Using 3D Tubes to Solve the Intersecting Line Representation Problem

Lindsey Spratt and Allen Ambler

Pages 254-263 in *Proceedings 1994 IEEE Symposium on Visual Languages,* October 4-7, 1994. St. Louis, Missouri.
IEEE Computer Society Press: Los Alamitos, CA.

Abstract

In this paper we discuss 3D rendering as a solution to the line crossing problem associated with using connecting lines for illustrating relations in visual programming. The Bezier tube was introduced for modeling connecting lines in general and hyperedges in specific and an algorithm is given for laying out hyperedges. 3D hyperedges are shown to have features that enhance their readability over that of 2D representations of connecting lines.

Introduction

A problem common to visual programming languages is the need to display and edit relationships among elements within pictures or "scenes" that are the visual representation of some part of a program. Using 2D representations, the programmer may edit scenes in four ways:

(1) the programmer may draw the program representation “free hand” and have the system infer what the intended program is;

(2) the programmer may draw the program representation using system-provided elements (box, circle, line, text, etc.) and have the system infer what the intended program is;

(3) the programmer may draw the program representation using system-provided semantically-specific elements (statement boxes, variable ovals, etc.), positioning them and sizing them explicitly; or,

(4) the programmer may indicate the program representation by specifying semantic modifications (“add a statement”, “change variable reference”, “reference a function”) with automatic layout, an extreme version of a syntax-directed editor.

Representing Relationships.

There are several reasons that a layout may be incomprehensible. A common problem is simply fitting all of the program elements of interest on the screen. If a program is sufficiently complex such that it has many “tightly” interacting elements, there just may not be room to show all of the elements and their interactions at one time. Another common problem is showing relationships among elements.

These two problems are related. A representation which must show that two items are related has relatively few general purpose options. The major ones are connecting related items with a line of some type, having the representation of one item visually contained within the representation of another item, and using same appearance for the same visual or textural appearance for related items (e.g., same color, same shape, same text). All of these options have their problems.

Connecting lines create a graph layout problem. Line crossings are inevitable unless only planar graphs are allowed and even then the need for a representation that is flexible about the relative positioning of the “nodes” is a problem. Directed planar graphs can be laid out without line crossings, but the directed nature of the graph is generally obscured by such a layout.

Simple containment (for representing “tree” relationships) is easy to layout if screen space is not an issue. However, it uses up screen space faster than connecting lines. This is in part due to the necessity of leaving part of a container blank when a regular layout of that container’s contents don’t use up the available space in the container. For readability one wants to make containers of the same kind be obviously similar in appearance (making it easy to “read” the similarity of kind), thus they should have very similar shapes. This pushes one to avoid changing the shape of a container simply to more efficiently use screen space. Generalized containment allows overlapping containers, which can represent directed-acyclic-graphs instead of only trees. Generalized containment is much harder to layout; it is

much like the graph layout problem and it consumes screen space more rapidly than do connecting lines.

The "same appearance" approach requires that relations between items have the same appearance. It is limited to representations of equivalence classes with relatively few classifications of items and relatively few items overall, wherein each classification is readily distinguished from all of the other classifications. The relatively few classifications requirement springs from the difficulty of distinguishing very many different "appearances". Also, there should be relatively few items overall as the viewer must search through all representations to locate all of the items having the same appearance. If there are many items and/or the distinguishing aspect of the appearance is difficult to see (e.g. very small shapes, small variation in color), then this search becomes tedious.

Many representations use all three of techniques for different aspects of the programming language. In PHF [3], "same text" and "same icon" is used to classify particular operators such as "plus" or "times", containment shows "subroutine" relationships, and connecting lines show data flows. For the editing environment, the programmer draws program elements using system-provided, semantically-tagged shapes, explicitly positioning and sizing them.

Overall, the attractiveness of lines is that:

(1) they don't require searching as does "same appearance",
(2) they are more flexible for displaying various relations than is containment, and
(3 they don't use up screen space as rapidly as does containment.

Thus, using lines to show relatedness is clearly an interesting approach in spite of the difficulty of generating a layout which avoids line crossings. This brings us to the point of this investigation. Since the main objection to lines is their crossings, and the main problem with crossings is the visual confusion created by a lack of depth in 2D representations, can we by the use of 3D representations reduce or eliminate this visual confusion?

