FaceSketch: An Interface for Sketching and Coloring Cartoon Faces

Nordin Zakaria and Hans-Peter Seidel

MPI–I–2003–4–009 June 2003

Authors' Addresses

Nordin Zakaria
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken

Hans-Peter Seidel
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken

Abstract

We discuss FaceSketch, an interface for 2D facial human-like cartoon sketching. The basic paradigm in FaceSketch is to offer a 2D interaction style and feel while employing 3D techniques to facilitate various tasks involved in drawing and redrawing faces from different views. The system works by accepting freeform strokes denoting head, eyes, nose, and other facial features, constructing an internal 3D model that conforms to the input silhouettes, and redisplaying the result in simple sketchy style from any user-specified viewing direction. In a manner similar to conventional 2D drawing process, the displayed shape may be changed by oversketching silhouettes, and hatches and strokes may be added within its boundary. Implementation-wise, we demonstrate the feasibility of using simple point primitive as a fundamental 3D modeling primitive in a sketch-based system. We discuss relatively simple but robust and efficient point-based algorithms for shape inflation, modification and display in 3D view. We discuss the feasibility of our ideas using a number of example interactions and facial sketches.

Keywords

Sketch-based modeling, Non-phororealistic modeling and rendering

1 Introduction

There is something in cartoon faces that is amiss in synthetic 3D faces meant to look "real". It is the combination of simplicity (to comprehend and to produce), aesthetic, expressive extravagation, and sometimes humor that makes it a unique communication paradigm. While much easier to create and requiring much less resources than 3D modeling, cartoon drawing, be it faces or whole-body characters, for many applications including animation, requires separate drawings for different viewing angles. Skills and care must be exercised, or proportions may be lost and the result less than pleasing. Similarly, colors and strokes assigned to the face or character must be properly aligned across different drawings. At the same time, at the artist's own discretion, for aesthetic purpose and to emphasize a point, the appearance of a character may be tweaked for a particular drawing. Hence, drawings may contain view-specific features or distortions that correspond to no consistent 3D representation.

We are concerned in this paper with the systematic production of facial human-like cartoon. We concentrate on the face as that is a major problem in itself and has many potential applications (web comic chat, digital assistant, animated features to name a few). We choose sketching as the primary interface medium, as in SKETCH[21] and Teddy[11], as it naturally suits the nature of the problem. We envision the notion of "sketch coming to life". The artist put in a few strokes of a face, in her natural style, colors it, and 3D is done algorithmically. As she turns over her drawing, there may be parts done automatically which is not quite what she has in mind. All she has to do is to pick up an eraser, erase those parts and sketch in what she wants. The vision as also described in [20] is one of bridging the gap between 2D drawing programs, which have fixed views, and 3D modeling programs that allow arbitrary view.

Towards our vision, we implemented FaceSketch, a sketch-and-color face drawing program that provide an interface well-suited for the task of producing facial cartoons. FaceSketch differs from SKETCH and Teddy in that it is domain specific. A general purpose modeler tends towards a complicated interface as the complexity of model creatable increases. Even if all required from the user are gestures, in the end there would be too many gestures and implicit rules to be learnt by the user [21]. FaceSketch also differs from both systems in that its fundamental modeling primitives are not triangles and polygons, but simple point primitives. With points, implementation issues are simpler, and relatively simpler and robust algorithms for inflation and shape modication can be designed. This has implications that are obvious especially if we consider embarking on a suite of domain-specific shape-from-

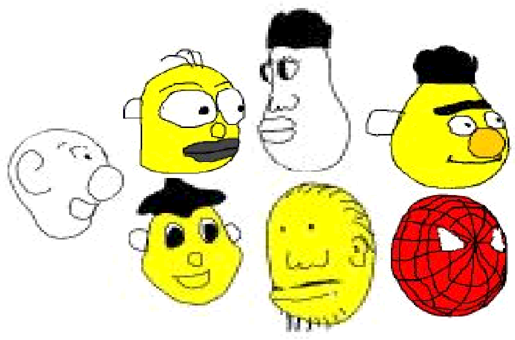
Figure 1: Cartoon faces created using FaceSketch

silhouette applications. Finally, of course, FaceSketch is tuned towards the 2D problem of cartoon creation rather than actual 3D modeling itself. Hence, our contribution in this paper is twofold: 1) we define an interface to tackle the problem of cartoon face creation and, by showing how simple interactions with it lend powerful drawing functionality, demonstrate the advantage of taking a domain-specific approach to sketch-based modeling, 2) we define a novel implementation of sketch-based modeling based on point primitive data structure.

Our current prototypal implementation of FaceSketch supports shape inflation for the head, eyes, nose, mouth, ear and hair. For the head, eyes, nose and mouth, the basic inflation algorithm, as in [11], is based upon a constrained Delaunay triangulation of the closed polygonal silhouettes sketched in by the user. The details of the algorithm differ, as will be detailed out in the sections to follow. Algorithms for the hair and ear are entirely different, as will also be detailed out. A user knows her drawing is '3D' only because dragging her mouse rotates it. No special window or widget is needed for the task. Our user can color her drawing as a child would with pictures in a coloring book. Arbitrary strokes and hatches can be added onto the displayed drawing from any viewing angle. If extruded shapes from a particular angle is unlikable, the user can change it with a simple oversketch. View-dependency, a concept peculiar with cartoon objects, is supported with a feature that enables the user to translate and/or rotate selected facial features and strokes along the surface of the head.

