HyperSnapping

Toshiyuki Masui
Sony Computer Science Laboratories, Inc.
3-14-13 Higashi-Gotanda, Shinagawa, Tokyo 141-0022, Japan.
masui@acm.org

Abstract

We introduce a new drawing technique called HyperSnapping. In many drawing editors, various operations are provided for object alignment, and snapping is one of the most frequently used techniques, with which users can snap the mouse cursor or dragged objects to existing objects or square grids. HyperSnapping is an extension to snapping operations, where users can control the behavior of snapping grids and the constraints between objects only by dragging objects, without explicitly changing drawing modes. With HyperSnapping, simple constraints and drawing macros can easily be constructed and used for further editing without explicit end-user programming.

1. Introduction

When drawing figures using drawing editors, various constraints can be used to make them look better. Aligned figures usually look better than randomly laid out figures. When drawing a rectangle at the center of a page, it looks better if the left and right margins are equal. Many drawing editors provide special commands for alignment and centering operations.

In many drawing editors, "snapping grids" can be used for object alignment, where users can drag objects and snap them to the grids, just like drawing figures on graph paper. Using grids, users can easily make multiple objects have the same X/Y coordinate, or lay out multiple objects with the same margins. Sophisticated editors provide "Snap-Dragging"[2], with which the mouse cursor automatically snaps to a position which is aligned to existing objects.

Functions for grids and Snap-Dragging are usually predefined in each drawing editor, and users cannot customize the behavior of the constraints. For users who want to use more complex constraints, various "constraint editors" have been developed. Using a constraint editor, users can define arbitrary constraints between drawing objects and make the editor solve the constraints automatically. For example, if a user defines a constraint which requires that the distance between rectangle A and rectangle B should always be the same as the distance between B and C, B is always at the center of A and C, no matter where he moves these objects.

Constraint editors have long been studied and they have great potential, but they are not widely used at this moment yet, partly because solving the constraints is not always an easy task, but mainly because most users do not want to take time to define and maintain constraints for simple drawing tasks. Users have to debug the constraints if the specified constraints are wrong or inappropriate, and the system could not solve them.

Gleicher's Briar[4] is a constraint editor where users can define constraints easily by using Snap-Dragging. Instead of defining constraints with extra operations, users can tell their intentions to the system only by performing Snap-Dragging operations. This technique has the nice feature that users cannot define unsolvable constraints. Briar can be viewed as a programming-by-example (PBE) system[3][6], since users can make a constraint-based program just by showing the system what they want to get, not by telling how to solve the problem.

Using Briar is much easier than using a constraint editor which requires explicit programming, but it is still too complicated for end users. Because Briar is based on Snap-Dragging, telling the users' intention can be ambiguous. For example, when a user places the mouse cursor at the intersection of line A and line B, the user's intention might be either 1) put the cursor at the intersection, 2) put the cursor on line A, 3) put the cursor on line B, or 4) put the cursor at that point. To tell his intention, he has to select one of them with extra operations.

We propose a simple drawing technique called *HyperSnapping*, with which users can lay out multiple objects using simple and powerful snapping operations, define constraints between objects only by the snapping operations, and define macros and constraints from operation histories.

2. HyperSnapping

HyperSnapping is a simple editing technique which enables simple constraint programming. HyperSnapping has the following features:

- Automatic snapping

  When a user starts dragging an object, it follows the cursor without snapping to anything. When the user moves the object farther than a certain distance, a small grid appears and the object begins snapping to other objects and grids. If the user moves the object furthermore, the size of the grid becomes larger, and the object snaps to larger grids. When the user finishes dragging, the grid disappears and snapping is disabled again.

- Automatic grid selection

  When a user drags an object and snaps it to other objects or the grid, the size and position of the grid does not change. If the user drags an object and does not snap it to anything else, the object becomes the basis for later snapping operations, and new grids are created based on the size and the position of the object.