Requirements for a 3D Line Representation

Since crossing lines are inevitable in many representations, some technique to make them less confusing is needed. We propose to use a 3D representation that will clearly delineate line crossings, thereby removing the visual confusion associated with 2D representations of lines.

One of the challenges of providing such a 3D representation is providing the programmer with a convenient way to edit scenes. With 3D representation, scenes can be enormously complex to define, even when there are relatively few (10 or 20) elements. For 3D representation, either the scenes must be extremely simple, or the system must provide substantial aid in generating them. A "semantic-modification" environment, which does all of the layout, provides the most such aid. This approach is feasible if there is an appropriate modification scheme. A modification scheme is "appropriate" if it is usable, and if at any distinguished part in a representation there are only a "small" number of modifications possible, and if this scheme can be used to create all valid programs. Not all programming language representations lend themselves to such a modification scheme.

Fully automated layout is feasible if an algorithm can be implemented which can find a comprehensible layout for most valid programs in a reasonable amount of time and space. For those valid programs which the algorithm cannot find a comprehensible layout there must be an alternative program which achieves the same ends as the difficult program and for which the algorithm can find a comprehensible layout. This last condition allows the system to "require" the programmer to use some kind of modularization to achieve a readable program representation. The "most valid programs" phrase means "most of the valid programs a programmer is likely to write". This means that the modularization requirement of the layout algorithm should at most infrequently force a programmer to divide something into modules only to aid in layout comprehensibility.

Choice of 3D Line Rendering Technique

A 3D representation is a rendering of a scene. The scene describes some geometry and associates texture information with the surfaces of this geometry. The rendering uses a lighting model of the scene (positioning and kinds of the lights) and a viewing model (a "camera" pointed at some point in the scene) to produce a 2D image for display. Thus, there are many variables to control in defining a 3D representation, many more variables than are needed to define a 2D representation. The rendering aspect of the representation can be (and probably should be) controlled by the viewer, although the viewer should be given reasonable default behavior by the rendering process. The interested reader should see [4] for a detailed discussion of 3D rendering.

There are many different techniques for rendering a scene. Choosing a rendering technique is one of the major challenges in providing a 3D representation. These techniques vary greatly in their speed, in the quality of the image produced, in the kinds of geometry, texturing, and lighting they handle. There is no best technique for all rendering applications; there sometimes isn't even an

obviously best technique for a particular application. The highest quality rendering is called "photo realistic" - it looks just like a photograph of a real scene. This quality of rendering is still unobtainable for scenes in general; but, for scenes which only use certain kinds of geometries and certain kinds of texturing, photo-realism is achievable.

The choice of rendering technique is dependent on performance requirements, desired geometries, desired texturing, and desired lighting effects (shades or shadows, diffuse and/or specular, reflection, refraction). The performance requirements need to take into account whether the viewer has a movable point of view, or if a single point of view on a scene is sufficient. Generally, applications which render images provide a choice of techniques which differ primarily in their performance. This allows the user to view many rough drafts of a scene quickly, then to very slowly generate a high-quality final version.

This choice of rendering techniques is further complicated by the availability of specialized hardware to speed up certain techniques. Thus, one's choice of technique is also dependent on the hardware platform for the programming system.

Ray tracing and radiosity are two of the most realistic techniques [4]. These have different strengths and they are difficult to combine. Ray tracing is good at shiny and transparent surfaces - reflection and refraction. Radiosity is good at diffuse lighting effects. Radiosity has the additional advantage of allowing one to calculate the lighting effects independent of the observer's point of view. Thus, it lends itself to making many different views of the same scene. Radiosity is difficult to apply to general models (i.e. Constructive Solid Geometry (CSG) models, and ones with curved surfaces other than a cylinder), while ray tracing works well with CSG and a variety of curved surfaces (any surface described by a quadratic implicit equation - e.g. cylinders, spheres, and cones). Ray tracing is generally easier to implement than radiosity. Ray tracing is currently the photo-realistic rendering technique most widely used.

Other, less realistic techniques include z-buffer and spanning scan-lines [4]. The z-buffer technique is very popular because it is very fast. This is frequently the technique which specialized graphics hardware supports. The spanning scan-lines technique is less fast, but is a little more realistic. Both of these techniques are widely used.

Tubes: 3D Connecting Lines.

