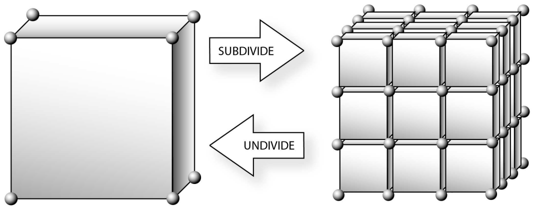


FIG. 2. Subdivision principle. Many choices are left, such as replacing the left *Cell* by the subdivided *Cells* on the right or referencing the subdivided *Cells* as children of the left *Cell* (i.e. enriching the left *Cell*), duplicating or not the *Vertex* elements that have the same position, ...

the sub-cells are its *successors* or *children*. As we do not want to restrict the user to any resolution limit (as well as not restrict the extent of his model in space), we need to allow dynamic creation or deletion of such successors sets. This *a priori* precludes the use of most classical octree storage optimizations. So, we rather try to reduce the structure overcost by allowing a direct jump to much finer resolutions. We subdivide space by a constant factor n on each dimension ($n = 4$ on Figure 2). This *ntree* structure resembles the *Recursive Grids* used in bucket-like space partitioning data structures [5]. Figure 3 (left) gives an outline of the ntree *Cell* structure.

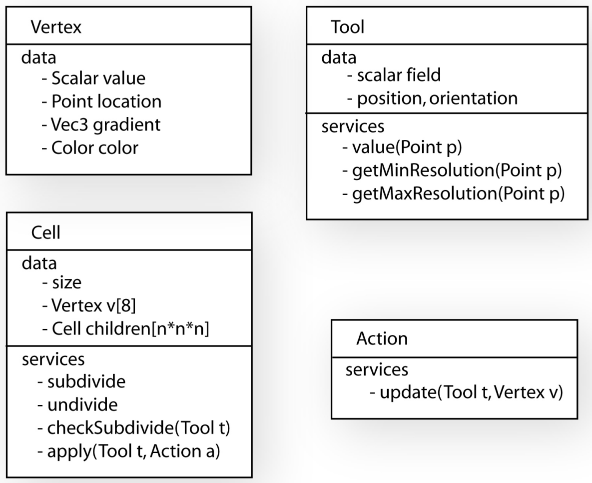


FIG. 3. Data-structure used for the field representation (left) and for applying a tool that modifies the field (right).

The next point to discuss is whether to: (1) express the samples at a finer level, $k+1$ as a *delta* contribution over the average or median value that would be stored at the coarser level k; (2) store directly the field value in each sample, thus using simple subsampling for coarser levels.

Solution 1 appears more elegant, as it looks like a wavelet decomposition of the 3d scalar field. However, it yields the extra cost of maintaining the hierarchy coherency. Coarser levels would need to be updated when detailed modifications are conducted on smaller levels, in order to recompute the average or median field's values.

Solution 1 also suggests that large changes on the low-resolution levels could effortlessly be reflected on the higher ones: storing some min-max information along the hierarchy would allow rapid pruning of the volume parts that become completely outside or inside of the modelled shape. However, the hierarchy exploration from the root node to the leaves (which is also requested in solution 2) cannot be avoided, since the surface representation has to be updated.

With the subsampling approach of solution 2, there is no need to compute the interpolated values from the coarser levels: the field value at a vertex is directly given. Moreover it allows **no duplication of** the *Vertex* nodes between resolutions. In contrast, solution 1 would force the eight corners of the *Cell* at the left of Figure 2 to be distinct from their counterparts in the sub-cells on the right because the sub-cell values define a *delta* contribution over them. Lastly, solution 2 offers a kind of *vertical independence* over the hierarchy: each level is completely independent from its ancestors, and can thus be updated independently. Therefore, we have adopted solution 2.

When subdividing the grid (i.e. during *Cell* creation), we pay special care to share the *Vertex* nodes among common faces or edges between the adjacent *Cells* of the same level. Once these shared structures are wired, their forthcoming updates won't cost more than a time-stamp comparison to prevent useless computations.

2.1.2. What is a tool?

Basically, a tool is another scalar field that can be positioned, oriented and scaled as the user wants (see Figure 3, right). It could be another hierarchical sampled field, or any bounded primitive that can return a scalar value from a given location.

The simplest example (the only one we implemented at the moment in the multi-resolution version of our sculpting system[2]), is an ellipsoidal tool. In our current implementation "material" is represented by positive field values, the field being supposed to be null elsewhere. Our ellipsoidal tool is based on Wyvill's field function:

$$f(p) = \begin{cases} \text{if } d \geq 1 & 0 \\ \text{else} & 3 * (1 - \frac{22}{9}d^2 + \frac{17}{9}d^4 - \frac{4}{9}d^8) \end{cases} \tag{1}$$

where p is the query point, and d^2 is the squared distance from the tool center to the point p, **expressed in the tool local frame coordinate**. Translation, rotation, and scaling of this local frame gives the tool's current position and shape. Using Wyvill's field function yields several advantages:

[2]The use of various tools ranging from simple primitives to user-designed tools was explored in our previous work [7]

• it is computationally cheap (it uses the squared distance, instead of its squared root);

• it has a spatially limited domain of *influence* (i.e. non-zero values);

• it is bell shaped, with C^1 variations. This allows smooth blends when different tool contributions are summed.