For the rest of this paper, we organize our presentation as follows: In section 2, we discuss work closely related to ours. In section 3, we present our user interface philosophy and design. In section 4, we discuss the algorithmic nuts and bolts of our system, following which in section 5, we discuss our

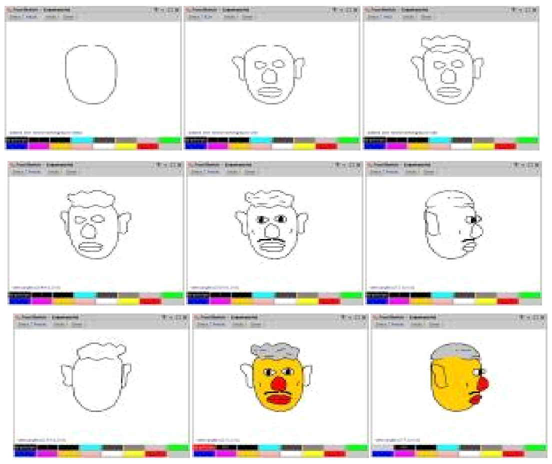
Figure 2: An example interaction with FaceSketch. From left to right, top to bottom: a head silhouette, more facial details, hair added, face rotated, more sketches added, side view, back view, colored face, rotated

actual implementation and initial user evaluation. Finally, in section 6, we plot future work and conclude our paper.

2 Related Work

The most accessible computing tools one could use nowadays for the task of cartoon drawing are general purpose paint utilities such as Microsoft paint (www.microsoft.com) or GNU gimp (www.gimp.org), coupled with perhaps ULead Gif Animator (www.ulead.com). An easier but much less flexible way would be to use tools such as Comic Creator (www.nfx.com) that provides prebuilt components for building up characters and faces. More expensive commercial software typically used in professional animation or cartoon magazine production studio include TicTacToon[5], SoftImage Toonz (www.softimage.com) and CreaToon
(www.creatoon.com). Sophisticated functionalities built into these tools generally center around features such as layered sprite processing, interpolation and warping. The authors know of no commercial tools based on user sketches.

In the Computer Graphics research community, the problem of cartoon production and animation has received attention since the early days. Levoy in 1977 [13] presented a 2D animation system, while Hackathorn in the same year [8] presented a system for 3D animation. In 1993, Siu Chi Hsu et al [10] presented the idea of skeletal stroke: a stroke formed by using an arbitrary picture as 'ink'. This picture can undergo deformation to give the impression of being rotated. But its lack of true 3D rotation limits its flexibility and use for general cartoon applications. Rademacher [17] approached the problem of cartoon creation from a different viewpoint. He described a technique for view-dependent model that rely on the definition of key deformations at specific viewpoints. At arbitrary viewpoint, the appearance of the object is interpolated from the closest key viewpoints. More recently, Frank and Van Reeth [6] discussed the problem of 2D cartoon as specifically one of getting the perspective right and the volume retained, and suggest a solution based on adapting the approach in [11] for cartoon animation. The solution dictates that 3D models be approximated from input 2D silhouettes, and the approximate 3D models be used to guide the display of cartoon drawing. We adopt a similar philosophy. We differ and innovate, however, in the details of our interface and algorithmic design.

Other works, not directly related to the cartoon problem, include physically realistic facial modeling. The goal and intention in this field, starting from the earliest work in [15] is different. Despite an array of techniques built throughout the years (see [14] for a survey), the subtleties of real face remain difficult to digitally reproduce with a convincing realism.

The work most influential on our system is Teddy [11], a freeform 3D modeling tool, for which the output is meant to be 3D, and user is to operate in a 3D setting. Operations supported in Teddy includes extrusion, 3D painting, cutting and smoothing. The underlying mesh representation may render certain operations unstable at times. Very recently, Karpenko [12] discusses an alternative implementation based on variational implicit surfaces. The disadvantage of implicit surface is its computational expense. Polygonization or ray tracing is needed for display.

Teddy builds upon the idea in SKETCH[21], an earlier work on sketch-based interface, that also has a major impact in the computer graphics community. SKETCH uses a system of intuitive guesses to create a geometric object guided by user's strokes and gestures. Once the user has mastered a set of implicit rules, it is easy to create complex models consisting of many primitives. Its primary problem is scalability. Real world usage requires more functionality, and more functionality would lead to to more complex gesture-based interface thus defeating its own purpose of simplifying user's life.

While both Teddy and SKETCH infer 3D geometry from 2D drawing, Cohen's Harold [4] and Tolba's projective stroke [20] takes a different strategy to make 2D drawings appear 3D. In both work, no 3D geometric model is employed. Tolba's projective stroke represent each stroke points in an

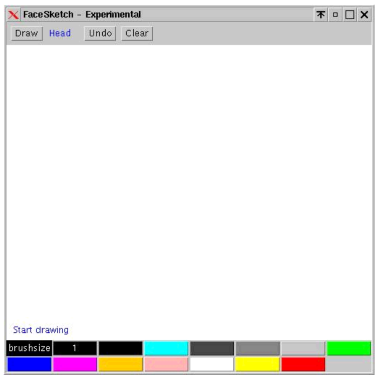

Figure 3: The visual front end for FaceSketch.

oriented projective set. Projective points are obtained by backprojecting drawn image points to lie on the surface of a unit sphere centered about the viewpoint, with the drawing window assumed to subtend some solid angle viewing port. In Harold, the primary primitive used is an image billboard, into which user's strokes are saved. Each billboard is oriented in a view-dependent way as the user moves through the world.

The primary novelty of our own approach to sketch-based modeling is in our use of point samples as the underlying data structure. In the modeling community, using point samples without additional topology is becoming increasingly popular, especially for the purpose of representing dense surfaces. Most research has been concerned with utilizing points to represent scanned models (see for example [18] and [16]), where the mesh tends to be highly densed. Points have also been shown to be practical for procedural model[19] and as a mean for efficient level-of-control[2]. We, on the other hand, use points to design simpler and more robust algorithms to inflate a 3D object from a 2D closed shape and to make subsequent modifications to it.