SPARCL [5] is a visual logic programming language based on sets. It uses all three relatedness techniques discussed above, but heavily depends on connecting lines. SPARCL uses "same text" to classify predicate names (clauses and literals), "same shape" or "same texture" is to classify term types (variables and constants), containment to show elements in a set, parts of a partitioning, and literals in a clause, and connecting lines to show term co-reference (all terms in the same hyperedge unify).

There are several 3D representation techniques that have been employed in SPARCL. First, items connected by a line are equivalent (i.e., they unify). There may be many items that are intended to unify with each other (in a textual language these items could be references to the same variable name). SPARCL uses a hyperedge to connect these many items (a hyperedge is an "edge" that connects more than two items). There may be several hyperedges in a single scene, but generally there are fewer than 10. These hyperedges have the same crossing problem common to connecting lines. In SPARCL, these crossings are made less confusing in several ways. Hyper-edges are depicted as smoothly curving tubes which curve through all three dimensions. A tube is composed of one or more segments. The segments join each other smoothly (their tangent lines are identical at the join point -- the geometry of these tubes is discussed in more detail below). Each hyperedge is a different color, i.e., all of the segments of a hyperedge are the same color, segments of different hyperedges are different colors. Hyperedge tubes do not actually intersect. Generally the tubes rise to different heights above the basic plane of the "program". Viewed from some particular angle, such as "in front of" the program, these tubes will appear to cross each other. But, if the viewer changes her view point the crossings will change. The tubes are shaded and shine according to the lighting, and they cast shadows on each other and the other program elements.

Figure 1 shows part of a SPARCL program for implementing the ID3 machine learning algorithm [10] using a 2D representation. It requires two clauses. Under-standing these clauses is not important here; we are only interested in noting the complexity of the line crossings, particularly in the larger clause. Figure 2 shows the first clause for this same program using 3D representation.

There are several aspects of SPARCL's approach that help the viewer comprehend what items are equivalent in spite of the line crossings. The main problem is to easily identify which apparent intersections of segments are accidental crossings and which are joins. The common colors (obviously, not visible here in black and white) are a simple and effective way to see that two segments do or do not belong to the same hyperedge. The smooth joins within a hyperedge help the viewer to distinguish a crossing that is "accidental" as most accidental crossing

