DETC2004/57705

TEXTUAL ANNOTATION IN A HEAD TRACKED, STEREOSCOPIC VIRTUAL DESIGN ENVIRONMENT

Eric Karasuda
Computer Science Department
University of California, Berkeley
Email: ekarasud@kingkong.me.berkeley.edu

Sara McMains[*]
Mechanical Engineering Department
University of California, Berkeley
Email: mcmains@me.berkeley.edu

ABSTRACT

In this paper we describe the textual annotation system we have implemented in CHaMUE, our Collaborative Haptic Mark-Up Environment software. This head tracked, stereoscopic virtual environment for design review allows remote participants to critique and refine product design ideas in real time, both by drawing on 3D virtual representations of the objects and by placing textual annotations linked to specific object features. Each textual annotation is displayed in a floating note with a 3D line path connecting the note with the point being referenced on the object geometry. The exact note positions and the 3D paths are automatically adjusted on a per-user basis according to each user's current viewing position, balancing the ease of associating each note with its reference point on the object and the occlusion of both paths and the object under discussion. We discuss evaluation criteria for judging well placed notes and well chosen paths, and address the challenges posed by stereoscopic vision and head tracking to optimal note placement and path generation.

1 INTRODUCTION

Using our networked design review software CHaMUE, users across the globe can gather in virtual meetings to refine product designs in real time by drawing on virtual models and adding textual annotations. The ability to draw on objects allows users to indicate a change in shape or contour of an object, or a region of an object requiring change. Textual annotations allow users to provide more precise descriptions, such as exact dimensions, and are a more natural way to specify certain design changes, such as the addition or removal of entire object features. Text can also be used to leave reminders of potential design prob-

lems for subsequent review. Drawing and text can also be used in conjunction; for example, to leave a comment about a particular region of an object, users can circle the region on the object and leave a note in the region indicating the comment.

Each textual annotation is placed in a rectangular floating note with a solid-colored background to ensure that its text is readable even when the text and the object behind it have similar colors. Each note references a single point on the surface of the object, which we refer to as the anchor point for that note. Since we usually can't place each note right at its anchor without the note obscuring some portion of the object, we allow notes to be placed anywhere in the virtual world; to allow users to easily visually associate each anchor with its corresponding note, a 3D line path connects each note-anchor pair. In cases where a straight line path between the note and the anchor would intersect the object, we need to route it around the object in 3D.

The VR hardware setup we use with CHaMUE is shown in Fig. 1. The user stands in front of a large Immersadesk display while wearing CrystalEyes VR head-tracked stereo shutter glasses. We set up the OpenGL view frustum and position the virtual object under discussion so that the object appears to float in front of the monitor; head-tracking allows us to continuously update the image so that the object appears to stay in the same position in space even as the user moves. The user interacts with the virtual object by means of a stylus at the end of a PHANToM 3.0 haptic arm mounted on top of the display. The stylus is used to touch the virtual object and manipulate textual annotations. Text is entered with a keyboard mounted on the side of the display; the integrated keyboard trackball is used to rotate the view of the object.

The major challenges in implementing CHaMUE's textual annotation system stem from the users' ability to change the
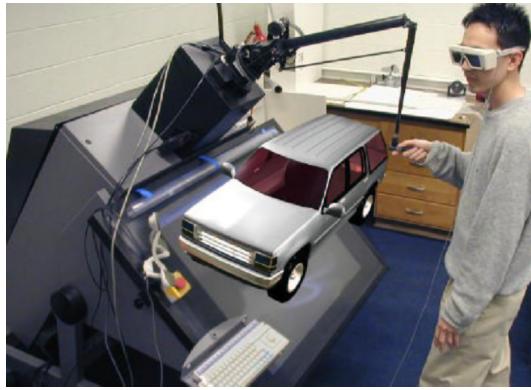
---

[*]Corresponding author.

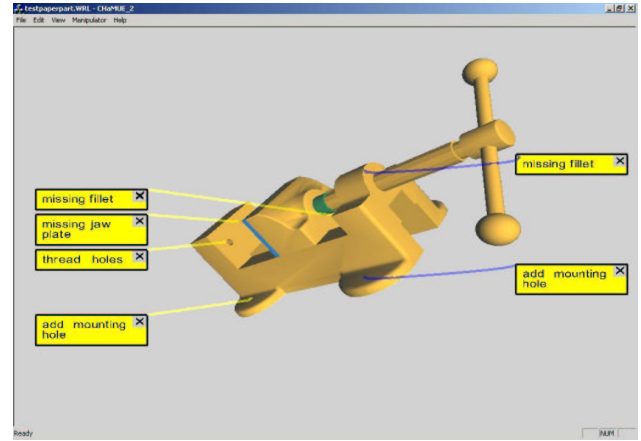Figure 1. *Hardware configuration (vehicle image simulated). The user is holding the haptic arm's stylus.*

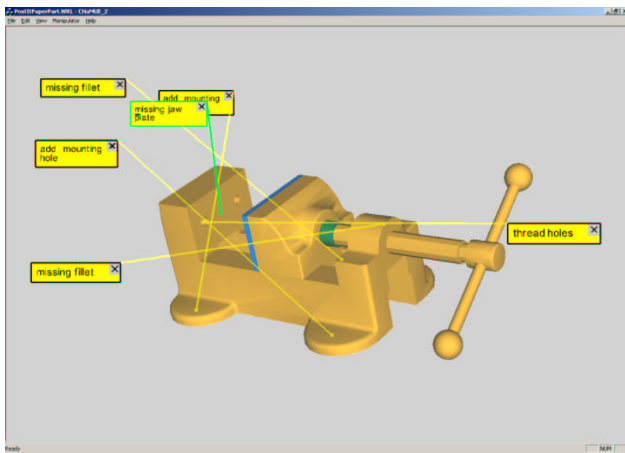Figure 2. *Poorly placed notes with hard to follow paths make it difficult to determine which notes refer to what.*