3 User Interface

As shown in Figure 3, the front-end of FaceSketch is dominated by a single orthographic canvas into which the frontal view of a cartoon face is drawn. As made evident in the figure, our primary design principle is simplicity. We seek a minimal visual interface, and focus more on stroke-based interaction. A stroke is defined when user pressed the left button on the mouse, dragged it and then released the button.

FaceSketch is based on a finite state machine model in which each state denotes a specific mode. Strokes are categorized according to the mode in

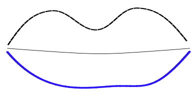

Figure 4: The three lines (each colored differently) making up the mouth.

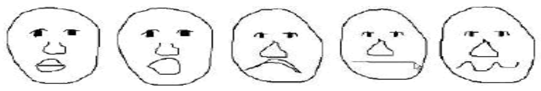

Figure 5: Mouth styles

which the interface is functioning. Modes supported in the current version of FaceSketch are namely 'Head', 'Eyes', 'Nose', 'Mouth', 'Ear', 'Hair', and 'Free'. By defining modes in which the interface can operate, we avoid the need for sophisticated stroke recognition algorithms, avoid recognition errors, raise user's confidence in the system, and generally avoid the pitfall encountered by pen-based technology[1].

3.1 Modes for Inflation

In the 'Head' mode, the first stroke is taken to define the silhouette of the head. All other strokes are interpretated to be sketches layered on the surface of the head. A similar treatment applies in the 'Eyes', 'Nose', 'Ear' and 'Hair' mode.

The 'Mouth' mode is a little more involved. We categorize the lines defining a mouth as illustrated in Figure 4. The first stroke defines the upper line, the second stroke the middle line, and the third stroke the lower line. Our current implementation does not support inflation of an open mouth. The mouth is thus currently assumed to be closed if it is to be inflated. Should an open mouth be required, user needs to select the 'Free' mode, and draw an open mouth as a free sketch on the surface of the head. Figure 5 shows various examples of mouth creatable using FaceSketch.

A constraint that applies to an inflatable stroke is that it cannot self-intersect. A similar constraint was reported in [11].

3.2 'Free' Mode and Colors

Strokes in the 'Free' mode can fall into one of five categories: 1) a silhouette oversketch, 2) a hair segment, 3) a surface sketch, 4) a feature translation, 5) a feature inversion.

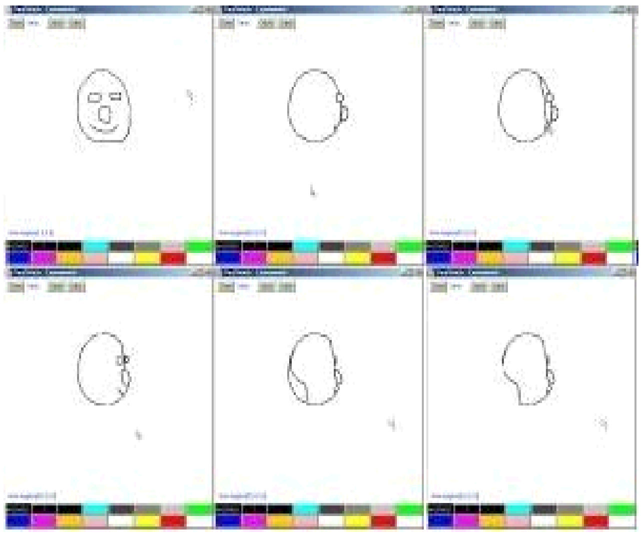

Figure 6: Modifying inflated shape. From left to right, top to bottom: front view, side view, silhouette oversketch, part of front deleted, another silhouette oversketch, part of back deleted

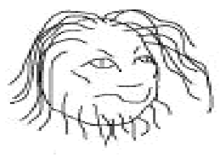

Figure 7: Hairytoon

A silhouette oversketch changes the overall shape of an inflated feature and is defined when at a viewpoint about perpendicular to the front view, the user enters a stroke the start and end of which falls onto the silhouette of the same object. Segments of a silhouette oversketch may fall within or outside the boundary of the projected face shape. An example interaction involving both silhouette oversketch and surface sketch is shown in Figure 6.

A hair segment is a line segment or curve protruding from the surface of a facial component, and is defined when user enters a stroke the start of which falls on the facial component, and the end of which falls outside the projection of any part of the face. A hair segment can be defined at any viewpoint, and repeated definition can lead to rather hairy faces, an example of which is in Figure 7.

A surface sketch is simply a line segment or curve formed on the surface

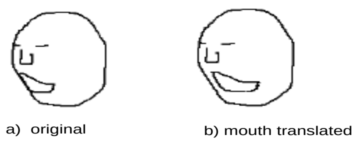

Figure 8: Face with translated mouth

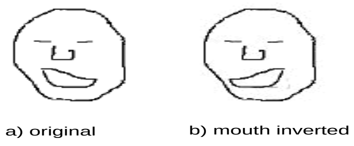

Figure 9: Face with inverted mouth

of an inflated feature, and is defined with a stroke the start and end of which falls on the projection of the face but not on any silhouette. A surface sketch can be used for arbitrary purpose. It can, for example, be used to create a drawing of a 'flat' pair of eyes on the surface of the head, should eyes be required that is not to be inflated but merely overlaid on the head.

A feature translation, an example of which is shown in Figure 8, is a functionality that permits translating of nose, eyes and mouth along the surface of the head. The stroke that defines a feature translation starts on a surface sketch or on the silhouette of a facial component other than the head itself, and ends anywhere else on the canvas. Another functionality that changes or tweak the appearance of the drawn face is what we call feature inversion. A feature inversion inverts a stroke horizontally and is defined when the user double-clicks the silhouette of the eyes, nose or mouth. Figure 9 illustrates.