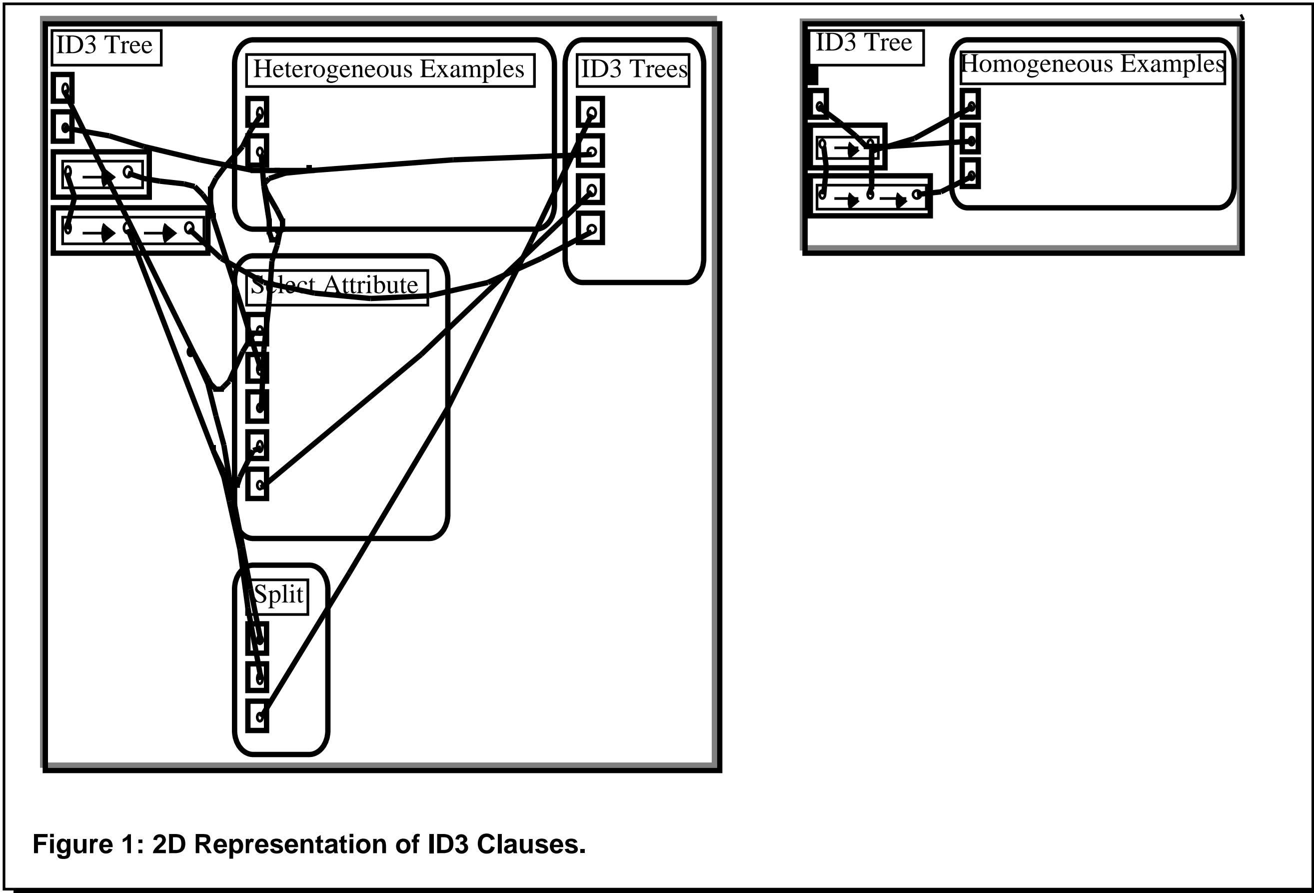


Figure 1: 2D Representation of ID3 Clauses.

are obviously not smooth. The shading, highlights, and shadows all help the viewer discern that two segments accidentally cross since they are not in the same location of the scene. The highlights and shading are different due to a different viewing angle. If the viewer moves her viewpoint, then many of the accidental crossings will change relative position. Also, the view dependent lighting features (highlights) will also change their relative position. The texturing of the segments can help further if they are reflective or translucent. If reflective, then one of them may reflect the other in its surface. If translucent, then the one passing behind the other will be dimly visible through the front one.

These aspects of a 3D scene with tubes for connections provide the viewer with many different cues for interpreting the scene. These cues are all ones that people are used to interpreting in dealing with real scenes and, consequently, require no conscious effort on the part of the viewer to utilize them even without explanation. Thus, a very complex collection of hyperedges can be understood more readily in 3D than in 2D.

Since line crossings are not as devastating to comprehension in 3D as they are in 2D, graph layout programs can allow more crossings to occur as a tradeoff for other improvements (such as better node placement, or a simpler or faster layout algorithm).

Bezier Tubes.

The segments of a hyperedge are a “Bezier tube”. The 3D model of a Bezier tube is a pair of bicubic Bezier patches [4] (for convenience called “top” and “bottom”). A bicubic Bezier patch has 16 “control points” which determine the shape of the patch. This can be seen in Figure 3. If these control points are arranged in a four by four matrix, then the points in the outer rows and columns of this matrix correspond to the four outer edges of the patch: e.g., the four control points in the top row determine the shape of a Bezier curve which is an edge of the patch. The tube is constructed by having the top and bottom patch have the same control points in their outer columns: column 1 of the top patch’s control matrix equals column 1 of the bottom patch’s control matrix. The bottom patch’s “interior” control points are a special “inversion” of the interior control points of the top patch.

The model for a Bezier tube is generated from a cubic Bezier space curve. The cubic Bezier space curve uses only four control points, one at each end and two in the