- Defining constraints from successive snapping operations

  When a user drags an object and snaps it to the grid, the object becomes the "anchor object." A point on the object becomes the "anchor point", which is used as the pivot of various operations. When more than one vertex snap to the grid, one of them is selected by the cursor's moving direction. When the user drags another object and snaps it to the anchor object, a "subanchor point" is set on the object and the two objects are grouped. When the user drags yet another object and snaps it to the anchor object or the subanchor object, another subanchor point is defined on the new object and the three objects are grouped. A linear constraint is defined between the anchor point and subanchor points.

- Defining constraints from operation histories

  Just like the Dynamic Macro system[7] used in text editors, all the editing operations are recorded and used for defining macros. If a macro is defined from repetitive operations and executed afterwards, linear constraints are defined between the objects.

3. Examples

In this section, we show the features of HyperSnapping by showing examples.

Standard dragging operation

A user can drag an object by clicking the object and move the mouse. Figure 1 shows how a rectangle can be dragged freely using a mouse. (The mouse cursor is not shown in this figure.)

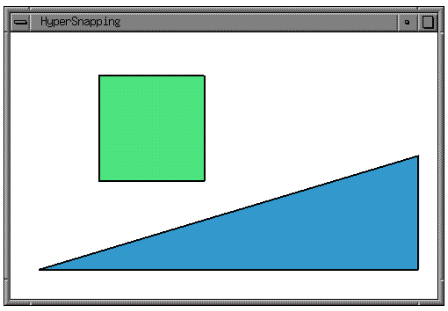

Figure 1. Dragging a rectangle.

Automatic control of grid size

When the user drags an object more than a certain distance, a snapping grid automatically appears and dragged objects begin to snap to the grid. A vertex snapped to the grid is called the anchor point. A small blue rectangle is displayed on the anchor point, and the object which has the anchor point is called the anchor object. The anchor point works as a pivot for various operations.

When more than one vertex snap to the grid, one of them is selected by the moving direction of the object. If the user moves the mouse to the right, rightmost vertex becomes the anchor point, even when other vertices are also snapped to the grid.

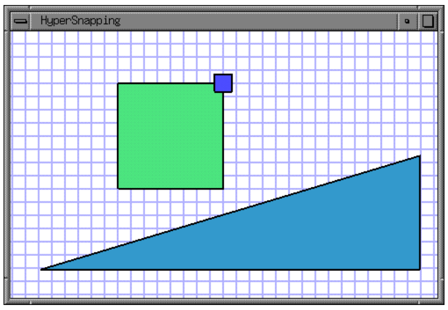

Figure 2. Snapping to fine grid.

If the user keeps dragging the object, the size of the grid becomes larger and larger, according to the amount of the dragging. (Figure 3)

The size of the grid changes automatically. A user can easily move an object to a distant place using a coarse-grain grid, and after that, he can adjust the position of the object without snapping, by starting the dragging operation again.

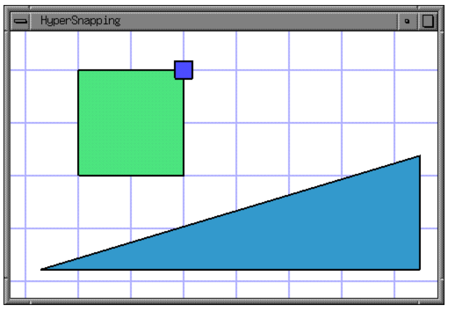

Figure 3. Snapping to coarse grid.

Changing the origin and the size of the grid

When a user moves an object without snapping, later snapping is performed based on that object. For example, if a user moves a 1cm^2 square, 1/8cm, 1/4cm, 1/2cm, and 1cm are selected as the grid sizes. If the triangle in Figure 3 is used as the base object, the grid size is calculated from the object, and other objects can easily be placed to have the same X or Y position as the triangle. (Figure 4)

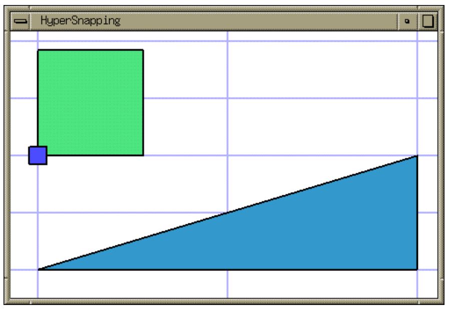

Figure 4. Using grids based on the base object.

