A Calligraphic Interface for Interactive Free-Form Modeling with large datasets

Bruno Rodrigues De Araújo Joaquim Armando Pires Jorge
Department of Information Systems and Computer Engineering
INESC-ID/IST/Technical University of Lisbon
R. Alves Redol, 1000-29 Lisbon, PORTUGAL
brar@immi.inesc-id.pt jorgej@acm.org

Abstract

We present a sketching interface for modeling shapes defined by large sets of points. Our system supports powerful modeling operations that are applied directly to the points defining the surface. These operations are based on sketch input to allow directly creating objects using simple strokes. Objects may be edited by either by boundary oversketching, or by cutting, relief drawing on surfaces, merging and cloning. By combining these operations we can create complex shapes, including objects with sharp features. Our work uses the Multi-level Partition of Unity Implicits (MPU) technique to convert point clouds into implicit surfaces. Furthermore, we have devised a fast adaptive incremental polygonization algorithm which takes advantage of the MPU structure. This makes local re-polygonization possible and allows real-time modifications to large point sets since it avoids re-calculating the whole polygonal representation from scratch after each modification.

1 Introduction

Illustrations and sketches are the starting point of any design process to create innovative manufactured products or to produce digital contents for 3D Computer Animation. However, sketch-based modeling interfaces are still unused or under-utilized by most CAD systems which continue to explore the WIMP (Windows,Icon,Mouse,Pointing) approach instead of taking advantage of new pen-based devices. Free-form sketching interfaces such as TEDDY [9] allow modeling simple objects using few operators over polygonal representations. We propose FreeFormSKETCH, a calligraphic modeling system which, through a reduced set of operators allows designers to create and interactively edit complex free-form shapes from large point datasets. Our goal is to provide precise and local modeling to complex objects without computational time penalty. We use the Multi-level Partition of Unity Implicits (MPU) [12] as our internal shape representation, providing modeling operators based on sketching metaphors.

The major contribution of this work, is the ability to handle large, complex forms through sketch-based operators. We present an innovative solution to interactive modeling using MPUs. By exploiting this technique, we are able to interactively modify and create sharp features on objects defined by tens of thousands of control points with local precision, as can be seen by videos we have made available[1]. This is achieved by means of a new algorithm which is able to re-polygonize only the affected parts of the object by generating meshes adapted to local shape features. The remainder of this paper is organized as follows. After surveying related work, we present an overview of our system. Then we describe how we adapted MPUs to allow interactive modeling. In Section 5, we present our operators in detail explaining how they manipulate the implicit representation. Then we describe our polygonizer, highlighting its support of sharp features, how it creates an adaptive mesh in a single step and how it makes local re-polygonization possible. Finally we draw conclusions.

2 Previous Work

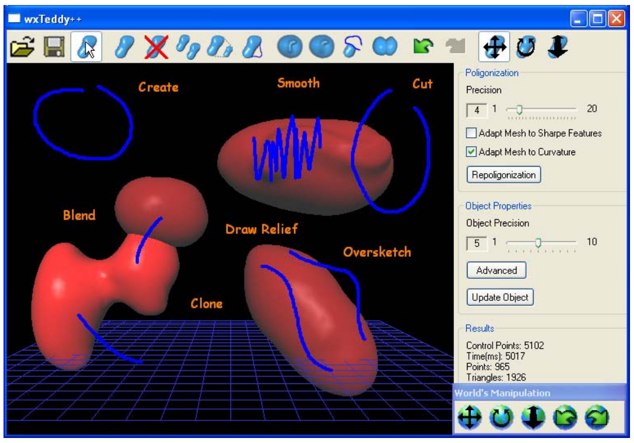
In the 90's, calligraphic interfaces for modeling appeared thanks to the first generation of pen computers. During this decade, the more relevant works proposing interactive modeling tools based on input stroke were the SKETCH [19] and Teddy [9] systems. SKETCH system used both sketch and gesture recognition, proposing a wide array of operators targeted to solid modeling, objects defined by planar faces, extrusion and CSG constructs. Its approach to sketching over a 3D view using a predefined gesture syntax was adopted by several recent works [15, 16]. Alternatively, Teddy proposed an interface for free form-modeling with a simpler interaction, providing operators better adapted to

[1]http://immi.inesc.pt/~brar/bear_xvid.zip,
http://immi.inesc.pt/~brar/venus_clown_xvid.zip

free-form such as bend, cut and smooth. However, Teddy presented several limitations due to using polygonal meshes which limited the variety of shapes supported. Moreover it did not allow creating complex objects.