Both feature translation and inversion are features that enable one to tweak a drawing for a particular viewing angle. Such view-dependent distortion is popularly done by cartoonists such as Lat (www.lathouse.com.my) and the late Uncle Fred (www.unclefred.com). As shown as an example in Figure 10, a common inclination among face and character cartoonists is to draw the face such that features such as the nose, eyes, or mouth always face the user when viewed from the front or from the sides.

For added attractiveness, drawing of silhouettes, hair segments and surface sketches can be done with colors. The surface of a facial component can also be filled with color. An object for which the user input silhouette has a designated color will be displayed with a silhouette of the same color from any other angle. User specifies a color by simply picking on any of the color icon below the drawing canvas.

Figure 10: An original character from Lat

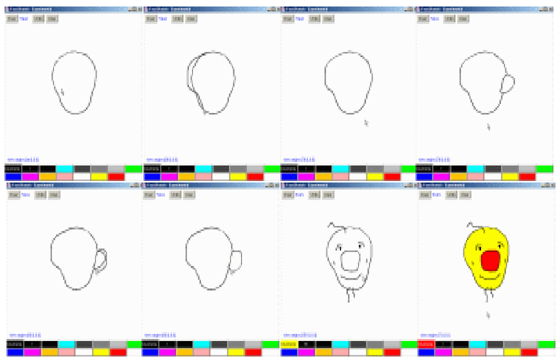

Figure 11: Sample snapshots from an example interactive session: head silhouette, head silhouette oversketching, modified head, head with nose, nose silhouette oversketching, modified nose, more face details, colored face

3.3 Example Interaction

We round up our discussion of FaceSketch's interface by describing in this section an example interactive session involving a hypothetical user, Sally. Figure 11 shows snapshots from the session. The full session is captured in the accompanying video file. Note that our presentation of the example includes a hint of how animation of user's sketching in FaceSketch could be done. While sketch-based animation in FaceSketch is beyond the scope of this paper and is still in an early experimental phase, we include it in our example to lend it a sense of completeness.

For the sake of narration,lets imagine Sally is a fan of the late Uncle Fred and she thought that it would be cool to use FaceSketch to draw Uncle Fred's Snuffy Smith character herself. So she started off the application and having noted that she was in the 'Head' mode, began by drawing the silhouette of a head on the canvas. As soon as she finished, the stroke was inflated to form the corresponding 3D shape. Sally would not see this unless she dragged the mouse a bit with the right button down, which she did. She was not satisfied with the result as it did not quite look as what she thinks Snuffy Smith's head should look like from the side, so she modified the shape from the side by oversketching the silhouette. She played around with the resulting head a bit, and was satisfied. She thought of going on to the 'Eye' mode, then decided otherwise as she felt that the character's eyes should not be inflated. She proceeded on to the 'Nose' mode, and drew a round nose from the front view. She rotated it sideways and again she felt that Snuffy Smith's nose had that unique shape seen from the side that could not be captured algorithmically. So she changed the shape by oversketching to that which she wanted, and she was happy it could be done rather easily. She thought about the mouth, ear and hair and felt that, like the eyes, those are best left to the 'Free' mode. So she proceeded to the 'Free' mode, and drew sketches on the head surface to denote the eyes, the mouth and the ear. She then drew curved strokes indicating Snuffy Smith's scanty hair and beard, while rotating the head around to get more accurate views. Then, as a final touch she colored her sketch. Now, she needed to do something with her cartoon. She would like to save frames from it to be later composed into an animated sequence for the web. So, she chose the initial desired viewing angle, rotated the head to that angle, then press the keyboard's 'S' key to save the corresponding image to a file. And she continued, and for selected frames, to make her animation more interesting, she modified her mouth stroke by oversketching it, to give it a different state of action at different time. Also, to make the moving mouth more visible from the side, Sally simply dragged the mouth a bit to where she wanted it to be. Finally, Sally was finished with her task - she had produced more than a dozen frames of Uncle Fred's Snuffy Smith in less than five minutes.

4 Algorithms

We describe in this section the algorithmic details of FaceSketch. The collection of algorithms that makes up FaceSketch include those for 1) processing user's input stroke, 2) inflating the input stroke, 3) sketching on the surface of an inflated facial component, 4) adding colors to the face, 5) changing shapes by oversketching silhouette, 6) translating and inverting a surface stroke or a facial feature, and finally 7) displaying the silhouettes forming a face.

The first algorithm is commonly required in a sketch-based application, and performed to output a smoother polyline from the input stroke. The way in which this is done in FaceSketch is similar to that in [11]. For each stroke, a sequence of screen-space points is collected. The stroke is filtered to remove all points whose screen space distance from the previous point is less than some threshold. A Catmull-Rom spline is then fitted to the remaining points, and then sampled every few pixels to generate a smooth-looking sequence of points.

All other algorithms are novel, and will be described in the subsections to follow. A brief preliminary on the structure of our point data and our application will make the subsections easier to follow.

Each point that we work with stores a 3D position, an RGB color, and a boolean flag, *selected*. The boolean flag is used in surface sketching and coloring. Note the absence of normal vectors. We implement no lighting functionality, and backface culling, as will be detailed out in section 4.10, is done with a trick cheaper than a normal vector-based technique [7]. Also noteworthy, the z dimension in the positional variable will be referred to interchangeably as depth or height. A point is said to be raised if its depth or height is nonzero.

When raising point samples, we seek to form a continuous surface. We require a continuous surface as this would make it easier to extract correct or better looking silhouettes. A discontinous surface or a surface with gaps may lead to a badly form silhouettes for certain viewing angles. The detail of our silhouette extraction and display routine is also given in section 4.10.

Another point to note is that our application is structured such that the xy extent of the screen space coordinate is the same as that of the world-view coordinate. This simplifies many tasks, especially in selecting facial feature or silhouette or a single point sample via picking.