The tool is applied iteratively by cumulating its contributions into the sampled field. The modelled surface is the iso-surface at $iso = 1.5$ extracted from this field. As we use the limit of influence volume to display the tool's shape (i.e. the ellipsoïd outside of which the tool's influence is null), the material deposited by the tool always lies inside the tool's volume (see Figure 5 left). Continuously applying the tool at the same location progressively extends the created surface, which may eventually reach the tool's border.

2.1.3. Applying a tool

Applying the tool into the scalar field involves two important steps: (1). we must ensure that the tool is correctly sampled, i.e. the cell resolution of the field representation must be small enough to capture the tool's features and (2). for each covered vertex, we have to combine its current field value with the tool's contribution at that point.

The latter is issued using what we call an *Action* (see Figure 3, right). An action takes a *Tool* and a *Vertex* as arguments. It computes the tool's contribution at the vertex, and combines it with the current vertex attributes. For example, an *AddAction* simply adds the tool's field value to the vertex field value, uses these values as weights to sum-up the attributes such as the gradient and color, and clamps the field value between 0 and 3.

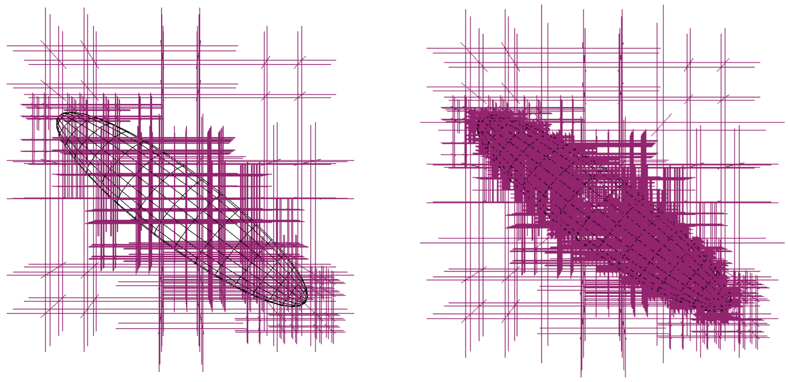
The algorithm for applying a tool to a cell of the hierarchical field representation is quite simple:

Algorithm 1 (Applying a Tool to a Cell).

```
Cell::apply(Tool t, Action a) {
    foreach Vertex v do { a.update(t,v); }
    if (I have no children) { checkSubdivide(t); }
    if (I have children) {
        foreach Cell child do { child.apply(t,a); }
    }
}
```

Now, how do we know if we need to subdivide a given cell (*checkSubdivide* test in the algorithm above) ?
Let's suppose that the tool has an ellipsoidal shape. We would like to obtain something like Figure 4, where the sampling rate increases (i.e. the cell size decreases) in regions where the tool has sharp features.

First, we might query the tool attributes for requirements on a minimal *security* cell-size to reach, in order not to miss any of its features. This could be a global in-



(a) (b)

FIG. 4. Sampling an ellipsoidal tool. (a). Only 2 consecutive levels. (b). All the levels hierarchically created.

formation constant over the tools influence region, or something locally computed. For example, if the tool field is stored as a hierarchy of cells, we could easily use the size of the leaf cell as the minimal size to reach.

Once we have reached this minimal size, the field could still be ill-sampled. For example, if we use a subtractive action, we might create discontinuities, even with spherical tools. Our choice here is to try to estimate the *discrepancy* of the field. If the cell's discrepancy is higher than a given tolerance, we go on subdividing (see Figure 5). Pragmatically, we use this strategy only on cells that are crossing the surface, as this is our region of interest (see section 4.1 for more details on the discrepancy estimator). Moreover, using this strategy in regions where no surface exists could disturb the surface when it reaches these regions; as the details existing only in the field (and consequently hidden from the user) would suddenly become visible on the surface being created.

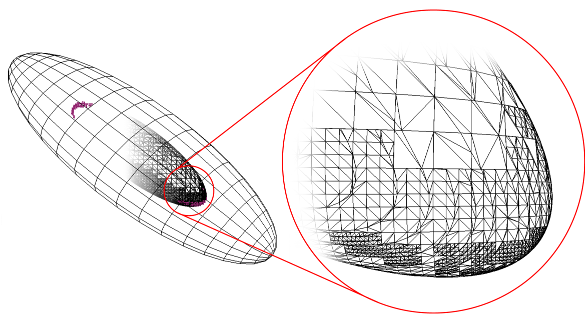


FIG. 5. Sampling an ellipsoidal tool: the large ellipsoid is the tool, and the small one inside it is the surface created. The figure shows the maximum resolution reached in highly curved areas.

As stated above, using a subtractive tool can cause discontinuities in the scalar field so the subdivision process might never end. Here again we rely on the tool to query a *maximum depth* to reach. In fact, this is formulated as a smallest cell size not to overpass, which we call the *maximum resolution.* It could, exactly as the *minimum resolution* above, be locally adapted inside the tool's region of influence. At the moment, our ellipsoidal tools only expresses it as a constant factor, depending on the tool's scale used.