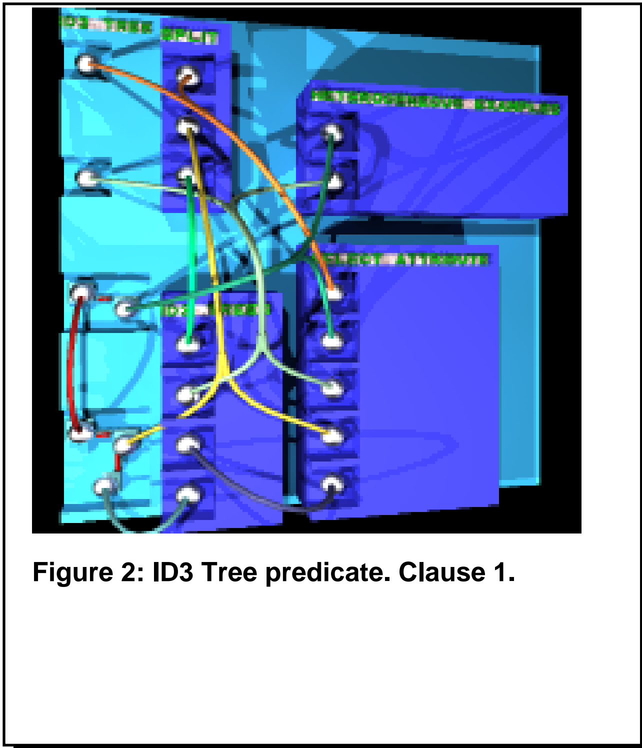


Figure 2: ID3 Tree predicate. Clause 1.

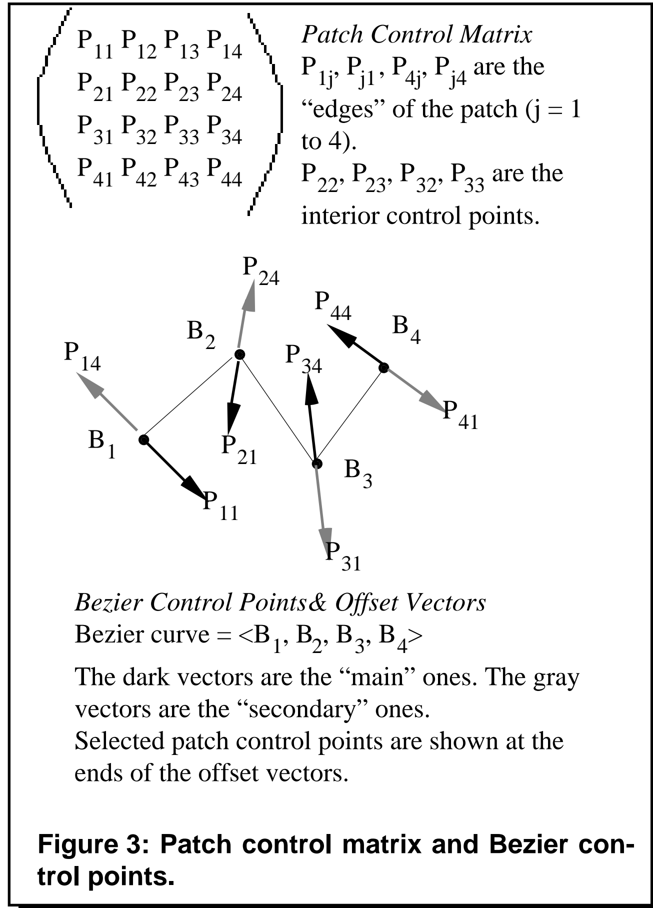


Figure 3: Patch control matrix and Bezier control points.

middle. The Bezier curve is defined to start and end at end control points, and to smoothly approximate the middle control points. This provides a simple way to design smoothly curving lines. With careful choice of control point values, Bezier curves can be joined (i.e. share an endpoint) such that they have identical tangent lines at the join point.

A hyperedge connects N items (each item has an attachment point). The layout of the segments which make the hyperedge is a recursive process, starting with the set of connection points being the set of all attachment points. Each segment is a Bezier tube made of two patches. There is one segment in a hyperedge which connects two items, three segments to connect three items, five segments to connect four items. In general, there are 2(N - 2)+1 segments in a hyperedge which joins N items.

Determining Bezier space curves for a set of connection points.

First, find the two connection points closest together, and calculate the point which is the centroid of the remaining N-2 connection points. Calculate a "join" point which is the centroid of the two selected connection points and the N-2-points centroid. Calculate the centroid of the join point and the two selected connection points. This last centroid is the 'JoinInterior' point. For each selected connection point, if that connection point was a previous join point then use the associated JoinFollowing point as the ConnectionInterior point, otherwise calculate the average of the (current) join point and that connection point as the ConnectionInterior point. Create two bezier curves, joining each of the two points to the join point. The control points of the curve for a particular connection point are: the connection point, the ConnectionInterior point, the JoinInterior point, and the join point. The two curves have the same third and fourth control points. This ensures that the two curves will join tangentially at the join point. Finally, calculate a JoinFollowing point which is the JoinInterior point "mirrored" across the join point. This JoinFollowing point is used when defining the curve which starts at the join point. Add the join point (its JoinFollowing point as an annotation) to the set of points to be combined in the hyperedge. Recursively invoke the segment curve generating process on this new set.