Figure 3. *The notes in Fig. 2 after using our algorithms for note placement and path generation.*

viewing position, both by rotating the virtual object and by moving their heads. As the viewing position changes, the visible portion of the world and the relative ordering of objects changes. As a result, even if notes with fixed 3D locations are neatly arranged for one viewer's initial viewpoint, a change in perspective and a quick rotation of the object frequently yield long, crossing, and difficult-to-follow paths which are virtually unusable, as illustrated in Fig. 2. The repositioning of textual annotations in commercial CAD systems is often performed manually by the user when the viewpoint changes — a tedious process that is undesirable in a design review session. In this paper, we describe the design and implementation of an automated solution which uses new note placement and path generation algorithms to automatically choose good note positions and easy-to-follow paths, as shown in Fig. 3.

In a CAD environment without VR, solving the note placement and path generation problems is considerably simpler. This is both because the viewpoint is not constantly shifting due to

head-tracking, and because stereo viewing requires a true 3D solution. Without stereo viewing, we could solve the problem in 2D by considering only the 2D projection of the current scene on the monitor screen: we could simply overwrite the scene with our annotations and straight line paths to the anchor points by writing directly to the graphics card's frame buffer. With stereo glasses, however, because the user's eyes have slightly different perspectives on the virtual world, 3D objects not located on the plane of the monitor project to different pixels on the monitor for each eye. If we just wrote our annotations and paths directly to the frame buffer in the same spot for both eyes, they would appear to be in the plane of the screen, cutting through virtual objects that float in front of the screen. Therefore for proper stereo viewing we need to determine the actual 3D locations of the notes and generate 3D paths that remain in front of the object so that they are not hidden by it. An additional challenge with stereo viewing is avoiding double vision, which occurs when a portion of the virtual scene comes too close to users; thus the seemingly simple solution of just putting all our annotations and paths in a plane right in front of the user's eyes won't work either.

Before describing our algorithms and implementation for note placement and path generation in this 3D VR context, we review related work and discuss the criteria for evaluating what constitutes a well-placed note and an easy-to-follow path.

2 RELATED WORK

Collaborative, non-immersive virtual reality systems that support textual annotation of 3D objects by allowing text written in side panels and child windows to reference locations in the virtual world include [1,2], but textual annotations on 3D scenes are more commonly seen in augmented reality systems [3, 4]. These systems apply labels (e.g. object names) or brief text to objects in the physical world, typically via a translucent, head-mounted display (HMD) capable of overlaying graphics on the

view of the real world [5]. These systems typically position text over or near the objects they describe, ignoring occlusion [6], although more advanced systems are beginning to perform occlusion calculations to determine which areas of the plane contain unimportant objects and can therefore be overwritten with annotations [7]. Drawing paths to such annotations is much simpler with augmented reality than in a virtual reality system, since the paths displayed on HMD will never be occluded by the real-world 3D objects being annotated.

Legibly labeling point features has long been a problem, even for cartographers attempting to label static, planar maps. Christensen et al. showed that the problem of placing text labels near point features while maximizing readability is NP hard [8]. Thus we conclude we should use a heuristic for note placement in our system, rather than trying to achieve the impossible goal of finding an optimal real-time solution.

Several of the criteria we use for judging the effectiveness of our choices for note placement and path generation are inspired by the graph visualization literature. Graph visualization attempts to geometrically represent graphs in order to reveal their structure and structural patterns [9,10]. Graph visualization algorithms first arrange the nodes in the target graph and then choose smooth, short, non-crossing paths to represent graph edges. Our task is similar in our attempt to clarify the structure of the data (i.e. the associations between notes and anchors) via visually appealing and easy-to-follow paths. Most existing systems for graph drawing are for 2D abstract graphs, however (see for example AT&T Labs' open source graph visualization software [11]). With our task we are constrained by the fact that anchors are fixed in space, unlike the nodes in an abstract graph, which can generally be placed at arbitrary locations; adding constraints to existing algorithms does not appear to be promising [12].

Graph layout problems that are both non-planar and include location constraints occur in VLSI path routing [13]. However, VLSI routing, which tries to find short electrical paths between related parts of circuits, typically produces jagged, step-shaped paths which are difficult for the human eye to follow because circuits usually consist of distinct layers in which wires can be routed.

3 CRITERIA FOR EVALUATING NOTE PLACEMENT AND PATH GENERATION

Before we present our algorithms for placing notes and generating paths, we consider what the ideal algorithms would achieve. Note placement and path generation are closely related tasks, and path generation often depends on good note placement to achieve good results. A well placed note is characterized by the following properties:

1. it is visible
2. it does not occlude the object
3. it is easily associated with the point on the object it describes (its anchor)

A well routed path is one which achieves property (3) above. In other words, a well routed path is one which is easy to follow. Properties which affect the ease with which a path is followed include:

1. visibility — paths are most easily followed when fully visible.
2. curvature — paths with minimal curvature, and, in particular, few changes in concavity are easier to follow. For example, a straight path is easier to follow than a "C" shaped path, which is easier to follow than an "S" shaped path.
3. smoothness — smooth, continuous paths are easier to follow than jagged paths.
4. length — shorter paths are easier to follow than longer paths, particularly when the path is partially occluded, jagged, and/or curvy.
5. crossings — paths whose projections on the screen do not cross are easier to follow.
6. path density — high path density makes it difficult to distinguish between adjacent paths because they are close together.
7. double vision — paths which are too close to the user result in extremely distracting double vision.
8. magnitude of path movement when adjusting to user movement — paths which move suddenly or jump when the user moves are not only difficult to follow, but are also distracting when the user is performing other tasks, such as drawing.