This leads to the following algorithm:

Algorithm 2 (Testing a cell for subdivision).

```
Cell::checkSubdivide(Tool t) {
  if (size > t.getMinResolution() ) {
     subdivide();
  } else {
    if (size < t.getMaxResolution() ) {
      if (estimateDiscrepancy() > acceptableDispcrepancy) {
        subdivide();
      }
    }
  }
}
```

We do not have *a priori* knowledge of the field's profile before we reach the bottom level of subdivision. Thus we can not stop the subdivision to conduct any simplification ("undivide" of Figure 2) at any particular level, as we can not guarantee that the finer level will not add surface details. As a result, we conduct a separate simplification pass over the whole sampled field at idle moments of the interaction. The simplification strategy we are currently using is rather drastic, as it destroys every *Cell* that does not (and whose children do not) cross the surface.

Updating the coarser levels first, as depicted in the **apply** algorithm, is crucial to provide an interactive visual feedback. This means careful initialization of the newly created *Vertex* set. Figure 6 represents a slice of the scalar field value for a cell that is to be subdivided. Figure 6.b shows the example effect of applying a tool with an *Add Action.* The value of the two border vertices of the cell is modified with the tool's contribution. The creation of the new vertices for the sub-cells requires interpolation of the field value **prior** to the current tool application, as illustrated in Figure 6.c. so that the tool's modifications can properly be added.

We could simply use a kind of *reverse action* to obtain these values and attributes back from the current ones. Unfortunately, as the field values are clamped, the tool application is not reversible. So we have to explicitly cache the field values prior to current tool/action modification.
Our current implementation of this *Vertex cache* uses *STL* hash_map indexed by the memory addresses of the *Vertices* to map the Vertex state prior to the current tool's modification. We thus have one *Vertex cache* per couple Tool-Action application.

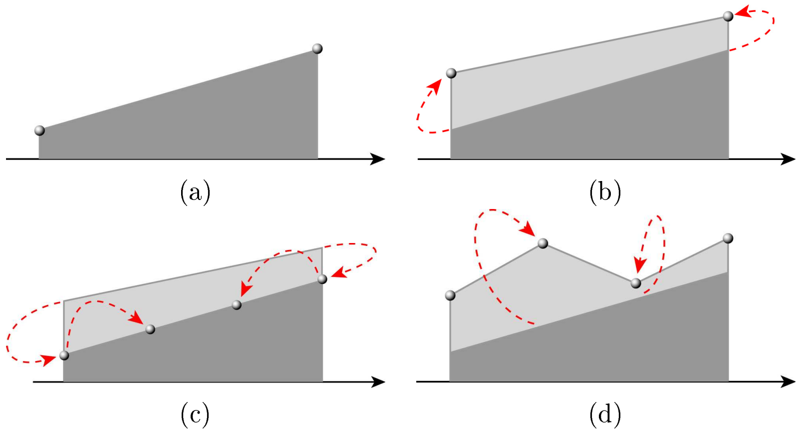


FIG. 6. Applying a tool at level k (1D representation). The figures represent the field value as a function of vertex location : (a). The field before any modification. (b). Updating vertices at level k. (c). Creating the level $k+1$ and initializing it with the values interpolated from level k **before** the update. (d). Updating vertices at level $k+1$

As a side effect, this provides almost enough information for undoing the effect of the currently applied tool.

2.1.4. Undo

Undo files[3] are directly generated from the *vertex cache* built during the field update. These files are simply a dump of the set of *modified* vertices. The memory addresses used to retrieve the vertices are not stored as the vertex might be destroyed or reallocated between the save and the reload of this undo-file.

We decided to treat *Undo* as a special *Action*. The first consequence of this is that we can easily use the current tool's contribution to *weight* our undo-action. We call this process a *Progressive Undo Action*. On the same basis, we can simply use the tool's contribution as an *in/out* selector to activate or not the undo, we call this a *Local Undo Action*. We also provide the commonly expected *Undo Action* behavior, i.e. a global undo, which is independent from the current tool being used. The second consequence of treating *Undo* as an *Action* is that, as the undo-action is handled internally as a normal tool action, it transparently generates the *redo*.

3. PRIORITY QUEUE BASED FIELD UPDATE

The recursive approach for performing field updates, described in Section 2.1.3, isn't suitable for an interactive update. Actually, it walks along the hierarchy depth first, thus missing the requirement to update coarser levels before considering finer

[3]The use of undo files does not forbid interactivity. We suspect that this may be thanks to the Operating Systems filesystem caching. Both under Irix, Linux and WindowsNT/2000, this appears to work surprisingly well.

ones. Next, we explain how to get this feature, which is essential for achieving interacive display.

3.1. From coarser to finer levels

We use a priority queue sorted on the level addressed and on the tool/action concerned to ensure an update from coarser to finer levels. The tools/actions must be sequentially applied, but we should update the coarser levels first. Thus we perform a straightforward priority evaluation based on these two criteria.

Our implementation uses a *STL* priority queue, which itself relies on a *STL* vector by default. The level is simply the cell size and the tool/action couple to apply is enclosed in a local *ToolCopy*. The *ToolCopy* is a *frozen* copy of the tool at its application time. It contains some state information (such as position, orientation, scale, color, etc), a reference to the tool and action involved and a unique identifier (an integer, acting like a time counter, see Figure 7) that also serves as a time-stamp to avoid useless computations.
The queueing internals are enclosed in a *Level Manager* (Figure 7). It initiates the application process from a *Tool/Action* couple by creating a *ToolCopy* from them and inserting it with the current root-cell's size into its priority queue.