Over more recent years, sketch-based modeling has become a growing area of Computer Graphics with the emergence of several systems [10, 6, 3, 17] which adopt implicit surfaces to overcome Teddy's limitations. Karpenko's work [10] was the first to adopt variational implicit surfaces(VIS) proposed by Turk [18] to support real smooth surfaces. However the proposed modeling operators did not scale well because the system uses meshing information to perform merging or oversketching of shapes. Our first attempt to solve this problem, BlobMaker[6] did not use meshing information. We augmented implicit functions with a 3D skeleton structure retrieved from 2D morphologic analysis of user input strokes. By doing so, the system could handle more complex shapes. The same approach was followed by [3, 17], using implicit models other than VIS. However, these representations are not powerful enough to handle complex shapes from well-known large datasets. Furthermore, none of them supports sharp features such as corners or edges, sharp cuts or boolean CSG, because they are based on C2-continuous implicit representations. This problem is partially solved in SmoothTeddy [8] by using a mix of polygonal and quadric implicit representations to combine advantages of both, i.e sharp features and smooth shapes. However, this is achieved through a separate mesh beautification step and it still is not adequate to support shapes of higher complexity. To this end we adopted the MPU implicit representation [12] in our FreeFormSKETCH system. While several systems [14, 4] are able to handle large datasets, however, they do not provide an attractive interface and they share the same problems as classical CAD systems.

Even though implicit surfaces provide many advantages over polygonal meshes, they require slow methods such as ray-tracing or polygonization to compute a visual representation. Since current hardware is still not adapted to offer real-time ray-tracing even using GPU programming, most practical approaches rely on meshing the implicit function. Marching Tetrahedra [5] the most popular method uses space subdivision following the original ideas of Marching Cubes [11]. However, as explained in our previous work [7] these generate poor quality meshing and require a post-remeshing step as devised by Ohtake [13] to improve mesh quality and be able to approximate sharp features correctly. Surface-tracking approaches seem to be more reliable approaches to achieve good quality meshes without an excessive amount of triangles or the need for a post-remeshing step. In this vein, FreeFormSKETCH improves on our previous surface-tracking algorithm [7] by taking advantage of MPU implicits, offering support for sharp features and, most importantly, local re-polygonization. Previously, Akkouche [2] presented a technique for adaptive meshing of implicit functions, based on Marching Triangles [1], a surface tracking algorithm. However, his method performs polygonization using two steps. An initial expansion step generates an incomplete mesh which gets improved by a second step which solves cracks and fills holes. Edge lengths of mesh triangles are adjusted using a midpoint projection heuristic and Delaunay triangulation. Our approach relies on curvature information directly extracted from the implicit function. Other improvements are presented in more detail in Section 6. Even if our polygonization algorithm rivals Marching Tetrahedral in running time, it is still too slow for interactive manipulation of large, complex shapes. However, to achieve interactive performance, our operators yield local modifications. To exploit this, we have developed local re-polygonization methods to avoid recomputing the polygonal mesh from scratch after local edits. This method is detailed in the next section.



Poligonization
Precision
4 1 20
Adapt Mesh to Sharpe Features [ ]
Adapt Mesh to Curvature [✓]
Repoligonization

Object Properties
Object Precision
5 1 10
Advanced
Update Object

Results
Control Points: 5102
Time(ms): 5017
Points: 965
Triangles: 1926

Figure 1. FreeFormSKETCH interface with strokes for gesture based interaction

3 FreeFormSKETCH Overview

The FreeFormSKETCH interface is organized into three areas as depicted in Figure 1. On the top, a toolbar showing all operators supported by the system which can be activated by clicking corresponding button. The main area provides a three-D perspective view of the scene allowing users both to model and to edit 3D shapes. Finally, on the right a control panel allows users to configure parameters of the currently active operator. The interface provides the following features:

Shape creation: Shapes are created simply by sketching their profile over an invisible plane. After, they are automatically inflated by the system which also generates the corresponding mesh. Since the input stroke defines the boundary of a shape, self intersecting strokes are rejected. If the sketch defines an open contour, it is automatically closed before creating the corresponding shape.