The path generation algorithm described below takes all eight properties above into account, paying particular attention to visibility and those criteria which yield simple paths: curvature, smoothness, and length. Accordingly, the simplest path (the straight path) is chosen by the path generation algorithm whenever it is visible; when a straight path is not visible, a smooth visible path with a straight projection on the screen is chosen if one exists.

Because the path generation algorithm chooses paths with straight projections on the screen, note placement tries to find note locations which are visible, do not occlude the object, and yield straight path projections which are short and non-crossing. To keep notes visible and prevent them from occluding the object, notes are placed near the left and right edges of the user's view frustum. Notes on each side are then arranged to keep paths as horizontal (and thus as short) as possible, yet prevent path projections from crossing.

The full note placement and path generation algorithms are only run upon user request. This ensures that paths and notes do not distract the user by jumping around as the user moves. When the user runs the note placement algorithm, the path generation algorithm is automatically run on the assumption that the notes need to be moved significantly and thus require new paths. Users can also run path generation without note placement if they choose.

4 NOTE PLACEMENT

The automated note placement algorithm attempts to place notes so paths with straight projections on the screen do not cross and notes do not occlude the object of interest, yet remain visible and close enough to their anchors that the user can easily associate each note with its anchor. To ensure notes are visible but out of the way, each note is placed in a column attached to either the left or right side of the user's view frustum. Each note is placed in the closer of the two columns and initially positioned so its path is horizontal, and thus of minimal length. Notes are then adjusted up and down to prevent overlapping and are vertically reordered as necessary to prevent straight path projections from crossing.

4.1 Attaching Notes to the View Frustum

Although it may seem natural to treat notes as stationary objects in the virtual world, Fig. 4 illustrates how little user movement is needed to make a note whose position is fixed in the world fall outside the view frustum when the user moves in one direction, and occlude the object when the user moves in the opposite direction. To ensure that user movement does not have this effect, we "attach" the notes to the frustum by making each note's position a function of the current frustum location and continuously adjusting as the user moves. [1]

Although the way in which we position notes relative to the view frustum helps keep them visible when the user moves, it also benefits text readability by ensuring the same pixels draw each note regardless of the user's location [14]. If notes scan convert to different pixels when the user moves, blurry text caused by rapid variations in the thickness of the line used to draw the text often results.

The coordinate system of the virtual world is oriented so that the middle of the monitor screen is the origin, the positive z axis comes directly out of the monitor, and the positive x and y axes lie to the right and above the origin (in the plane of the monitor). To place notes relative to the view frustum, we specify the position of each note in two parts. The first is a proportion p of the distance z_{user} between the user and the monitor; the plane $z = z_{note} = p \cdot z_{user}$ contains the note. The second is horizontal and vertical offsets which specify the note's location in the cross-section of the frustum with $z = z_{note}$.

4.2 Initial Placement

The note placement algorithm places every note in the same plane $z = z_{note}$ corresponding to $p \approx \frac{1}{4}$. Using $p \approx \frac{1}{4}$ typically places the notes slightly behind the closest part of the object, making the notes close enough to the user to allow easy interaction with the stylus, but far enough away that they don't occlude more of the viewing volume than necessary.

---

[1]We often refer to "the" view frustum although there are actually two view frustums (one per eye) because we are interested in keeping notes visible to both eyes; hence, "the frustum" refers to the intersection of the two frustums unless otherwise stated.
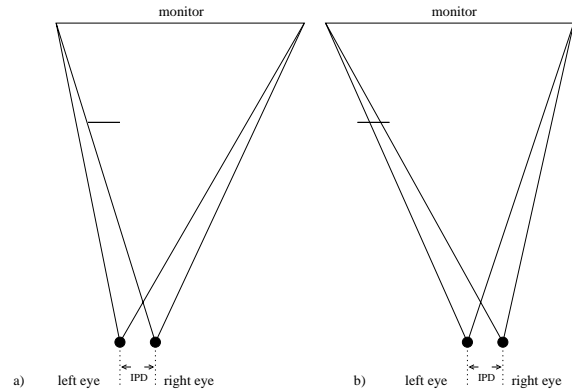

Figure 4. *Top view showing the effect of user movement and head tracking on note visibility when notes are fixed in the world and not attached to the view frustum. The horizontal line is a note whose location is fixed in the world. In (a), the note is nicely positioned so it is visible but out of the way. When the user moves to the right, as shown in (b), the note is no longer fully visible. If the user were to move to the left instead, the note would occlude the center of the view frustum, and likely much of the object.*

Our algorithm places each note in one of two columns: if the note's anchor projects onto the left half of the screen, the note is placed in a column near the left edge of the viewing frustum; otherwise, the note is placed in a column near the right edge. (In order to minimize the projected path lengths, we don't split the notes equally.) If all the notes in a given column cannot be displayed without overlapping, notes whose anchors are closest to the opposite column are moved to that column. (Strategies for displaying more notes than fit in the two columns are left to future work.) The notes in each column line up nicely because all notes are the same width (notes grow vertically to accommodate text). Placing the notes near the left and right edges of the viewing frustum reduces the chance that the notes occlude the object because the screen is wider than it is tall and our "fit-to-view" function places the virtual object in the center of the screen so it remains visible when the virtual world is rotated.