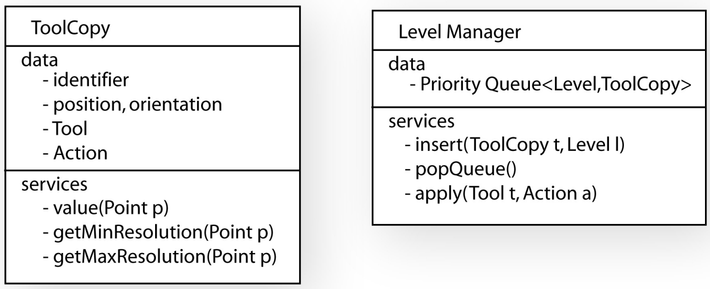


FIG. 7. Data-structure used for priority queues handling.

The *Apply* procedure outlined in 2.1.3 is not altered much. It still updates its internal vertices and subdivides if needed. Then, instead of recursively calling apply on the existing children, it simply inserts a new element made up of the same *ToolCopy* and the next level.

3.2. Emptying the queue

To empty the priority queue we need to find all the cells of a given size (or level) that are *intersected* by the *ToolCopy* (which is much like an image of the *Tool* at the moment its application was posted). Without any additional structure, this would mean recursively walking through the cells hierarchy from the root-cell until we reach the cells having the desired size. To the cost of walking from the root-cell we must add the extra-cost of the intersection test with the tool for each cells of the intermediate levels.

To avoid these useless computations, we use a simple *cell-queue* with basic constant-time operations (pushback and popfront) to temporarily store the cells intersected by the tool from one level to the next. Cell-queues are indexed by a *ToolCopy* and the size of the cells it contains. They can be directly inserted/handled inside the manager priority queue, whose elements are then the cell-queues (Figure 8).
The *Apply* procedure of 2.1.3 is again slightly modified: it receives a cell-queue as an extra parameter. Children cells that intersect the tool are appended to this queue.

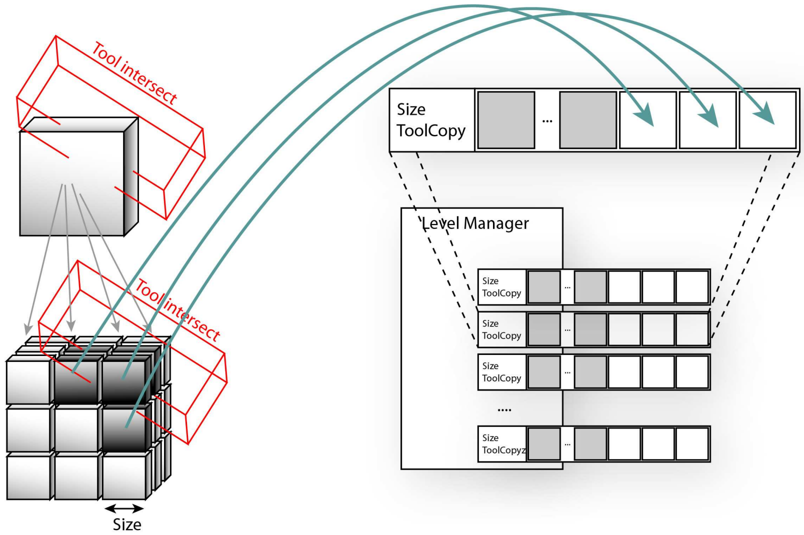


FIG. 8. The *Level Manager*: a priority queue of cell-queues.

Another benefit of these cell-queues is that they allow **interruption of** the processing of a given level if any coarser level is inserted inside the manager. The interrupted cell-queue is simply re-inserted in the manager priority queue, and is properly handled from where it was suspended when the working task returns to it.

3.3. Multi-process

A key feature to ease interactivity is to have an interaction/display thread running *separately* from the background update process. We considered and tried various multi-threaded strategies but finally a two-threads approach proved to be the most simple and efficient. It is simple because we only have one process modifying the data structure so little exclusion mechanism is needed. It is efficient for the same reason: no system call overhead or blocking/deadlock situation.

The *update-thread* is created at program start-up and empties the priority queue of the *Level Manager* in the background. The user interacts in a separate thread. Applying a tool becomes the insertion of one Cell-queue that contains only the current root-cell into the *Level Manager*. The *update-thread* then possibly interrupts its current cell-queue processing and handles normally the next sub-cells-queues insertion and processing.

The only blocking access concerns the insertion and removal of cell queues in the *Level Manager* priority queue. It is easily handled via a classical locking mechanism. It is important to note at that point that the update process gets most of its workload to empty the specific *Cell queue* it extracted from the *Level Manager* and filling one for the next level. Consequently, the access to the *Level Manager* to insert or retrieve a *Cell queue* is not really blocking as both processes do not intensely access it. Note also that the update process is the only one accessing the Cell-queues it empties or fills, as the draw process accesses the Cells through the hierarchy and not through the Cell-queues. So inside each Cell-queue no exclusion mechanism is needed.

We implemented the *threading/locking* facilities both with Posix threads, IRIX sprocs, and Windows threads with no remarkable speed impact. When the priority queue of Cell-queues to update becomes empty, the update thread conducts the simplification step evoked in section 2.1.3. Once this simplification is achieved, we use the blocking facility of the lock system calls to suspend the update thread and save processor resources.