This procedure can be described symbolically as follows: Let P be a set of connection points. Let CLOSEST(X) be a function which returns a pair set of points which are two points in the set of points X which are closes together. Let CENTROID(X) be a function which returns the point which is the centroid (average) of the points in X. Let CARDINALITY(X) be a function which returns the cardinality of the set X. Let THIRDS(X) be a function which returns a tuple of four points, <P1, T1, T2,

P2>, where P = {P1, P2} and T1 and T2 divide the line between P1 and P2 into thirds. Let JOINPLANESHIFT(V) be a function which returns V with its z component set to JOINPLANE (some value input to this procedure). Let FOLLOWINGMAP be a set of ordered pairs (an association list) with first element a point in P and the second element the "following" point for P. Not all points in P have associations in FOLLOWINGMAP. It is initially empty.

The function CURVES(JOINPLANE, X, FOLLOWINGMAP) returns a set of Bezier space curves (specified by four-tuples) given a list of points X and a FOLLOWINGMAP. The curves with join plane JOINPLANE for a set of connection points P is found by CURVES(JOINPLANE, P, {}).

If CARDINALITY(P) = 2, then do: If both points in P = {P1, P2} have following points in FOLLOWINGMAP (<P1, F1> and <P2, F2>) then the Bezier space curve is <P1, F1, F2, P2>. Else, if only one point has a following point (<P1, F1>), then let M = CENTROID({F1, P2}), and CURVES = {<P1, F1, M, P2>}. Else, neither point has a following point, so let <P1, T1, T2, P2> = THIRDS(P), J1 = JOINPLANESHIFT(T1), and J2 = JOINPLANESHIFT(T2). The Bezier space curve result is CURVES = {<P1, J1, J2, P2>}.

When CARDINALITY(P) > 2, then do: Let A = {A1, A2} = CLOSEST(P), P' = P - A, C = CENTROID(P'), and M = JOINPLANESHIFT(CENTROID(A ∪ {C})). The result of the rest of the procedure is to define a Bezier space curve that connects A1 and M and another one which connects A2 and M. Also, a "following point" is calculated for M and added to the FOLLOWINGMAP. To connect A1 to M: Either <A1, F1> is in FOLLOWINGMAP, or else let F1 = CENTROID({A1, M}). The curve from A1 to M is CURVE1 = <A1, F1, C, M>. Similarly, F2 is determined for A2 and M, and the curve for A2 to M is CURVE2 = <A2, F2, C, M>. The "following point" for M is F = (M + (M - C)). The resulting set is:

```
CURVES = {CURVE1, CURVE2}
      ∪ CURVES(JOINPLANE,
            P' ∪ {M},
            (FOLLOWINGMAP
                  - {<A1, F1>, <A2, F2>})
            ∪ {<M, F>}).
```

The new connection point set (P' ∪ {M}) has one fewer points in it than does P: P minus the two attachment points, plus the new join point M. Thus, this recursion will eventually halt. The JoinFollowing point (F) in the above construction ensures that the segment "leaving" from the join point will join tangentially with the segments "entering" the join point.

Calculating the control points of the Bezier patches for a Bezier tube.