Once a note's column has been determined, the note's x and z coordinates are fixed since the plane $z = z_{note}$ containing all notes is fixed. In this case, a path's projection onto the screen is of minimal length if and only if its projection is a horizontal line. Note placement which yields such a path of minimum projection length is one form of locally optimal placement. Short path projections are desirable because they make it easy to associate notes with anchors. Fig. 5 shows that note placement resulting in horizontal path projections is locally optimal for each individual note rather than globally optimal for all notes because notes can overlap if, for example, the higher of two adjacent notes is too tall. To obtain a globally acceptable solution, notes must be adjusted up or down from their locally optimal positions to prevent overlap.
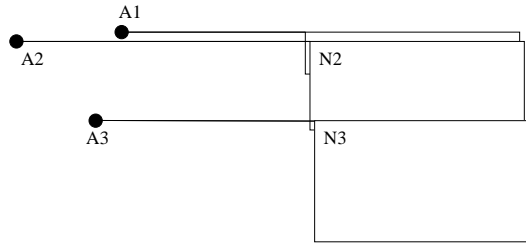
Figure 5. *Front view. Three centered, singleton groups on the right side of the view frustum which need to be merged and recentered. Notes are actually displayed in a single column, but are shown staggered to make their overlapping more clear.*
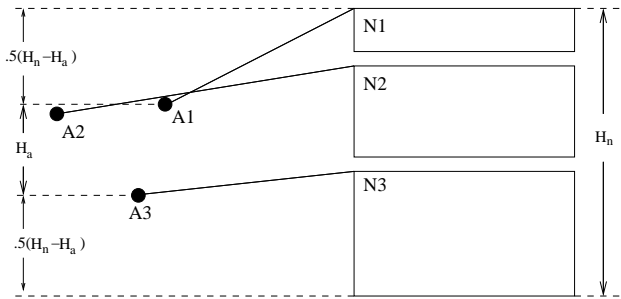

Figure 6. *Front view. Notes from Fig. 5 after merging N1 and N2, recentering, merging the result with N3 and recentering again. Note that the projections of N1's and N2's paths cross.*

4.3 Grouping Notes to Eliminate Overlap

After placing notes locally optimally, we eliminate note overlap by combining each set of overlapping, adjacent notes into a group and repositioning the notes in the group so they do not overlap. When repositioning notes, we move them up and down as little as possible so their paths remain short.

The process of repositioning notes in a group to eliminate overlap is called recentering. As shown in Fig. 6, recentering a group of more than one note spreads out the notes in that group to prevent overlap and attempts to choose short paths by placing notes so they are vertically balanced or centered around the group's anchors. The notes are placed so that the amounts by which they extend above the top anchor in the group and below the bottom anchor are equal. This equality is guaranteed unless it places notes outside the view frustum; in this case, notes are moved up or down to keep them visible. For a singleton group, the recentering method places the note in that group locally optimally.

Thus, initial placement of a note in its locally optimal position consists of placing the note in its own group and centering the group. The grouping algorithm then sorts the groups by their note locations so adjacent notes are in adjacent groups. Adjacent, overlapping groups (i.e. adjacent groups containing overlapping notes) are then merged and recentered to eliminate note overlap.

Nonadjacent groups need not be checked for overlap because if two nonadjacent groups overlap, then all groups in between must overlap as well. To see this, note that if there exist nonadjacent, overlapping groups, all anchors belonging to groups in between are between the two groups' anchors because all groups are initially ordered by their locally optimal placements (i.e. their anchor locations); since all groups are centered and the two groups overlap, the notes belonging to groups in between must also overlap.

The algorithm keeps track of the current group g being processed and if g overlaps an adjacent group, the two groups are merged and the resulting group is recentered. If g does not overlap either adjacent group, we process the group below it. Pseudocode for grouping a single column of notes is given in GroupNotes (Algorithm 1). Note that we must check the group a above g for overlap even though a has already been processed because recentering g may move it up, causing it to overlap with a.

4.4 Eliminating Path Crossings within Each Group

After GroupNotes partitions the notes into non-overlapping groups, the notes within each group must be ordered so that the projections of their paths onto the screen do not intersect (which they often will, as shown in Fig. 6). To ensure no path projections cross, we only need to eliminate crossings among paths in the same group, since paths from different groups cannot cross. To see that the projections of paths from different recentered groups cannot cross, consider a group's note height (the sum of the notes' heights, including gaps, or H_n in Fig. 6) and anchor height (the vertical distance between the top and bottom anchors, or H_a in Fig. 6). If a group's note height is greater than its anchor height, recentering ensures the y coordinate of each anchor's projection is between the y coordinates of the top and bottom notes' projections. Thus, if no groups overlap, every group is recentered, and every group's note height is greater than its anchor height, the projection plane is partitioned so each group's notes, paths, and anchors lie in a single partition.

To see that note height is greater than anchor height for any group created by GroupNotes, first observe that the proposition holds for singleton groups because their anchor height is zero. Next, notice that larger groups are only created by merging two overlapping, recentered groups and assume the proposition holds for these two groups. In this case, none of the top group's anchors are above or below all notes in the top group; likewise for the bottom group. So the distance d from the top note of the top group to the bottom note of the bottom group before merging is greater than the anchor height of the merged group. Since the two groups overlap, the note height of the merged group is greater than d, so the note height of the merged group is greater than the anchor height of the merged group. Thus, if all groups are recentered and non-overlapping, the projections of paths from different groups cannot intersect, and only path crossings within each group need be eliminated.

```
Input: nc, a column of notes
Result : s, a sequence of non-overlapping groups of
         notes
Sequence<Group> s = {}
foreach note n in nc do
    put n in its own group
    recenter the group (to place n locally optimally)
    add the group to s
end

// sort groups top to bottom by their notes' locations
Sort(s)

Group g = first group in s;
// loop invariants:
// a is always the group above g (so a is before g in s)
// b is always the group below g (so b is after g in s)
while true do
    if a overlaps g then
        merge a and g
        recenter the new group
        set g to be the new group

    else if b overlaps g then
        merge b and g
        recenter the new group
        set g to be the new group

    else if g is not the last group in s then
        set g to be the group after itself in s

    else
        return s
    end
end
```

Algorithm 1: GroupNotes().