Oversketching and Merging: To change the silhouette it is sufficient to sketch a new boundary directly over the shape to be edited. The oversketching operator can be performed under any combination of camera view and shape orientation. As in BlobMaker, the merge operator is invoked by sketching a stroke connecting two shapes.

Cutting objets: The system allows users to cut an object by sketching a lasso over its 3D representation. The "cylindrical" volume defined this lasso and the view vector, is subtracted from the shape. This allows either to create holes in a shape or trim its boundary.

Local features: Other operators specify local changes on shapes such as creases and smoothing. To create a crease, users draw a path over the form to be edited. The resulting crease can be controlled in terms of its width, depth or height. It is also possible to control the transition smoothness in the area adjacent to the new crease. To overcome unwanted sharp features created by creases, we provide smoothing controlled by sketching over the shape. The area benath the convex hull of the stroke is smoothed. Users can control both the intensity of the smoothing and the width of affected areas.

Projecting 2D user inputs: Each time the user sketches over the 3D scene, the 2D input stroke is projected over the scene to compute the semantics of the operation. For example when oversketching a shape silhouette, we need to know which objects in the scene should be affected by the operation. We use a projection similar to BlobMaker using the 3D camera parameters. Differently from BlobMaker, FreeFormSKETCH supports changing the view at any time, via a floating toolbar.

Checking operator validity: Before applying any of the operators presented above we must check validity constraints as follows:

Shape Creation The input stroke defines the profile of the new shape. Self intersecting strokes are rejected.

Oversketching Objects Input strokes should start and end on the same shape. However, part of the stroke must lie outside the shape.

Merging Objects Both input stroke endpoints identify different shapes. The intersection of both shapes can not be empty

Cutting Objects The input stroke should represent lasso. At least part of the lasso must strike-through a shape.

For input strokes specifying *creases* or *smoothing*, there are no constraints which invalidate the operation, since these strokes directly identify the affected region. When any of these two operators are active, the shape is selected by the projection of the first point of the stroke on a surface.The area affected lies underneath the convex hull of the stroke, from the camera point of view.

Increasing usability: The editor provides two interaction modes to model objects. The first uses a toolbar to select the sought operator before use. The second uses the constrains attached to each operator. In this mode, each stroke, depending on its context (what object(s) underlie the stroke) directly invokes the appropriate operator without requiring any pre-selection. This is done by performing stroke recognition, combined with additional constraints to avoid ambiguity. Figure 1 depicts the different types of strokes. In both modes, world manipulation and object manipulation are performed in the same manner, i.e. by selecting the action on the toolbar before issuing the command.

4 Free-form Shape Representation

Using the MPU implicit model combines the advantages of point-based graphics with implicit representations. This allows editing shapes with free-form operators such as those implemented in our system. However, to support the operators introduced in the previous section, we had to extend MPU data structure. Notably, links between the implicit function and its corresponding mesh had to be added to support local re-polygonization to provide more responsive modelling tools.

4.1 Extending MPU for local re-polygonization

Polygonization is a time-consuming process because it requires pseudo-sampling of the implicit function. As previously stated, we follow a surface-tracking approach to generate a mesh, which is adapted to the curvature of the shape, in a single step. To support local re-polygonization, we create links between mesh points and the MPU data structure to be able to identify the meshing areas affected by modelling operators. Table **1** describes the content of the MPU cell data structure. Both *openPoints* and *closedPoints* are lists of references to mesh vertices which are under the influence of the local shape function of this cell. When performing polygonization whether global or local, the lists are updated. Using this information, for each control point of the implicit function which gets the corresponding MPU cells are invalidated. Through appropriate bookkeeping, it is easy to recover the corresponding mesh points which should be deleted. Thus we can quickly discard and recompute meshes corresponding to subareas of the surface affected by any operations on the shape.

4.2 Supporting control point changes on the MPU

Control points are the basis of the MPU data structure. These are organized into an octree during MPU creation. These points were used for defining each cell's local shape functions. When users edit a shape, we identify the region(s) affected by each operation. Then we apply the operator to the implicit function. Finally we update the corresponding mesh. From the MPU standpoint, there are two types of operators. Displacement operators update control points. Editing operators add to or delete control points from the structure. To support this, we extended the MPU data structure with three new functions. When performing changes to control points, we identify affected cells by checking the point against all the cells in the octree. This verification is made by applying the same distance test as used by the MPU to create local shape functions. If the cell is affected, then the local shape function must be rejected and the cell should be rechecked, thus resetting its internal checked flag. After this step the MPU octree remains incomplete. However, the data structure will be regenerated during local re-polygonization to generate new local shape functions which fit the new control points.