4.1 Inflating an Input Stroke

We assume in this section a closed input stroke meant to be inflated into a internal 3D shape. After smoothing the input stroke, we perform a constrained Delaunay triangulation (CDT) [3] on the resulting polyline. As in [11], the resulting triangles are divided into 3 categories: triangle with two external edges (terminal triangle), triangle with one external edge (sleeve triangle), and triangle with no external edge (junction triangle). Figure 12 illustrates. The task now is to generate points within each of the triangles according to its classification. The generic idea is to sample 2D points within a triangle and then to raise each point such that the result is a triangle with an elliptical curvature. The triangulated surface is raised such that for each internal edge in each triangle, the midpoint is at a height which is half the length of the edge. There is actually no hard rule on the height value. We base it entirely on visual observation and preference.

4.1.1 Generic Concept

To inflate a triangle with points, we first identify the vertex in the triangle for which the height is to remain at zero. We denote this vertex as v_0. Generally,

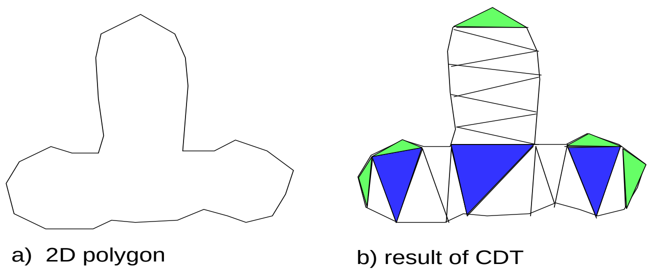

Figure 12: Example polygon and its CDT: unfilled triangles are sleeves, green triangles are terminals and blue are junctions

it is the vertex which lies on the boundary of the input stroke. For purpose of discussion in the subsections to follow, we will also refer to it as the *zeropoint*. We next identify the edge for which its two vertices have nonzero height. We denote this edge e_1. In general, this edge is an internal edge. Also, for the purpose of discussion in the subsections to follow, we will refer to it as the *heightedge*. Let the vertices at the endpoints of e_1 be v_1 and v_2. For each sampled point, p, lying within the boundary of the triangle, we compute the line, L, from the zeropoint to p. L is subtended infinitely and intersected with e_1. Let the point of intersection be x. We now need to find the height value at x. Let us assume that v_2 is at least as high as v_1. Let the height of v_1 be z_{v_1}. We compute the difference in height between v_1 and v_2, dz. The height, h_x, at x is then computed according to the following equation:

$$h_x = \sqrt{(1 - \frac{dist(x, v_2)^2}{dist(v_1, v_2)^2})dz^2}$$

where *dist(a,b)* is a function returning the distance between a and b.

Finally, we compute the height, h_p, at p according to another elliptical equation:

$$h_p = \sqrt{(1 - \frac{dist(p, x)^2}{dist(v_0, x)^2})(dz + z_{v_1})^2}$$

4.1.2 Inflating a Terminal Triangle

The basic idea in inflating a terminal triangle is to subdivide it such that the algorithm in section 4.1.1 can be applied to each resulting subtriangle. Let the vertices in a terminal be denoted v_0, v_1, and v_2. Let us assume that v_1 and v_2 forms an internal edge. Let the midpoint between v_1 and v_2 be m, and its height be half the distance between v_1 and v_2. Let L be a line segment

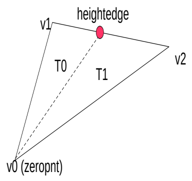

Figure 13: Subdividing a terminal triangle.

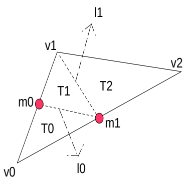

Figure 14: Subdividing a sleeve triangle.

from v_0 to m. L partitions the triangle into two pieces, as shown in Figure 13. We denote one piece T_0, and the other T_1. We now generate points for T_0 and T_1 according to the algorithm in 4.1.1. For T_0, the zeropoint is v_0, while heightedge is the line segment from v_1 to m. For T_1, the zeropoint is also v_0. The heightedge is, however, the line segment from m to v_2.

4.1.3 Inflating a Sleeve Triangle

Our approach to dealing with sleeve triangle is to first partition it as illustrated in Figure 14. We let the vertices in a sleeve triangle be denoted v_0, v_1, and v_2. Let us assume that v_1 and v_2 form an external edge. We let m_0 be the midpoint between v_0 and v_1, and m_1 be the midpoint between v_0 and v_2. Let l_0 be a line segment from m_0 to m_1, and l_1 be a line segment from m_1 to v_1. Together, l_0 and l_1 divides the sleeve triangle into 3 pieces, T_0, T_1 and T_2. We generate points for each of the subtriangles according to the algorithm in section 4.1.1. For T_0, the zeropoint is v_0, while heightedge is l_0. For T_1, the zeropoint is v_1, and the heightedge is l_0. For T_2, the zeropoint is v_2, and the heightedge is l_1.

4.1.4 Inflating a Junction Triangle

Inflation algorithm for a junction triangle is not as straightforward as that for a terminal or a sleeve triangle. We have tried without success to devise a scheme for generating points in a junction similar to that for sleeves and terminals. Our experimental effort generally results in point sets that are

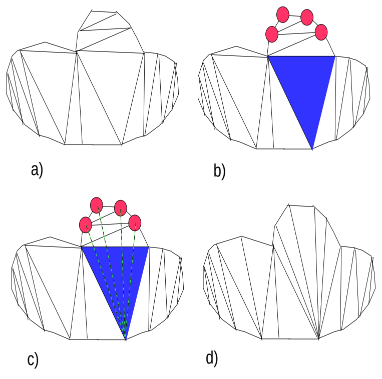

Figure 15: Merging triangles with a junction triangle

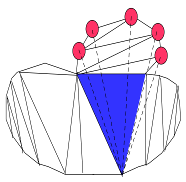

Figure 16: Merge followed by triangulation into sleeves not possible for highlighted junction

visibly rough, especially when each edge in the triangle has very different length and thus has midpoint raised to very dissimilar height. We therefore settled for a scheme whereby a junction triangle is merged with neighboring triangles lying to the side of its shortest edge, e_1, to form a single big polygon, which is then triangulated into sleeves. This is illustrated in Figure 15.