3.4. Flexibility

This priority queue mechanism appears very flexible. Though we are not exploiting this at the moment, it could handle several tools acting on the field at the same time without requiring any modification. These tools, possibly controlled by different users, would be inserted into the *Level Manager* as usual and transparently handled by the update-thread. The tools could even interact in overlapping regions as they would be properly inserted into the priority queue thanks to the exclusive insertion mechanism.

More pragmatically, priority tuning could also help improving the interaction quality. Suppose the user makes a large change at some large low-detail levels of the cell hierarchy, then comes to another region where small details already exists, and tries to edit these small details. With our current priority scheme, no update will happen until the global update sequence level reaches this level. We could easily avoid this problem with little impact on the rest of the modules, by tuning the priority evaluation according to the user's current focus.

4. RENDERING AND SIMPLIFICATION

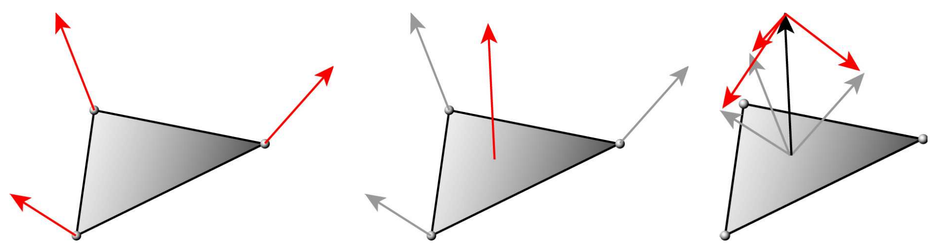
Excepting [15, 1, 2, 12] that use ray-casting to visualize the iso-surfaces of the scalar field, all other approaches are based on the Marching Cubes Algorithm.
Basically, the ray-casting process allows an update of only the portion of the screen that is being edited (tool projection footprint). This may appear as more efficient. However, it also forbids movement of the object while editing it because the redraw effort would then become too important.

Our first approach [7] did confirm that the Marching Cubes algorithm was well suited to interactive update and visualization of the surface, exploiting graphics hardware. In our resolution-adaptive sculpting system, we still use the costless sphere-mapped environment texturing that was introduced in [7] to improve shape perception by generating high quality highlights.

4.1. Surface creation

To display the iso-surface, we test each cell against *iso-crossing*. This consists of comparing the eight corners' field value to the iso-value. These comparisons serve to compute a Marching Cubes configuration index. If the cell *crosses* the iso-value, we associate a *Surface Element* to it. This structure stores the Marching Cubes configuration index (an integer) and at most twelve pointers to some *Surface Points*. The *Surface Points* are the intersections of the iso-surface with the current Cell's edges. They are linearly interpolated from two adjacent *Vertices* to match the iso-value. They also interpolate the vertices attributes, such as the gradient (that becomes the surface normal) and color information. Simple *time-stamping* mechanisms avoid multiple computations of these points during the vertices update step through the Cells exploration.

Additionally, the *Surface Element* is used to estimate the surface *discrepancy* introduced in Section 2.1.3. We need a quantity that indicates the *flatness* of the extracted surface. We decide to exploit the normals extracted at the surface points. If the normals are all pointing in a similar direction, the surface will be well represented by our linear approximation. On the contrary, if they have very different directions, our linear approximation is poor and the sampling rate should be increased to better match the underlying iso-surface. We use a straightforward estimator that computes a kind of standard deviation of the surface normals (see Figure 9).



(a) (b) (c)

FIG. 9. Estimating the *flatness* of a surface element: we first compute an average normal of the normals computed at the surfaces points (b) and then sum-up the squared length of the difference vectors between the surface point normals and the average (c).

As a result, we obtain many approximations (*Level Of Detail*) of the iso-surface at each level of the cells hierarchy. Figure 10 shows, for the same surface, different approximations that are computed and stored along the cell hierarchy.

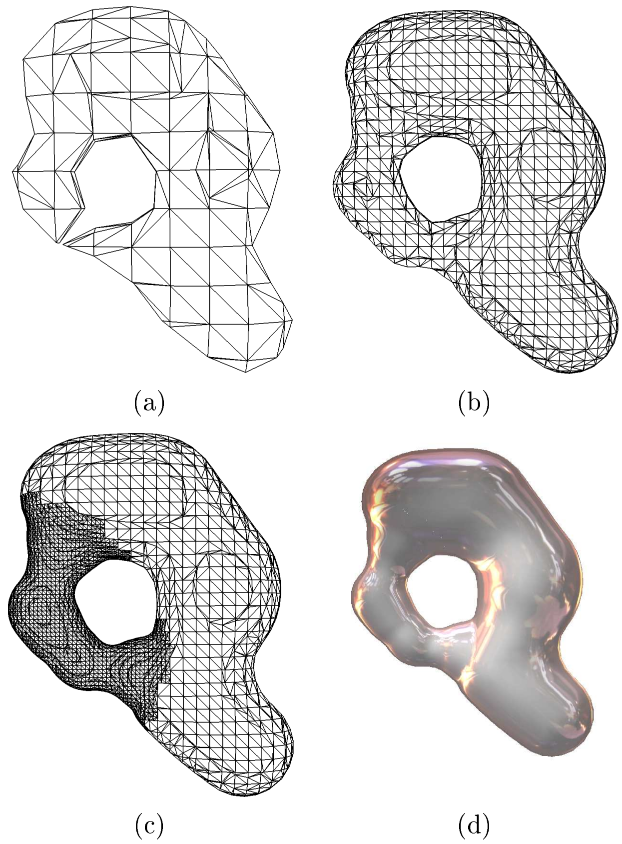


FIG. 10. Different levels of surface elements from coarse (a) to fine (c), with an intermediate level (b). Figure (c) illustrates the adaptive refinement of the sampling rate: the wireframe surface displayed reached the bottom of the hierarchy. (d) shows a textured version of the polygonal approximation in (c).