5 Modelling operators over MPU implicits

In a similar manner to BlobMaker or Teddy, users sketch a contour to create a new shape (Figure 2 left). However, in order to create the MPU implicit representation, we need to collect enough control points and their normal vectors. We chose to follow the same approach as BlobMaker by generating a VIS using the 2D contour to create boundary and normal constraints which is sufficient to handle the simple shapes created by users with a single stroke. Then the VIS representation is sampled, generating control points to fill the new MPU implicit function (Figure 2 right). This sampling step is performed using a simple octree to subdivide the 3D space and create a cloud of points with enough density to uniformly populate the MPU structures. Finally, we generate a mesh for visualization (Figure 2 middle). This process is computationally expensive enough to render simple shapes (such as those created in single interactions) in real time. FreeFormSKETCH offers also the ability to import Polyformat (PLY) or Stereolithography (STL) formats typical of large data sets. In this scenario all the information needed is present in the datasets. We reuse the importation facilities provided by the MPU authors [12].

Parameter	**Description**
root	Refers to the root cell of the octree
subcells	Refers to cell children
laf	Refers to the local shape function
checked	flag indicating if the local shape function should be generated
leaf	flag indicating if cell is a leaf
center	3D position of the cell center
size	size of the cell
openPoints	list of meshing unexpanded points
closedPoints	list of meshing closed ponts

Table 1. MPU Cell content

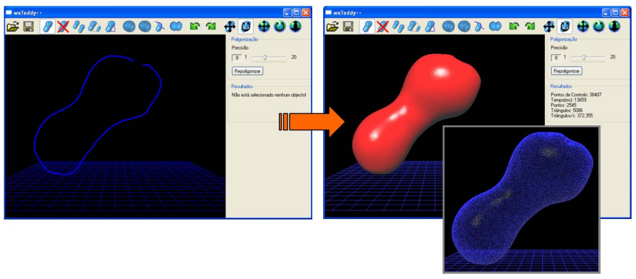


Figure 2. Shape creation example

5.1 Merging objects

The merge operator uses an approach similar to BlobMaker's. However, we extended it in order to allow users to control the smoothness of the transition between the two shapes involved. On the order hand, if smooth blending is disabled, merge behaves like a boolean CSG addition. This operator is implemented over the MPU thanks to the control point manipulation described on Subsection **4.2**. The new shape is represented by merging two MPU structures. However, due to the different orientations of both octrees, it is only possible to reuse one of the object data structures. This is composed by the updated MPU from the first object, whose cells need to be rechecked and the meshing information attached to it (which presents broken link due to removing the intersection of the objects). After this step, we

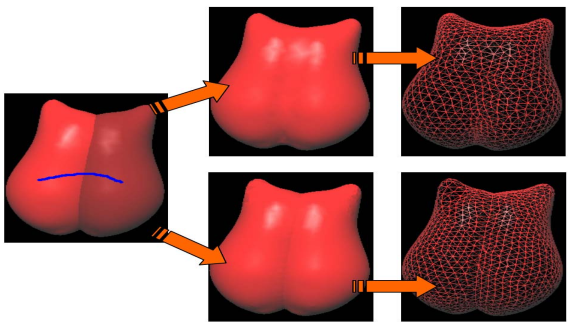


Figure 3. Merging results

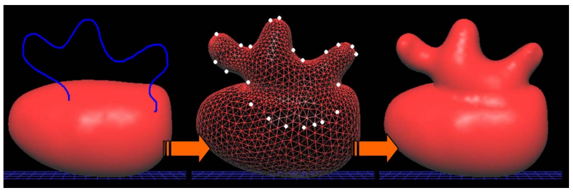


Figure 4. Oversketching example

complete the MPU information with new points and their normal vectors retrieved from the remaining part of the second object. We then regenerate the MPU's kd-tree. After, smooth blending is applied to control points located near the junction region if the smoothing parameter is not zero. Figure 3 depicts both scenarios, where the top row shows a merge with smooth blending. This is explained in more detail on the following subsection. As result of these operations, the new MPU corresponding to the merge of both objects is obtained. However, broken links still remain in the mesh. Moreover, the mesh corresponding to the second object is still missing. So, instead of re-polygonizing the whole shape, we perform a local re-polygonization step starting from the first object mesh.