Eliminating path intersections within a group only involves choosing the order of the notes in that group, since the actual area occupied by a group's notes is determined by the recentering method in GroupNotes. To eliminate path intersections, we must first determine each note's path attachment point — the point on the note to which the line path attaches. The path attachment point of a note on the right side of the screen is the top left corner of the note; the attachment point of a note on the left side of the screen is the top right corner. The first note chosen by the reordering algorithm is placed at the top of the group's area, and the remaining notes are placed below. When deciding which note to place, the algorithm considers, for each candidate note, the angle formed by the ray extending up from the next note's path attachment point and the straight path connecting the attachment point to the anchor. As shown in Fig. 7, the next note placed is the one whose angle is minimal.

Proof of correctness: As shown in Fig. 7a, let l be the (infinite) line coinciding with the path of the note just chosen by the algorithm. Since the note with minimal angle was chosen, no unplaced notes in the group have anchors in the half plane above l. Thus no path of any note chosen later can intersect l or the path just chosen. Each unplaced note's angle must be recalculated in each iteration of the loop since the angle depends on the path attachment point.

Once notes have been positioned, we generate the paths between anchor points and notes.

5 PATH GENERATION

The path generation algorithm chooses paths with straight projections on the screen since note placement guarantees such paths will not cross. Paths are either straight lines or C^1 continuous curves which project to straight lines at the time of generation. Straight paths and C^1 curves are used because they are simple, smooth, and easy to follow. Straight paths are used if they are fully visible. If some part of the straight path is occluded by the object, a curve is routed in front of the object as long as both path endpoints are visible. If one of the path's endpoints is occluded, the entire path will never be visible, so the benefit of a curved path is limited and the straight path is used instead.

The need for curved paths varies with object geometry; a simple sphere, for example, is far less likely to partially occlude straight paths than a complicated part with concavities. But even on symmetric convex objects, straight paths attached to the closest part of the object (which may well be the most important feature placed at the center of interest) are often partially occluded by adjacent faces because notes are typically farther from the user than the closest part of the object. Notes are not placed closer to the user than the object because doing so would cause them to occlude too much of the viewing volume and the object of interest.

Path generation, like note placement, only occurs at the user's request. Although the note placement and path generation algorithms typically yield smooth, visible curves with non-crossing projections at the time of generation, visible and non-crossing paths are not necessarily maintained once the user or object moves. However, the nearly horizontal paths produced by the algorithms do not frequently cross due to the way the user interacts with the system. Since users typically remain standing in front of the display screen, their vertical position rarely changes. Users rarely move forward or backward because they need to maintain a fairly constant distance from the object in order to comfortably manipulate it with the haptic arm. Thus users move primarily left and right, so the largely horizontal paths do not frequently cross.

5.1 Curve Styles

We have implemented two types of curves — adaptive and semi-fixed. Both adaptive and semi-fixed curves are initially calculated the same way, are composed of two quadratic Bezier splines with a straight line in between, and look identical as long
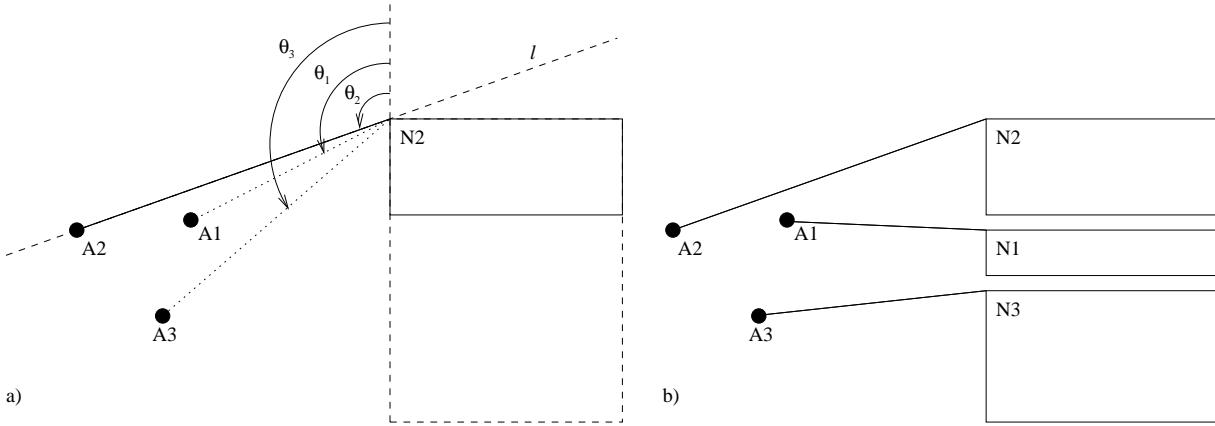
Figure 7. *Front view. Reordering three notes to eliminate path crossings. (a) The process of choosing the top note in the group. N2 is chosen as the top note because Θ_2 is minimal. The dashed box below N2 indicates the region in which the remaining notes will eventually be placed. All unplaced anchors (A1, A3) are below l. (b) The result of reordering.*

as the user does not move. When the user moves, each adaptive curve adjusts so it projects to a straight line. Semi-fixed curves, on the other hand, do not significantly adapt to the user's movement. Notes are attached to the view frustum, so all notes and paths move slightly as the user moves, but semi-fixed curves only change enough to remain C^1 continuous; only the segment of a semi-fixed curve closest to the note changes with the user's position.

5.2 Initial Calculation of Curves

Fig. 8 shows the curve for a given anchor location A and path attachment point N, along with the values calculated in the process of choosing the path. Each curve lies in the triangle T with vertices A, N, and E (the user's eye location). Notice that T projects to a straight line on the screen, so any curve in its interior will also project to a straight line. To calculate the final path, we first find the smallest triangle U which has A and N as vertices and contains the part of the object in T. We define the foremost vertex (FV_U) as the third vertex of U. Although a path routed in front of triangle U is fully visible, it can come unnecessarily close to the user, as it would in Fig. 8, possibly resulting in double vision. To keep the path as far as possible from the user, we calculate max_z, the largest z coordinate of the object in T, and place the path on the far side of the plane $z = max_z$.