4.2. Surface display

The refinement process guided by the discrepancy estimator enables correct sampling of the field. However, at the leaf level of the hierarchy we obtain far too many triangles for any current graphic hardware to display interactively.

Our first solution to remedy this was to conduct a simple object based view frustum culling. As the cells hierarchy also constitutes a good space partition, we can efficiently prune the cells outside of the view frustum. This is particularly useful when editing small features that are part of a large model. For example, most of the surface of the sculpture displayed in Figure 12.(a) gets culled very early in the hierarchy, thus the hierarchy exploration for displaying the surface is concentrated on the visible parts.

However, this is clearly not sufficient when the whole model is contained inside the view frustum. To address this problem, we compute for each cell an estimated projected size on the screen. It is estimated from the cell's size and the distance of the cell's center to the screen projection plane. Using this *projected size*, we can stop the hierarchy exploration when the projection of the current cell becomes too small. For example, if the projected size of a cell is smaller than a pixel, the triangles contained inside its children will be smaller, so we avoid visiting them and rather draw the surface element of the current cell.

This mechanism gives control over the number of displayed cells at each frame and dynamically selects a LOD dependent on the distance to the projection plane. We first exploited this display complexity control with a global *minimum projected size* that the user could edit (see Figure 11). Nevertheless, this fixed control was not sufficient because even during idle moments, the displayed surface remains unrefined. Moreover, if the user zooms in or out, the surface can result in a model that may be poorly or over populated.

Our solution here is to automatically adjust this *minimum projected size* from frame to frame to maintain a given framerate during user interaction. At the end of each frame, we measure the time spent from the previous frame end and we use the difference with the desired *display time* to weight the *growing factor* of the *minimum projected size*. Pragmatically, we used the third power of this difference to minimize its influence when the display time is near its goal, and emphasize it when it's far from it, keeping its sign. When the user is idle, the limit projected size is progressively reduced close to zero. So the fully detailed geometry can be rendered if the user waits sufficiently long; which will allow a non-interactive but accurate display during the session.

Contrary to other multiresolution iso-surface constructions, we pay no attention to the cracks that appear between adjacent cells of different sizes. Solving this problem by drawing more triangles would not be a solution, since the problem may come from the fact that the field is not defined at the same resolution in adjacent cells. A solution would be to simply reconnect the surface-elements found at each cell. We deliberately choose to NOT to do this. A first reason for this choice is that the surface is always changing during the sculpting process. Another reason comes from the fact that we dynamically select, at each frame, where to stop in the cells hierarchy display. Reconnecting surface elements would force us to always track the neighboring cells. This would largely slow down the display rate, which is especially true in our unconstrained hierarchy (adjacent cells could be distant from more than one level of resolution). Moreover, as long as a sufficiently large number of polygons is displayed, the cracks can remain hardly visible (see Figure 10 (d) for example), it is thus *a posteriori* not worth the effort.

If needed, performing a global offline reconnection process would be easy if the current surface has to be converted to a standard mesh representation to output an export file.

5. APPLICATIONS

We show here two examples of sculptures produced with our system. The objects were sculpted using a 2D-mouse with a "virtual trackball" metaphor (a Magellan spacemouse is also available). The images are direct screen shots from the interactive sculpting sessions.

Additional material, such as higher resolution images and a video showing our system interactivity are available at
`http://w3imagis.imag.fr/Publications/2001/FCG01/`.

5.1. Creation from scratch

Our first example is a character model created from scratch. The wireframe views in Figure 11 illustrates the display complexity control. In this case, we use a fixed *minimum projected size*, so the closer the model, the more detailed surface version we get. If the *minimum projected size* was automatically updated, as no part of the surface is culled, we should obtain approximately the same complexity on the four views.

FIG. 11. The different views show the adaptation of the LOD while the model is moved closer to the camera.

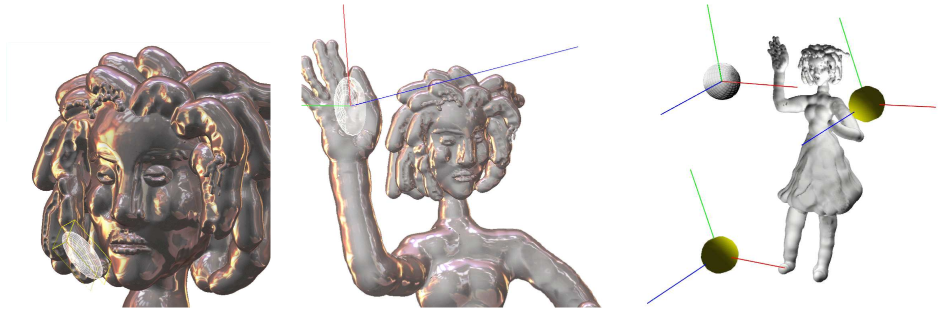
Figure 12 shows some steps of the modelling process. Face details have been modelled first using additive tools of different sizes to create the head, the chin and the hair (see Figure 12 (a)). Then, a negative tool has been used to remove material in the eye regions and the eyes have been created inside these hollows by successively adding and then removing material. High resolution editing was necessary for sculpting the lips. Next, a coarse body was progressively created (Figures 12 (b) and (c)). (b) shows details on the hand creation, which uses again successive addition and removal operations. The sculpture was created by a beginner with this multiresolution sculpting system in about two hours.