5.2 Oversketching objects

The oversketching operator changes shapes by adding new control points to the MPU. This is similar to creating an appendix shape followed by a merge with the existing shape. To perform oversketching, we create a new blob define by the user's input stroke using the same approach as BlobMaker. However, since the stroke only defines the outside part of the new object as depicted in Figure 4, we need to extract more information to create a new contour which defines the appendix. The input 2D stroke is closed by using a half-ellipse projected on the original shape to join stroke endpoints. We perform tests and adjustments to guarantee that the projection of new points still lies inside the shape (Figure 4 middle).

5.3 Cutting

The cutting operator is performed by sketching a lasso that intersects the object as shown in Figure 5. First, we delete all control points from the MPU and meshing points located inside the cut part. Then we reconstruct the MPU and re-polygonize the missing parts of the shape. To remove such points, we project the stroke over the shape, by creating a set of 3D planes defined by two consecutive projected points and the 3D camera look-at vector. Then we reject any control point of the MPU inside the cut volume. We use the set of planes to delimit this volume. This process is similar to a 3D version of the Point-in-Polygon algorithm.

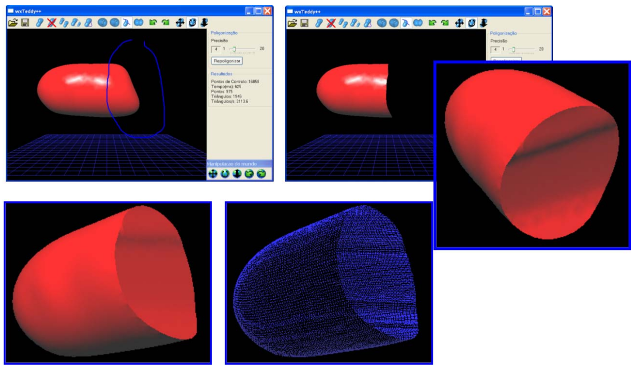


Figure 5. Steps of the cutting operator

After rejecting the unneeded control points, new ones are created using the projection of the stroke. These points are defined step by step by sampling the missing intersection area of the shape. In a manner similar to previous operators users can specify smoothing which is automatically applied in the transition region. Finally we perform a local re-polygonization step to complete the mesh.

5.4 Creating Creases on shapes

This operation changes control points in the MPU data structure which are located near to the region beneath the user-drawn stroke. First, we project the stroke on the 3D shape to get the affected region. By adjusting the parameters which define the crease (width, height and amplitude), users are able to create different effects such as bumps or valleys. The MPU control points located near the affected region, are updated according to their distance to the projected stroke. along the stroke width, they are changed depending of the height value along the normal to the shape at each point. The amplitude defines the smoothness of the transition using a quadric B-Spline based on the distance to

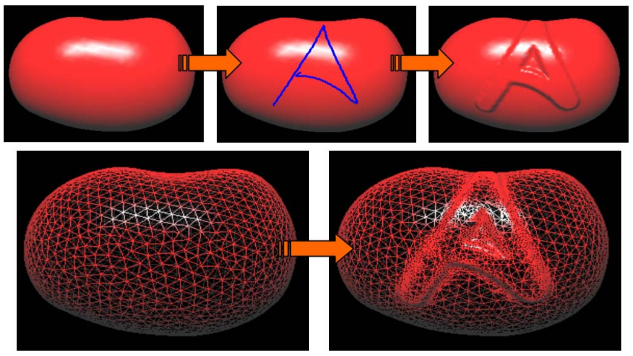


Figure 6. Relief Drawing Example: the rendered object and respective meshes

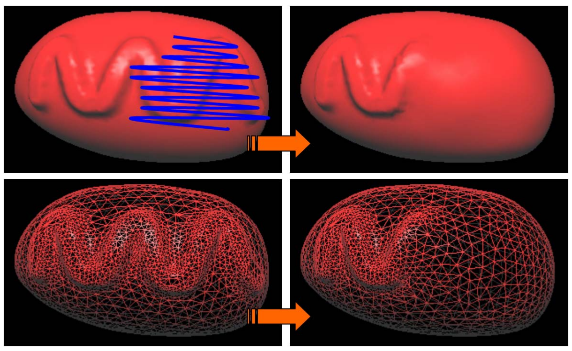


Figure 7. Smoothing operator example

the stroke. Finally the mesh is updated by re-polygonization as shown in Figure 6.