Our junction inflation algorithm as described works well for many shapes, but aesthetic problem does arise when the merged polygon has a point the distance of which from the vertex in the junction opposite e_1, v_0, is greater than the distance between v_0 and the midpoint of e_1 by a large enough amount. The merged region will appear too bloated from the side. More serious problem also do arise for certain shapes. Consider the case of head shape as shown in Figure 16. Following our scheme, it can be seen that merging results in a polygon that simply cannot be subdivided into sleeves.

Our solution is to merge a junction with its neighbors only where possible or appropriate. Let v_0, v_1 and v_2 be vertices of the junction, and let v_1 and v_2 form the shorter edge, e_1. We first measure the maximal distance between v_0 and points on the polygon formed by merging the junction with neighbors lying to the side of e_1. If the distance is greater than the distance between v_0 and the midpoint of e_1 by some threshold, the merging is considered to be unsuccessful. Also, if the merged polygons cannot be triangulated into sleeves, the merging is considered to have failed. In both cases, the merging is undone and the junction is treated as a sleeve and neighboring triangles involved in the merging each treated according to its respective category. Furthermore, to prevent gap from forming, the point set generated from the neighboring triangles are extended into the point set generated from the junction.

4.2 Inflating the Head, Eyes, Nose and Mouth

To inflate the head stroke, we first take its constrained Delaunay triangulation, and process each of the resultant triangles in a way as has been described in the previous subsection. This results in a point set for the frontal part of the head. To form the back part, the point set is simply mirrored about the $z=0$ plane.

A similar process applies for the eyes, nose and mouth, with the difference being that 1) no back part is extruded, and 2) each point of these features needs to be pushed to the front of the head.

To push a point, p, in a feature, f, to the front of the head, we need to compute the depth of a point, x, in the head having an x-y coordinate coinciding with that of p. A brute force algorithm would require iterating through each point p in f, and for each p, iterate through the points in the head until the desired point is found. We initially tried kd-tree to organize the search, but modification of point set generally requires rebuilding of the kd tree and we find that since this can be done rather frequently, at the user's discretion, it is too expensive for our interactive application. We opt instead to partition the head frontal point set into a 2D uniform grid structure along the xy plane. Search can then be done by iterating through head points in a single grid cell.

4.3 Inflating the Ear

The ear is treated differently from any other facial feature. We assume the ear to lie within a cylinder. For each point on the silhouette of the ear, we cast a ray into the screen and compute intersection with the cylinder surface. The silhouette point is projected onto that intersection. This is as illustrated in Figure 17. This simple scheme, an approximation of Tolba's projective stroke [20], seems to work well for cartoon ears. As a rule of thumb, radius of cylinder is taken to be a third the diameter of the head.

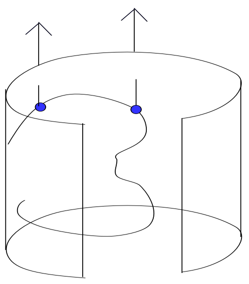

Figure 17: Drawing ear in 3D.

4.4 Inflating the Hair

To inflate a hair silhouette, we first iterate through each point on the silhouette and check if it lies within the polygonal silhouette of the head. If it does, we raise it according to the depth of the head at that point. Next, for each sample point, p, we cast a upward ray, s, and a downward ray, r. There are 2 possible cases for r: 1) r hits a border line for which either vertices have nonzero height., 2) r hits a border line for which both vertices have zero height. We let the point of intersection of r with the border line be x and that of s with another border line be y. If case 1) above is true, we linearly interpolate the depth, d, at x using the depth at both vertices of the line. Next we compute the depth at p, h_p using the following formula:

$$h_p = \sqrt{(1 - \frac{dist(p,x)^2}{dist(y,x)^2})d^2}$$

In the second case, we assume that the midpoint, m, of the line connecting x and y has the highest depth for any point on the line. We let the depth, d_m, of the midpoint be half the distance between x and y. If the distance between p and x is less than the distance between y and p, we use the following formula,

$$h_p = \sqrt{(1 - \frac{dist(p,m)^2}{dist(m,x)^2})d_m^2}$$

else we use the following formula:

$$h_p = \sqrt{(1 - \frac{dist(p,m)^2}{dist(m,y)^2})d_m^2}$$

4.5 Filling Up Holes in Point Sample Set

Filling up holes in point sample set proceeds by looking for and filling gaps noticeable from projecting points in each axis aligned direction into a binary buffer. The buffer is first initialized to 0s. In projecting points to the buffer, a pixel in the buffer is set to 1 if it receives a projected point.

We then iterate through the pixels lying within the bounding rectangle of the projected points. For each pixel in the buffer, we compute its screen space coordinate and check to see if it lies within the polygonal silhouette of the projected object. If it does and its pixel value is 0 (it is unset), a gap is assumed to have been found, and a new point sample is linearly interpolated from neighboring points.

4.6 Surface Sketches and Hair Strains

We define a surface sketch as a set of 3D points layered on a surface. Each point is obtained by projecting a screen space point onto a point in the face model point set with a matching screen space coordinate. Model modification requires recomputation of surface sketch. Recomputation relies on a data structure that stores a list of 2D screen-space hatch strokes and the viewing parameters.

We define a hair strain as a sequence of 3D points, for which only the first of which rest on a surface. The position of all other points depend on the viewing angle at which the hair stroke is sketched. The eye-space coordinate z value of the first point is propagated to all other points. The x and y eye-space coordinates are made to be equivalent to the screen space coordinate. The object-space coordinate value of each point on the hair stroke is then obtained by multiplying the eye-space coordinate of each point by the inverse of the modelview matrix.