5.2. Editing of a model imported from a mesh

The ability to import and edit existing models is an interesting feature.

Volumetric dataset are easily converted to our *material density* 3d field representation: it consists of performing a translation and scale of the input values to make the iso-surface match the desired iso-value, and the field values range in the desired min-max interval.

Importing polygonal models can be cumbersome, especially if the input models are non-manifold/orientable or contain some intersecting polygons. We initialize the field on each grid point using the signed distance to the polygonal mesh. Field



(a) (b) (c)

FIG. 12. Three steps of a character's modelling. Figures (a) and (b) show two closed views used for modelling respectively the face and the hand. Figure (c) shows the tool and two lights (represented as spheres) around the character, and uses a different rendering style, with no high lights. Note the different surface resolutions in the three views, which are quite apparent in the eyes region.

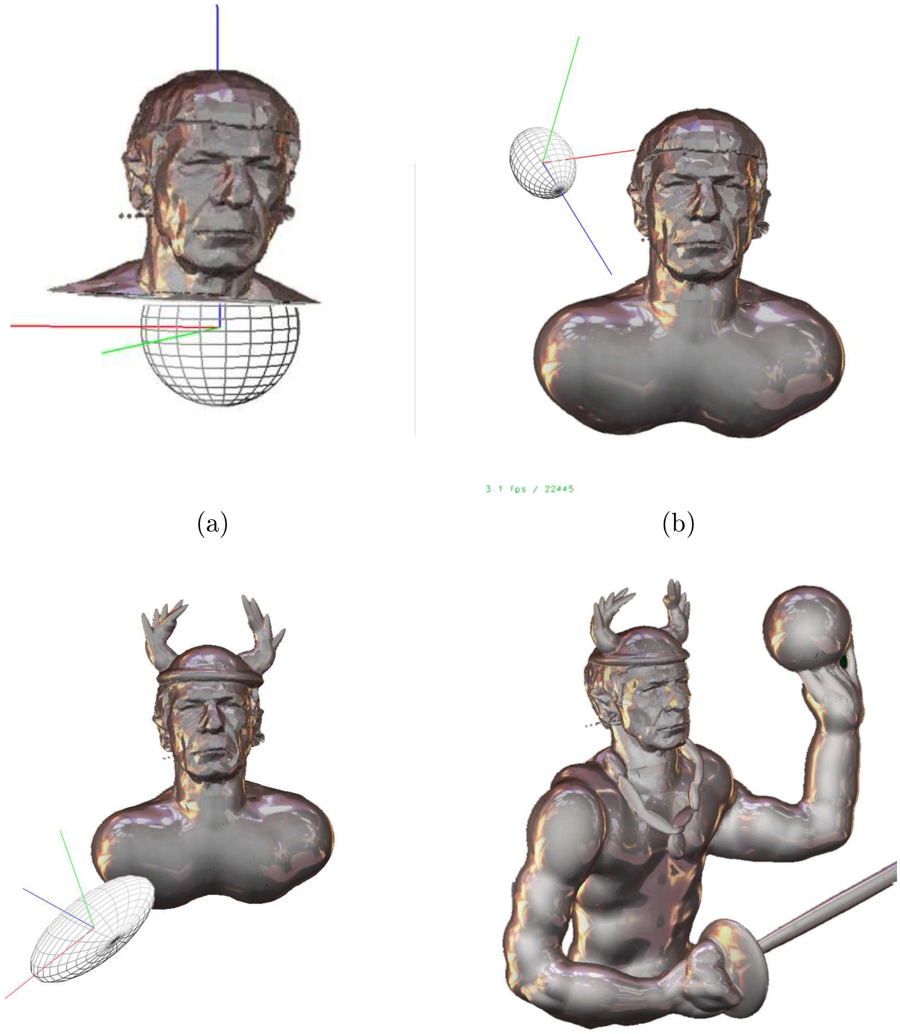
values are then translated and scaled to fit the desired iso-value and interval. A detailed description of this conversion method is beyond the scope of this paper, but note that the cell hierarchy can be of great help during the conversion process, since going deeper in the hierarchy can help disambiguate intricate situations.

Figure 13 shows an example where we edit a converted polygonal model. Large scale additive tools have been used for creating the main character's body features while preserving surface smoothness. A spherical tool inserted inside the head was used to create the helmet, and a slightly larger flat version of it for the helmet's border. The wings were created by successively using very small additive tools, thus creating high resolution details. The same technique was used for the chain. Creating the whole character took one and a half hour.

6. CONCLUSION AND FUTURE WORKS

We have presented a 3D sculpture system that provides direct interaction with the model – an iso-surface of a scalar field – at any modelling scale. The field representation, totally transparent to the user, is a fully dynamic hierarchical structure, that refines where and when needed. Interactive rates are obtained whatever the modeling scale due to a progressive update of the hierarchical field, from coarse to fine resolutions, using a priority queue mechanism. This mechanism enables continuous application of a tool while maintaining interactive update rates. Rendering relies on the hierarchical structure for performing local iso-surface extraction and displaying the adequate LOD depending on the size the region occupied on the screen. We have shown that our system can be used for both direct creation of a complex surface, or for editing a surface converted from another representation. Our examples also show that our system is especially well adapted to the design of organic shapes, such as the creation of virtual characters.