5.5 Smoothing Shapes

Similar to previous operators, the smoothing operator updates the position of control points and thus requires local re-polygonization. Smoothing is controlled by two parameters: width and intensity. The width defines the stroke's area of influence. The intensity is defined as the radius of the smoothing factor. For each control point located inside the area to be smoothed, we compute the centroid of neighbor points within the radius value and the average of their normal vectors, to define a 3D plane onto which control points are projected. If the control point is located outside the smoothing area its radius is gradually reduced to provide soft transitions such as depicted by Figure 7. Following this approach, we achieve good visual results without undue performance penalties while avoiding an extra processing step.

6 Polygonization of MPU implicits

Our polygonization is based on the technique developed for BlobMaker [7]. The algorithm generates an adaptive mesh using local characteristics such as curvature and taking advantage of the MPU representation in order to represent sharp features such as edges and corners. Following a surface tracking approach, it starts from a seed point of the surface, to construct an adaptive mesh in a single step.

6.1 Mesh expansion

Our algorithm uses direct point expansion instead of storing explicit representations of fronts as in BlobMaker. This allows us to avoid merging or splitting fronts. Unexpanded points remain stored in a simple bucket data structure. For each unexpanded point, we store a list with all its neighbor, while highlighting the two adjacent points on the expansion boundary. Another list contains the triangles defined by unexpanded points. We apply Blobmaker-type expansion using each point and its two adjacent points. During this step, we create new points are and insert them into each bucket while updating their connections. When the expansion of a point is finished, it is removed from the bucket. This process is repeated until there are no more unexpanded points in the list. During this step we update the lists presented in the subsection **4.1** to generate links between the meshing and corresponding MPU cells.

6.2 Adaptive Meshing

During expansion, we use heuristics based on the curvature of the implicit surface to compute edge length of output triangles as presented in [7]. To avoid unstable values of curvature, we choose an approximation by searching near the candidate point to be expanded. Even if this step is computationally more expensive, its provides better quality meshes. This approximation, based on binary search, is made comparing the expansion edge lenght of the unexpanded point *P* with its proposed expansion *P1*. If curvature changes between *P* and *P1* are greater than 80 percent of the value computed at *P*, the middle point between *P* and *P1* is used instead of *P1*. Using this solution, we achieve better guesses for edge length in regions with steep variations of curvature.

In order to avoid incorrect expansions, we introduced two new tests which proved essential to achieving better results. The first tries to avoid that the mesh expansion returns back on itself, and the other is the binary search step presented above to define the edge length for each newly generated triangle. The number of new points generated for each expanded vertex remains unchanged as compared to Blobmaker. However, this new approach requires some adjust-

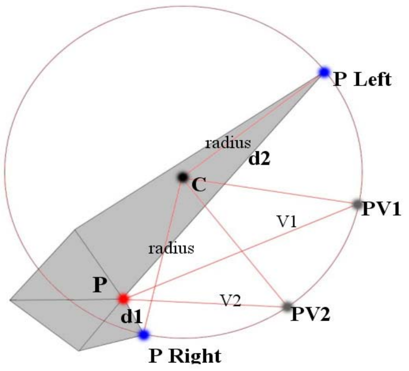


Figure 8. Edge Length readjustment during expansion.

ments as depicted in Figure 8 when the distances between the unexpanded point (red point) and each of its neighbors (blue points) are too different. In order to provide a mesh of better quality, we balance the distance between new points and the unexpanded point using the distance to booth adjacent points. First, we compute the center *C* of the circle with radius given by the average of distances *D1* and *D2*. Then we subdivide the angle $P_R\widehat{C}P_L$ following the same subdivision than $P_R\widehat{P}P_L$. The resulting new points *PVi* update the mesh with new triangles formed by joining them to the unexpanded point.