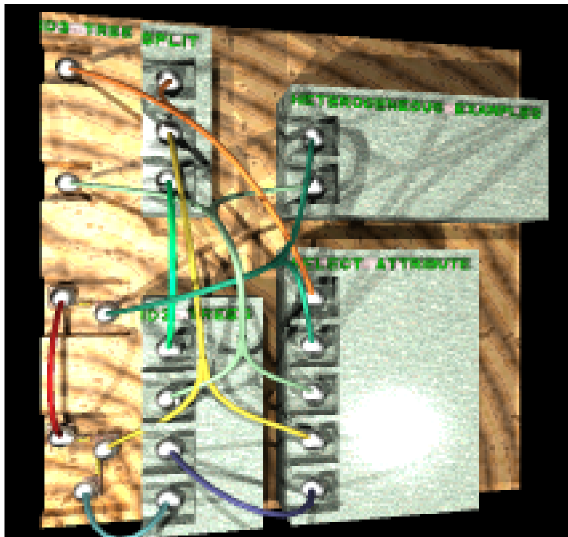
The Bezier tube for a given a cubic Bezier space curve has that curve as its (approximate) axis. The simplest way to model this is to "sweep" a circle along the curve, with the center of the circle following th curve and the plane of the circle always perpendicular to the tangent of the curve. The modeling system we used does not have the capability of sweeping a 2D figure along a Bezier space curve, so we approxiate this effect by specify two Bezier patches as described earlier, one for the "top" half of the tube and the other for the "bottom" half of the tube. The Bezier patch approach can be preferable to the sweep approach when both are available since the Bezier patches are usually faster to render than the sweep.

Figure 3 gives an indication of the setting of the patch control points. The "top" patch control points are placed at a fixed offset distance from the curve in certain directions. This fixed offset distance is approximately the radius of the Bezier tube. At the endpoints, the patch control point's offset direction is perpendicular to the curve. For an interior control point, the offset direction is halfway between the directions toward the two adjacent control points. Patch control points are placed at the offset distance in both the positive and negative offset directions. Also, an "up" offset direction is determined at each control point, and two more control points are placed, one each "up" from the first two offset control points. This gives four control points in the "top" patch at each control point of the original curve. Care must be taken in placing the patch's control points in the appropriate columns and rows of the patch control matrix to create a smooth "half tube" appearance. If these control points are placed in the wrong parts of the patch's control matrix the surface becomes twisted. The "bottom" patch's control points are generated from the "top" patch's control points. The exterior columns are the same in both patches. The two "inside" columns of the "bottom" patch are offset twice the negative of the "up" direction from the two inside columns of the "top" patch.

Related Work

There has been a limited use of 3D in visual programming languages. Cube [6] uses simple line drawings of cubes connected by lines positioned in 3D to represent higher-order logic programs. It was intended that Cube would eventually be a virtual reality interface allowing programmers to directly manipulate programs in its 3D space.

Lieberman [7] uses animated 3D blocks and text for LISP program visualization. 3D colored rectangular solids and solidly extruded text "rotate" to convey invocation



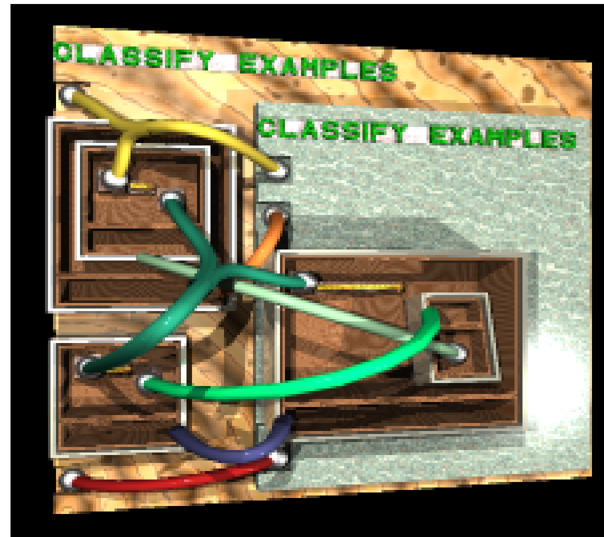


Figure 4: Two Alternative Representations.

and "stack" to convey a visual representation of the run-time call stack.

CAEL [8] is a PASCAL-like language that uses colored rectangular solids and solidly extruded text, as well as 3D “icons”, for naming. It extends flowcharting into 3 dimensions [9].

The SPARCL approach to 3D modeling uses textured solids of various kinds to represent the various kinds of elements of a SPARCL program. These include rectangular solids, hollow boxes, extruded text, spheres, arrows (made of a cylinder and a cone), and Bezier-patch tubes. These models are specified (internally) using Constructive Solid Geometry trees of “simple” solids (boxes, cylinders, cones, spheres, halfspaces) and Bezier patches.