Let A_T and N_T be the two intersections of the plane $z = max_z$ with triangle T, and let A_U and N_U be the intersections of $z = max_z$ with the two (infinite) lines passing through the sides of U adjacent to FV_U as shown in Fig. 8. The curve is composed of a straight line with endpoints A_U and N_U, a Bezier spline with control points A, A_T, and A_U, and a second Bezier spline with control points N, N_T, and N_U. This path is almost as far from the user as the object, and thus does not produce double vision unless the object itself does. Note that the curve is C^1 continuous
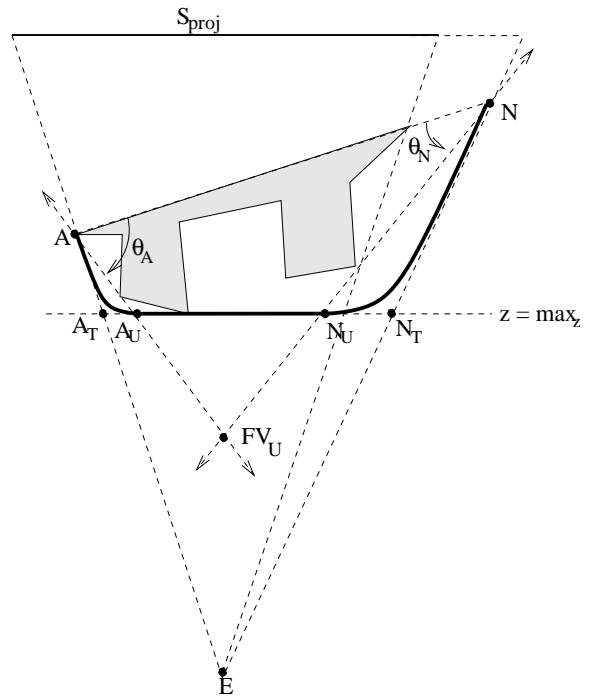

Figure 8. *Top view. Generating a path around the shaded object in the triangle T whose vertices are the anchor (A), path attachment point (N), and user's eye location (E). The foremost vertex FV_U is the third vertex of the triangle U determined by A, N, Θ_A, and Θ_N. S_{proj} (drawn as if in the plane of the screen) consists of the pixels displaying the shaded object.*

because A_T, A_U, N_U, and N_T are collinear, A_U and A_T are distinct, and N_U and N_T are distinct.

To calculate the foremost vertex and thus triangle U, the object is drawn to the graphics card and depth information from the

graphics card is used to determine the location of the object in triangle T. T projects to the straight line between the note and anchor, so the pixels S_{proj} on the straight line draw the part of the object in T as shown in Fig. 8. The set S of points in space drawn by S_{proj} contains the visible points on the object's surface in T and is calculated from S_{proj} using the graphics card's depth information and OpenGL's gluUnProject function. Set S defines two sets of angles: set $\angle NAS$ and set $\angle ANS$. Fig. 8 shows how A, N, and the maximums Θ_A of $\angle NAS$ and Θ_N of $\angle ANS$ determine U and the foremost vertex. Algorithm 2 provides pseudocode for `BoundObject`, which calculates the foremost vertex and max_z for a note-anchor pair using the graphics card's frame buffer.
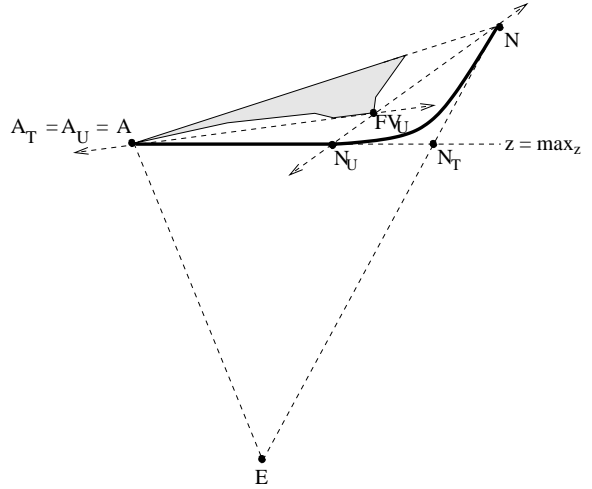

Input: A, anchor position
  N, path attachment point
  E, the user's eye position
Result : the foremost vertex and maximum z-coordinate of the object in the triangle (A, N, E)

A_{proj} = screen pixel which draws A
N_{proj} = screen pixel which draws N
S_{proj} = {pixels on the straight line between A_{proj} and N_{proj}}
S = {gluUnProject(p) : $p \in S_{proj}$}
Θ_A = max($\angle NAS$)
Θ_N = max($\angle ANS$)
max_z = max{$s.z : s \in S \cup \{A\}$}
$foremost\,vertex$ = third vertex in triangle defined by A, N, Θ_A, Θ_N

Algorithm 2: BoundObject(). Note: all points referenced in this algorithm lie in the plane passing through the anchor, path attachment point, and user's eye.

Note that if the foremost vertex is farther from the user than the plane $z = max_z$ as shown in Fig. 9, the Bezier spline adjacent to the anchor degenerates to a single point since $A = A_U = A_T$. The only special case occurs when the side of U between the anchor and the foremost vertex coincides with $z = max_z$ (since we don't get a single intersection point between the line and the plane containing it in this case). In this case, we define $N_U = FV_U$ and $A_U = A$ so that we get a curve similar to the above case consisting of a straight path from $A_U = A$ to $N_U = FV_U$ and a fully visible quadratic spline with control points $N_U = FV_U$, N_T and N.