6.3 Collision detection

Since our algorithm follows a surface tracking approach, the expansion of a point must avoid overlapping the existing mesh. We apply collision detection is applied to solve this problem, to guarantee that the expansion is safe. This checks the unexpanded neighbors of the pivot in our previous system. However, we take advantage of the MPU octree data structure. Using the cell information presented in subsection **4.1**, we are able to access those points directly. This avoids having to apply the collision test to all unexpanded points. This improvement makes our algorithm faster than BlobMakers even when polygonizing more complex objects. When a collision is detected, we create a new edge between the expanded point and the nearest colliding point. Then we link the points adjacent to each point by adding four new triangles to the mesh. In this way we avoid inconsistent unexpanded points during the meshing process. This solution is both simpler and faster than the previous one which used front division and unification.

6.4 Approximating edges and corners

In order to support sharp features, we use the implicit function value and its gradient to approximate corners and edges as depicted in Figure 9. When MPU cells represent edges or corners, the local shape function f is the maximum of two convex quadrics f_1 and f_2. We compute the intersection of both quadrics by $f(X) = max(abs(f_1), abs(f_2))$. The gradient is then defined as the larger value of the gradient values. The global implicit value is defined by a blending of the local shape functions using weighting factors and searching the octree. When implicit function is at an edge or at a corner, the weight value applied to the function is unity, discarding all the information from neighbor points. This allows that during expansion, we can easily change the function behavior for our polygonizer when corners are detected and adapting the edge length correctly. If during expansion, we detect edges that are too long as compared to the curvature value, we check the approximation with the new function. However if edges are still too long, we apply a normal expansion step, thus discarding the sharp feature. This approach makes our algorithm slower in edge or corner regions. Moreover, computing the normal vectors at points lying on edges or corners is problematic. This is because the blending mechanism of the MPU does not provide precise values for these features. For this reason, we approximate normal vectors in boundary points by interpolating the normals of their neighbor.

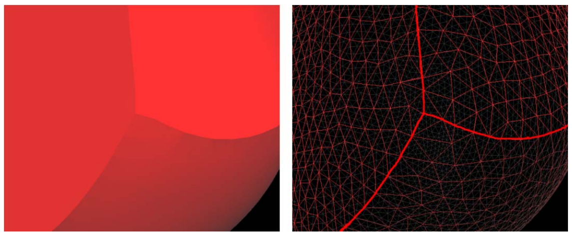


Figure 9. Example of corner approximation performed by our polygonizer

6.5 Local Re-polygonization

The method presented in Section **4** updates the MPU correctly by flagging cells which need to be rechecked during re-polygonization. However, to polygonize of missing parts, we need to identify the set of points that delimit the open mesh. This is done by creating a new bucket from the broken links remaining in the mesh. A recovery process checks triangles in the mesh which contain a broken link and discarding those which are not connected to a triangle. Otherwise if there are triangles, two cases may arise. First, if all the triangles are connected, the point is inserted into the bucket and marked as unexpanded. Otherwise, if any of the triangles is not connected, the recovery is not possible since more than two points are to be connected. In this case, the point is removed and its connected triangles are deleted from the mesh. After the recovery step, we apply mesh expansion to the unexpanded points such as before thus covering all the missing parts of the mesh as shown in Figure 10.

7 Conclusions

FreeFormSKETCH is a calligraphic interface for free-form modelling which allows editing data sets defined by tens of thousands of points. We have adapted sketch-based operators to the MPU implicit model providing a complete set of free-form operators in order to manipulate those functions. Indeed, our system makes it possible to create shapes by sketching their contours, to merge 3D objects, to oversketch the boundary (silhouette) of a shape and to perform