4.7 Filling with Coloring

Our color fill algorithm first determine the selected facial feature whose projected points are to be colored. This is also done by determining the projected point in the face point set with screen space coordinate that matches screen-space coordinate of the user's mouse click. Each unculled point in the point set of the selected feature is then set with a user selected color, and has its *selected* variable set to *true.*

4.8 Translating and Inverting

Both surface sketch and 3d inflated features (namely the eyes, nose and mouth) can be shifted along the surface of the head. Translating a surface sketch involves recomputing projection on the head surface using the data structure described in 4.6. Translating a 3D inflated feature simply involves computing new depth value for each point on the feature (see section 4.2).

Inverting simply involves mirroring a selected inflated point set or a surface stroke about the center of its horizontal axis, and has a similar requirement for recomputation as do feature translation.

4.9 Silhouette Oversketching

Silhouette oversketching changes the point set associated with a selected facial component. Points may be deleted and new points added. The algorithm first partitions the oversketching stroke into a sequence of internal and external segments. An internal segment lies within the projected shape of the selected component, while an external segment lies outside it. Partitioning can be done either by projecting the shape into a binary grid and testing each point in the oversketch to see if it lies within a marked (internal) pixel, or by forming a polygon from the silhouette and applying a point-in-polygon algorithm [9]. For efficiency and robustness, we chose the first approach.

For each internal segment, we delete points lying within the polygon formed from it and the silhouette segment it bounds. For this, we compute a sequence of points on the silhouette whose start coincides with the end of internal segment and whose end coincides with start of internal segment. We then delete points lying within the polygon formed from the sequence of points. This deletion operation alone leaves a gap in the surface of the face, which may affect the appearance of silhouettes when viewed from certain directions. We proceed then to fill up the gap with a surface. The basic idea involved, as shown in Figure 18, is to shift the segment incrementally and build up the surface with points from the shifted segment. We do this by first projecting deleted points in a viewing direction, v, perpendicular to the viewing direction of the oversketching, to a binary grid, g. The minimum z value, z_{min}, and maximum z value, z_{max}, in world space of deleted points are computed. We next shift the oversketch from z_{min} to z_{max} incrementally. At each step, each point from the shifted oversketch is added to a list of to-be-added points if it falls onto a marked pixel in g when projected in the direction v.

For each external segment, we proceed by first finding the convex hull of the segment. All points that is within the convex hull are deleted. Should the

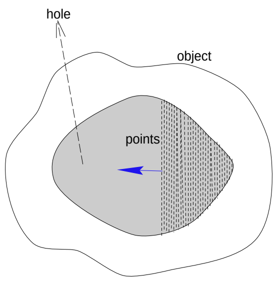

Figure 18: Filling up gap due to points deletion

external segment be defined on a flat segment of a silhouette, we stretch the hull a little into the selected shape before computing points to be deleted. Then, again, as for internal segment, the *zmin* and *zmax* of the deleted points are computed. We next compute the screen space distance between start of segment and end of the segment, assuming that it lies on the z=0 plane. We then shift the segment from z_{min} to z_{max} incrementally. For each shifted segment, we find the distance between the topmost point and the bottommost point in a subset of the deleted points having the z values in the range $z - e \geq z \leq z + e$ where e is a small value, and scale the segment appropriately. Resulting sequence of points are then added to a list of to-be-added points.

4.10 Silhouette Rendering

Current literature in silhouette extraction are focused mainly on extraction from polygonal meshes. That which we implement in FaceSketch is an image-precision algorithm suited for our point data set. The algorithm works only for external silhouette. Extracting internal silhouette would require normal vector information, and is beyond the scope of this paper. But we find that just the external silhouette alone is sufficient for a cartoonish display of the facial structure.

Our silhouette extraction algorithm proceeds as follows: For each point, p_i, we multiply it by the current modelview matrix to obtain q_i. As a quick (not entirely accurate but visually unnoticable) application-specific approximation to backface removal, noting that our head model is always centered about the plane *z=0*, we discard a point q_i if it has negative z value. Let the new set of point samples be denoted by Q.

The problem is now to find the shape of 2D point set formed by an orthogonal projection of each point in Q. We project each point in Q onto an screen-space precision binary grid, initialized to zeros. We now define, the notion of *extremum* in the context of the resulting set of 2d points. We define an extremum to be the leftmost set pixel (a pixel with value set to 1) or the rightmost set pixel of a horizontal scanline, or the bottommost set pixel, or the topmost set pixel of a vertical scanline. The procedure for finding extremums follows directly from this definition. We simply iterate through each horizontal and vertical scanline and collect extremums in each. The set of extremums form the shape of our projected point set. Note that this definition of extremum can be extended to shapes in which there exist scanline horizontal or vertical with more than two extremums. To find the extreme points in such a shape, for a particular vertical scanline for example, we not only need to collect the topmost and bottommost set pixels, but we also need to look for gaps in the scanline and collect points in pixels lying before the beginning and after the end of each gap.

The actual rendering algorithm proceeds, under the assumption of z-buffering being available and enabled. We first render all points, in background color if *select* = *false*, or in user-defined color if *select* = *true*. We then render each silhouette points with its user-defined color. Finally, we render all surface strokes and hair segment, each with its defined color.

5 Implementation and Evaluation

Our implementation of FaceSketch is in the Java language, and incorporates Jausoft OpenGL (www.jausoft.com) for the 3D graphics functionality. Our implementation, though still open to restructuring and finetuning, runs with a performance comparable to Teddy. Key to this real-time capability is an optimal use of OpenGL's display list, and graphics hardware where available.