A simply oriented scene is one where all of the elements have “natural” local coordinate systems, and these natural local coordinate systems of the elements of the scene are pairwise mutually parallel. A complexly oriented scene is one which is not simply oriented. All of the 3D approaches other than SPARCL use simply oriented scenes. SPARCL builds complexly oriented scenes: the tubes (which are used to connect other elements) may have no “natural” local coordinate system; also, certain elements are placed in a helix, which rotates their natural local coordinate systems into non-parallel relationships.

CUBE uses the simplest of rendering approaches - perspective line drawings with hidden line removal. 3D-CAEL, Lieberman and “dimensional flowcharts” use the simplest “solid” rendering: a shaded, anti-aliased rendering without shadows, no texturing, no reflections. SPARCL uses a ray-traced rendering which provides shadows, reflection, refraction, and complex texturing (simulating wood, stone, metal, glass, and plastic).

The environments of CUBE and 3D-CAEL are unified - the programmer works directly with the 3D representation. It is unclear how one creates Lieberman’s programs, but his approach should be amenable to a unified environment. SPARCL has a separated environment: the editing environment uses the 2D diagrammatic representation. The SPARCL generates program scene models and renders these as a separate procedure. The reason for the separated approach is that fully rendering the scenes is extremely time-consuming. It is possible to use much faster rendering techniques and specialized hardware so that a fully 3D environment could be implemented. However, the fully rendered version of a SPARCL program would still be too time-consuming to have it done in “real time” for interactive editing.

Future Work

As one can see from the figures, the hyperedges are just one part of an entire 3D program representation. Research in the overall representation is ongoing. Also, we are studying improvements in the approach to laying out the geometry of the hyperedges and alternatives for surface texturing. We plan to carry out user testing on

various 3D rendering parameters to determine which most facilitate understandability. Figure 4 shows other renderings under consideration. Stereoscopic viewing is another interesting possibility.

A major problem for this approach is the time-consuming nature of photo realistic rendering. Much of the benefit of Bezier tubes in terms of readability can be achieved using one of the less expensive approaches to rendering that provide just shading and color without highlights or shadows. This may allow one to use a rendering approach in real time on a graphics work station that supports real time animation.

Conclusion.

In this paper we have discussed 3D rendering as a solution for the line crossing problem associated with using connecting lines for relations in visual program representation. The Bezier tube was introduced for modeling connecting lines in general and hyperedges in specific. Other approaches for relating elements of a picture were also discussed.

The 3D hyperedges introduced in this paper were shown to have various features that enhance their readability over the readability of 2D representations of connecting lines.

References

[1] Ed Dengler, Mark Friedell, and Joe Marks, "Constraint-Driven Diagram Layout", Proceedings of the 1993 IEEE Symposium on Visual Languages, Bergen, Norway, 1993, pp. 330-335.

[2] C. Kosak, J. Marks, S. Shieber, "Automating the Layout of Network Diagrams with Specified Visual Organization" IEEE Trans. on Systems, Man and Cybernetics, 1993, (to appear).

[3] Alex S. Fukumaga, Takayuki D. Kimura, and Wolfgang Pree, "Object-Oriented Development of a Data Flow Visual Language System", Proceedings of the 1993 IEEE Symposium on Visual Languages, Bergen, Norway, 1993, pp. 134-141.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, Computer Graphics Principles and Practice, Addison-Wesley, Reading, Massachusetts, 1990.

[5] Lindsey Spratt and Allen L. Ambler, "A Visual Logic Programming Language Based on Sets and Partitioning Constraints", Proceedings of the 1993 IEEE Symposium on Visual Languages, Bergen, Norway, 1993, pp. 204-208.

[6] Marc A. Najork and Simon M. Kaplan, "The CUBE Language", Proceeding of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan, 1991, pp. 218-224.

[7] H. Lieberman, "A 3D Representation for Program Execution", Proceeding of the 1989 IEEE Workshop on Visual Languages, Rome, Italy, 1989, pp. 111-116.

[8] Frank Van Reeth and Eddy Flerackers, “Three-dimensional Graphical Programming in CAEL”, Proceedings of the 1993 IEEE Symposium on Visual Languages, Bergen, Norway, 1993, pp. 389-391.

[9] R. W. Witty, "Dimensional Flowcharting", Software - Practice and Experience, vol. 7, 1977, pp. 553-584.

[10] J. R. Quinlan, "Induction of Decision Trees", Machine Learning, vol. 1, no. 1, pp. 81-106.