5.3 Adjusting Curves after User Movement

The path generation algorithm saves max_z and the four control points A_U, A_T, N_U, and N_T. After the user moves, semi-fixed curves are drawn in the same manner as at the time of generation using these original four control points, plus the anchor and new

Figure 9. *Top view of generating a path around the shaded object when the foremost vertex FV_U is farther from the user than the plane $z = max_z$. Note that the anchor-side spline is degenerate because $A = A_T = A_U$.*

note position. If the user moves, the note's 3D position will adjust so that it continues to project to the same 2D pixels on the screen; thus the semi-fixed curve will not necessarily continue to lie in a plane nor project to a straight line.

Adaptive curves are drawn after repositioning A_U, A_T, N_U, and N_T so they once again lie in triangle T (which is itself adjusted to compensate for the user and note's new locations). A_U, A_T, N_U, and N_T are adjusted to lie in T by first modifying their y coordinates so they lie in the plane P containing T (in other words, vertically projecting them onto the plane). As Fig. 10 shows, each of these points that still falls outside T is then replaced with the intersection between $z = max_z$ and the side of T adjacent to the control point. For example, if either A_U or A_T is outside T, it is replaced with the intersection between $z = max_z$ and the line through the anchor and the user's location. If plane P is vertical, making it impossible to vertically project all control points onto P, a straight path replaces the adaptive curve. (This causes the path to jump, however; an alternative is to use the path displayed at the last eye position for which P was not vertical, even though this means the path won't quite project to a straight line.)

Control points are moved up and down to put them in plane P because paths are largely horizontal and close to the object, so moving control points left or right typically causes paths to become obscured. We restrict control points to always lie in T, rather than just plane P, to ensure the anchor and note are the endpoints of the path projection. If a control point is in P but not in T, the resulting path doubles over on itself and becomes difficult to follow because its two overlapping portions have the same projection; for example, in Fig. 10, if the projection of the original path onto P is used when the user is located at E_{new}, part of the anchor-side spline will double over on itself.
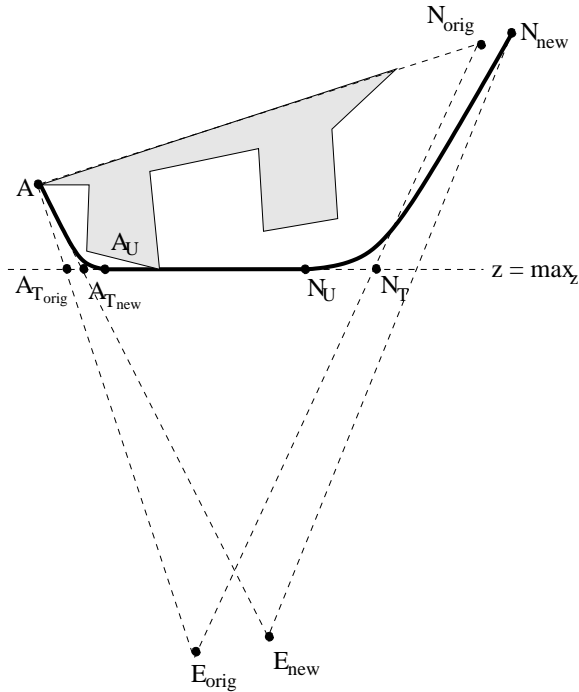
Copyright © 2004 by ASME

Figure 10. *Top view of repositioning an adaptive curve after the user moves. E_{orig}, N_{orig}, $A_{T_{orig}}$, A_U, N_U, and N_T represent the locations of the user, note, and control points at the time of path generation. E_{new}, N_{new}, $A_{T_{new}}$, A_U, N_U, and N_T represent the corresponding locations after the user moved. Note that N_U, N_T, and A_U, as shown in the diagram, represent the control points used to draw both the new curve and the original curve before the user moved; the control points used to draw the two curves are different, but appear the same from the top view since the new are obtained by vertically projecting the original.*

5.4 Comparison of Curve Styles

Although semi-fixed and adaptive curves are fairly similar, adaptive curves have the advantage that they tend to cross each other less often, whereas semi-fixed curves have the advantages that their largely stationary nature causes them to intersect the object less frequently and allows their 3D position to be more easily understood due to their more expected behavior when users move their heads.

The projections of semi-fixed curves are more likely to cross than those of adaptive curves because of the shape of their projections. Because semi-fixed curves project to arcs when the user moves, they cross in many cases when the straight projections of adaptive curves do not. Crossings are especially frequent when users stray far from their location at the time of path generation, causing semi-fixed curves to project to large arcs that cross many more curves than a straight projection.

Although the projections of adaptive curves are less likely to cross, they are more likely to intersect the object when the user moves. This is because the parts of the path often closest to

the object — the straight segment and the anchor-side spline — move with the user while the same parts of semi-fixed curves do not move.

Semi-fixed curves take greater advantage of head tracking than adaptive curves, making it easier for the user to determine their location. The projections of adaptive curves maintain their shape as the user moves, thereby eliminating most of the benefit of head tracking and making it difficult for the user to determine the exact location of the curve in space. With semi-fixed curves, users can easily determine the position of the curve in space by noting how the image changes as they move. Without the benefit of head tracking, curves can be difficult to locate because they are close to horizontal, and thus look almost the same to the left and right eyes. In preliminary testing, users preferred semi-fixed curves because they could use head tracking to determine the 3D location of curves. Furthermore, increased path crossing with semi-fixed curves did not significantly bother users.

6 FUTURE WORK

Although our algorithms initially generate visible paths, they do not guarantee path visibility when the user moves. Our full path generation algorithm cannot be re-run for every frame because the object's bounding information is obtained from the graphics card's depth buffer, a process which takes about a third of a second on the Evans and Sutherland Tornado 3000 graphics card in our system, whereas we run with a 30 Hz frame update rate. However, upcoming computers with faster video cards and higher bandwidth interconnects such as PCI Express X16 may enable continuous querying of the depth buffer. If path generation is performed continuously, however, significant care must be taken to prevent paths from jumping around as the user moves, because such behavior will make the notes distracting. A naive implementation of continuous path generation would be particularly distracting if a user is shifting back and forth so that a straight path is sometimes visible and sometimes occluded by a large object; in this case, a naive path generation algorithm, in an effort to maintain visibility, would likely switch back and forth between the straight path and a large curve in front of the object. Hence, future work includes finding better path generation algorithms which ensure paths are always visible (or at least mostly visible), yet do not jump when the user moves.