FaceSketch is our first step towards a vision of seamlessly integrated 2D and 3D drawing environment. In its current iteration, it has major shortcomings. User comments confirm this. Though conducted in a rather informal and very limited manner, our user testing indicates that more would be better in our system. The few persons who had seen and laid their hands on our system were non-computer expert/nerd cartoon enthuasists (we intentionally chose test users in this category). They were excited upon seeing what can be done with FaceSketch. But they were quite disappointed at first when they learnt that though the software had been designed to provide an interface as natural (2D) as possible, the state of knowledge has not progressed to the point where drawing strokes of arbitrary style and manner can be processed and recognized with interactive techniques. They were not happy with the need to indicate current drawing mode, and they would like to be able to draw with freer styles and strokes (rather than with just simple curved strokes). It took some convincing before they would continue with the software beyond the first 10 minutes. The same video, as had been submitted along with this paper, was presented to them. Slowly, disappointment turned into fun, and with guidance and supervision, they began to produce some nice outputs within less than half an hour. The cartoons in Figure 1 was created mostly by users.

6 Conclusions and Further Work

We have discussed in detail the design and philosophy of FaceSketch, and we have discussed the mechanism of user interaction with FaceSketch and the user interface issues involved. We have shown that implementation-wise, the hallmark of FaceSketch is simplicity. No 3D mesh processing needs to be done, and no 3D mesh topology needs to be maintained. Also, no expensive implicit function needs to be solved or raytraced. We have shown how these are made possible by deploying simple point primitive as the fundamental building block of 3D entities modeled within the system.

FaceSketch, as it is, is still an open research. We are currently branching or seeking to branch our research into a number of different but related areas. We are studying ways of optimizing specific properties of point set in order to obtain better quality silhouette and nonphotorealistic display. We are also looking at smarter sketch processing and recognition, and attempting to apply results from computer vision and artificial intelligence. Also, we are looking into a deeper study of facial cartoon animation, building upon the interaction design as described in section 3.3. Finally, we would like a more systematic study of the human factors involved in using FaceSketch. Specifically, we wish to actually study how people sketch, and see how the interaction workflow and presentation in FaceSketch maps onto people's actual behavior and characteristics when sketching.

References

[1] J. P. Allen. Who shapes the future?: problem framings and the development of handheld computers. In *Proceedings of the ethics and social impact component on Shaping policy in the information age*, pages 3–8. ACM Press, 1998.

[2] B. Chen and M. X. Nguyen. Pop: a hybrid point and polygon rendering system for large data. In *Proceedings of the conference on Visualization 2001*, pages 45–52. IEEE Press, 2001.

[3] L. P. Chew. Constrained delaunay triangulations. In *Proceedings of the third annual symposium on Computational geometry*, pages 215–222. ACM Press, 1987.

[4] J. Cohen, J. Hughes, and R. Zeleznik. Harold: A world made of drawings. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering*, 2000.

[5] J.-D. Fekete, rick Bizouarn, ric Cournarie, T. Galas, and F. Taillefer. Tictactoon: a paperless system for professional 2d animation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 79–90. ACM Press, 1995.

[6] F. D. Fiore and F. V. Reeth. A high-level sketching tool for pencil-and-paper animation. In *Sketch Understanding: American Association for artificial Intelligence (AAAI 2002) Spring Symposium*, pages 32–36, 2002.

[7] J. Grossman and W. Dally. Point sample rendering. In *Rendering Techniques '98*, pages 181–192, 1998.

[8] R. Hackathorn. Anima ii: a 3-d color animation system. In *Proceedings of SIGGRAPH 77*, pages 54–64. ACM Press, 1977.

[9] E. Haines. Point in polygon strategies. *Graphics Gems IV*, pages 24–46, 1994.

[10] S. C. Hsu, I. H. H. Lee, and N. E. Wiseman. Skeletal strokes. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 197–206. ACM Press, 1993.

[11] T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: a sketching interface for 3d freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 409–416. ACM Press/Addison-Wesley Publishing Co., 1999.

[12] O. Karpenko, J. F. Hughes, and R. Raskar. Free-form sketching with variational implicit surfaces. In *Eurographics 2002*, 2002.

[13] M. Levoy. A color animation system: based on the multiplane technique. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 65–71. ACM Press, 1977.

[14] J. Noh and U. Neumann. A survey of facial modeling and animation techniques. *USC Technical Report*, pages 99–705, 1998.

[15] F. I. Parke. Computer generated animation of faces. In *Proceedings of the ACM annual conference*, pages 451–457, 1972.

[16] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342. ACM Press/Addison-Wesley Publishing Co., 2000.

[17] P. Rademacher. View-dependent geometry. In A. Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 439–446, Los Angeles, 1999. Addison Wesley Longman.

[18] S. Rusinkiewicz and M. Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[19] M. Stamminger, , and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Eurographics Rendering Techniques 2001*, pages 151–162, 2001.

[20] O. Tolba, J. Dorsey, and L. McMillan. Sketching with projective 2d strokes. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 149–157. ACM Press, 1999.

[21] R. C. Zeleznik, K. P. Herndon, and J. F. Hughes. Sketch: an interface for sketching 3d scenes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 163–170. ACM Press, 1996.

MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	?
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nrnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nrnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition
MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-003	Y. Kazakov, H. Nivelle	Subsumption of concepts in $DL\ \mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-012	D. Fotakis, R. Pagh, P. Sanders, P. Spirakis	Space Efficient Hash Tables with Worst Case Constant Access Time
MPI-I-2003-1-011	P. Krysta, A. Czumaj, B. Voecking	Selfish Traffic Allocation for Server Farms
MPI-I-2003-1-010	H. Tamaki	A linear time heuristic for the branch-decomposition of planar graphs
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Soon to be published
MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	M. Dietzfelbinger, H. Tamaki	On the probability of rendezvous in graphs