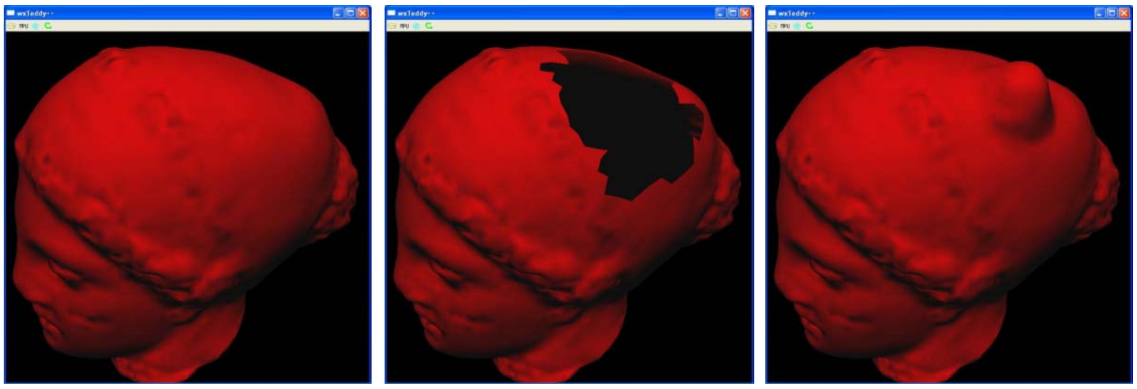


Figure 10. Local re-polygonization of the Igea head after relief drawing

CSG-subtraction cuts. Moreover, we offer local operators to create sharp features such as creases and valleys by sketching directly on the shape. To this end, we have adapted our surface tracking polygonizer in order to be able to render sharp features such as corners and edges. Notably, by using MPU information we are able to avoid complete re-polygonization of complex surfaces after users perform local changes. By doing so, we can interactively and quickly modify complex shapes retrieved from large datasets.

8 Acknowledgements

This work was supported in part by Portuguese Science Foundation (FCT) grant POSC/EIA/59938/2004 and by European Commission grant IST-2003-004785 (IMPROVE). The authors would like to thank Carolina Freitas Antão and José Alberto Almeida for implementing the prototype.

References

[1] J. I. A. Hilton, A. Stoddart and T. Windeatt. Marching triangles: range image fusion for complex object modeling. *International Conference on Image Processing*, 1996.

[2] S. Akkouche and E. Galin. Adaptive implicit surface polygonization using marching triangles. *Computer Graphics Forum*, 20(2):67–80, 2001.

[3] A. Alexe, V. Gaildrat, and L. Barthe. Interactive modelling from sketches using spherical implicit functions. In *Proceeding of the 3rd AFRIGRAPH '04*, pages 25–34. ACM Press, 2004.

[4] R. Allègre, A. Barbier, E. Galin, and S. Akkouche. A hybrid shape representation for free-form modelling. In *SMI*, pages 7–18, 2004.

[5] J. Bloomenthal. An implicit surface polygonizer. In P. Heckbert, editor, *Graphics Gems IV*, pages 324–349. Academic Press, Boston, 1994.

[6] B. de Araújo and J. Jorge. Blobmaker: Free-form modelling with variational implicit surfaces. In *12 Encontro Português de Computação Gráfica*, pages 335–342, 2003.

[7] B. de Araújo and J. Jorge. Curvature dependent polygonization of implicit surfaces. In *SIBGRAPI'04 Conference Proceedings*, pages 266–273. IEEE Computer Society, 2004.

[8] T. Igarashi and J. F. Hughes. Smooth meshes for sketch-based freeform modeling. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 139–142. ACM Press, 2003.

[9] T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: A sketching interface for 3d freeform design. *Proceedings of SIGGRAPH 99*, pages 409–416, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[10] O. Karpenko, J. F. Hughes, and R. Raskar. Free-form sketching with variational implicit surfaces. *Computer Graphics Forum*, 21(3):585–585, 2002.

[11] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. Proc. of SIGGRAPH'87.

[12] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[13] Y. Ohtake, A. Belyaev, and A. Pasko. Dynamic meshes for accurate polygonization of implicit surfaces with sharp features. In *SMI*, pages 74–81. IEEE, 2001.

[14] M. Pauly, R. Keiser, L. P. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003.

[15] J. P. Pereira, J. A. Jorge, V. A. Branco, and F. N. Ferreira. Calligraphic interfaces: Mixed metaphors for design. In *DSV-IS*, pages 154–170, 2003.

[16] A. Shesh and B. Chen. Smartpaper: An interactive and user friendly sketching system. *Comput. Graph. Forum*, 23(3):301–310, 2004.

[17] C.-L. Tai, H. Zhang, and J. C.-K. Fong. Prototype modeling from sketched silhouettes based on convolution surfaces. *Computer Graphics Forum*, 23(1):71–83, March 2004.

[18] G. Turk and J. F. O'Brien. Shape transformation using variational implicit functions. In *SIGGRAPH '99 Conference Proceedings*, pages 335–342. ACM Press, 1999.

[19] R. C. Zeleznik, K. P. Herndon, and J. F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. In H. Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 163–170. ACM SIGGRAPH, Addison Wesley, Aug. 1996.