7 CONCLUSION

In light of the criteria for evaluating note placement outlined in Section 3, we believe our note placement algorithm accomplishes:

1. visibility — notes are guaranteed to remain in the view frustum regardless of the user's location and are typically visible because they are placed at the sides of the view frustum, unlike the object, which is generally in the center of the screen.
2. low likelihood of occluding the object — notes remain near

the edges of the frustum, unlike the object, which is generally at the center of the screen.

3. easy association of notes and anchors — notes and anchors are easy to associate because paths are fairly easy to follow.

Assuming the object is not moved after path generation, the path generation algorithm yields fairly easy to follow paths because it achieves:

1. visibility — paths, at the time of generation, are entirely visible unless one or both endpoints is not visible. Semi-fixed curves are likely to remain visible when the user moves because the section of the curve closest to the anchor, and thus the section often closest to the object, does not move when the user moves.

2. minimal curvature — given the note locations chosen by note placement, straight paths are used if visible. Otherwise, at the time of path generation, paths hug the object geometry and do not have inflection points. After the user moves, paths largely retain their shape, although a semi-fixed curve will have an inflection point if the user backs up enough to cause the note to cross the plane $z = max_z$.

3. smoothness — all curves are C^1 continuous at the time of generation and remain so except in one special case: adaptive curves are only piecewise C^1 continuous when one of the quadratic Bezier splines degenerates to a straight line. However, C^1 continuity is of limited value in this case because this case only occurs when the anchor is not visible or the path attachment point is likely not visible.

4. short paths — given the note locations chosen by note placement, paths are short because they have minimal curvature and their projections are close to horizontal.

5. minimal crossings — path projections do not cross at the time of path generation, and as argued in the introduction to Section 5, paths are not likely to cross when the user moves because user movement is largely lateral and paths are largely horizontal.

6. low path density — the notes get spread out in the side columns to avoid overlapping and as a result, path density is fairly low.

7. rare double vision — paths never come closer to the user than the object at the time of routing, and thus do not produce double vision unless the object itself does.

8. little movement when adjusting to user movement — paths never move suddenly or jump in reaction to user movement except in one very rare case: adaptive curves switch to straight paths when the user, anchor, and path attachment point lie in a vertical plane.

In an effort to balance the often contradictory needs for visible, non-occluding notes and easy to follow paths, our algorithms first choose note locations that are likely visible and non-occluding, that guarantee paths with straight projections will not cross, and that yield relatively short path projections. We then attempt to construct smooth, visible paths with straight projections

and minimal curvature. We avoid the extremes of guaranteeing visible, non-occluding notes or choosing the easiest paths to follow because the former can result in long, hard to follow paths while the latter entails omitting line paths altogether and placing notes at their anchors, which often causes notes to occlude the object.

The above work enumerates the desirable qualities of an annotation system in a head tracked, stereoscopic environment and describes an implementation of such a system which produces readable, yet unobtrusive notes which gracefully adjust to user movement and allow users to easily associate annotations with the geometry they describe. We hope this system and others like it will improve the usefulness and frequency of online collaboration, thereby eliminating the travel time required to allow remote parties to physically meet and increasing productivity.

REFERENCES

[1] Craig, D. L., and Zimring, C., 2002. "Support for collaborative design reasoning in shared virtual spaces". In Automation in Construction, vol. 11, pp. 249–259.

[2] Jung, T., Gross, M., and Do, E. Y.-L., 2002. "Annotating and sketching on 3d web models". In International Conference on Intelligent User Interfaces (IUI), pp. 95–102.

[3] Fiorentino, M., de Amicis, R., Monno, G., and Stork, A., 2002. "Spacedesign: a mixed reality workspace for aesthetic industrial design". In Proc.of the IEEE and ACM International Symposium on Mixed and Augmented Reality, pp. 86–96.

[4] Schmalstieg, D., MacIntyre, B., and Takemura, H., Eds., 2003. The Second IEEE and ACM International Symposium on Mixed and Augmented Reality.

[5] Feiner, S., MacIntyre, B., and Seligmann, D., 1992. "Annotating the real world with knowledge-based graphics on a see-through head-mounted display". In Graphics Interface, pp. 78–85.

[6] Cornelis, K., Pollefeys, M., Vergauwen, M., and Van Gool, L., 2001. "Augmented reality using uncalibrated video sequences". vol. 2018 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 144–160.

[7] Bell, B., Feiner, S., and Hollerer, T., 2001. "View management for virtual and augmented reality". In UIST, pp. 101–110.

[8] Christensen, J., Marks, J., and Shieber, S., 1995. "An empirical study of algorithms for point-feature label placement". ACM Transactions on Graphics, **14** (3) , pp. 203–232.

[9] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G., 1999. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

[10] Liotta, G., Ed., 2003. Graph Drawing: 11th International Symposium, GD 2003, vol. 2912 of *Lecture Notes in Computer Science*, Springer-Verlag.

[11] AT&T Labs, 2003. Graphviz. http://www.research.att.com/sw/tools/graphviz.

[12] Tamassia, R., 2000. *Handbook of Computational Geometry*. Elsevier, ch. Graph drawing, pp. 937–971.

[13] Lengauer, T., 1990. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, ch. Introduction.

[14] Karasuda, E., and McMains, S., 2004. "Displaying readable object-space text in a head tracked, stereoscopic virtual environment". *Submitted for publication* .