A first extension to the system would be to interface it with a force feedback device, such as existing articulated arms (i.e. *Phantom*). One should note that combining such an haptic system with our multiprocessing technique based on



(a) (b)

(c) (d)

FIG. 13. Editing a model imported from polygonal data.

pthreads is not a problem, since the phantom interface is running in a specific thread under IRIX OS. We have already experimented with it in the previous version of our sculpting system [7] as described in [3]. Haptic interaction proved to be a great aid in the sculpting process, since the user can "feel" the model, and thus decide more easily if he is adding material onto or in front of the surface. Extending force feedback to the multi-resolution version of our sculpting system would however require changing the way the feedback force is evaluated. Similar to the rendering pass, the cell hierarchy could be used to decide at which level force evaluation is performed according to the available time-interval (usually, force feedback is computed at 1000 Hz). This would result in coarser forces when large tools are used, but real-time response would be preserved.

However, haptic interaction through a force feedback device is still very far from the sense of touch a designer feels when he sculpts with real clay. We are currently planning to use a more direct "real-object interface". The user would manipulate a real deformable object, serving as an avatar for the totality, or for a part of the sculpture. His hand gestures would be captured by cameras and the reconstructed gestures would be used to deform the virtual sculpture. Our hierarchical representation may be very useful in such a framework, since the ability to maintain interactivity while combining several tools and actions is essential for simulating interactions with the ten fingers.

Other interesting extension would be to extend from multi-tools to multi-users, maybe over a network. The manager mechanism seems flexible enough to be extended to multi-managers without enormous effort. One straightforward solution would be to use a *master manager* to handle and order the tools requests for application, and then dispatch them to the *slave managers*. These *slaves* would locally maintain a discrete field's copy, thus limiting the network traffic to a simple *Tool-Copy*. This would enable increase efficiency in a collaborative sculpting task by increasing the number of artists; the lack of collaborative design facilities often being cited as one of the main limitation of digital models [16].

ACKNOWLEDGMENT

We would like to thank Pauline Jepp for carefully re-reading this paper. Thanks to the reviewers for their helpful comments.

REFERENCES

1. R.S. Avila and L.M. Sobierajski. A haptic interaction method for volume visualization. *Computer Graphics*, pages 197–204, October 1996. IEEE Visualization'96.
2. Andreas Bærentzen. Octree-based volume sculpting. *Presented at IEEE Visualization '98*, 1998. http://www.gk.dtu.dk/Andreas/publications.html.
3. Renaud Blanch. Virtual sculpture with haptic feedback (internal report, in french), September 2000. http://www-imagis.imag.fr/Publications/2000/Bla00/.
4. Jules Bloomenthal, Chandrajit Bajaj, Jim Blinn, Marie-Paule Cani-Gascuel, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill. *Introduction to Implicit Surfaces*. Morgan Kaufman, 1997.
5. Frédéric Cazals and Claude Puech. Bucket-like space partitioning data structures with applications to ray-tracing. In *13th ACM Symposium on Computational Geometry*, 1997. http://www-imagis.imag.fr/Publications/1997/CP97.
6. Eric Ferley, Marie-Paule Cani, and Jean-Dominique Gascuel. Virtual sculpture, September 1999. Short presentations proceedings, Eurographics '99.
7. Eric Ferley, Marie-Paule Cani, and Jean-Dominique Gascuel. Practical volumetric sculpting. *the Visual Computer*, 16(8):469–480, December 2000. A preliminary version of this paper appeared in Implicit Surfaces'99, Bordeaux, France, sept 1999.
8. Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. *Proceedings of SIGGRAPH 2000*, pages 249–254, July 2000. ISBN 1-58113-208-5.
9. 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).
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. Kevin T. McDonnell, Hong Qin, and Robert A. Wlodarczyk. Virtual clay: A real-time sculpting system with haptic toolkits. *2001 ACM Symposium on Interactive 3D Graphics*, pages 179–190, March 2001. ISBN 1-58113-292-1.

12. Ronald N. Perry and Sarah F. Frisken. Kizamu: A system for sculpting digital characters. *Proceedings of SIGGRAPH 2001*, pages 47–56, August 2001. ISBN 1-58113-292-1.
13. Alon Raviv and Gershon Elber. Three-dimensional freeform sculpting via zero sets of scalar trivariate functions. *Computer-Aided Design*, 32(8-9):513–526, August 2000. ISSN 0010-4485, an early version of this paper appeared in the Proceedings of the fifth symposium on Solid modeling and applications, 1999, pages 246–257.
14. Alexei Sourin. Functionally based virtual computer art. *2001 ACM Symposium on Interactive 3D Graphics*, pages 77–84, March 2001. ISBN 1-58113-292-1.
15. Sidney W. Wang and Arie E. Kaufman. Volume sculpting. *1995 Symposium on Interactive 3D Graphics*, pages 151–156, April 1995. ISBN 0-89791-736-7.
16. Stephen H. Westin. Computer-aided industrial design. *Computer Graphics*, 32(1), February 1998.