Using anchor point for rotation and magnification

Dragged object can snap not only to grids, but also to vertices and edges of other objects.

Just like snapping to grids, a dragged object does not snap to other objects at the beginning of the dragging operation. After dragging the object more than certain amount, it begins to snap to other objects. Vertices of the dragged object snap to vertices and edges of other objects. (Figure 5)

If a user wants to place object A very close to B, he can first drag A and snap it to B, and then he can begin the dragging operation again and adjust the position of A.

If a user clicks a point close to a vertex, he can begin rotating the object, with the anchor point as the pivot. If a user clicks a point close to an edge, he can change the size of the object. When the user moves the mouse cursor without clicking the mouse, “shadow” shapes are displayed so that the user can tell what operation is possible at the position. Figure 6 shows the display when the user is trying to rotate the object by pointing at the vertex at the other side from the anchor point.

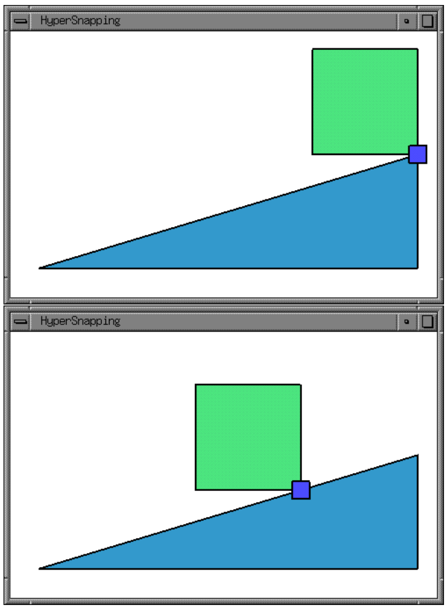

Figure 5. Snapping to vertices and edges.

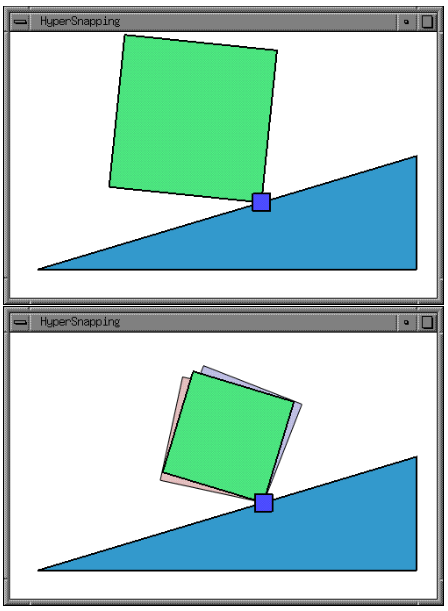

Figure 6. Rotation and magnification based on the anchor point.

Since rotated objects and enlarged objects snap to grids and other objects, the rotated rectangle in Figure 6 can easily snap to the edge of the triangle, as shown in Figure 7.

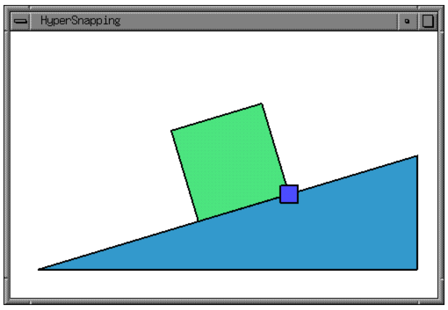

Figure 7. Rotating a rectangle and snapping it to an edge.

Group operations and automatic constraint generation

When an object is dragged and snapped to the anchor object, a constraint between the two objects is established, and the two objects will be treated as a group.

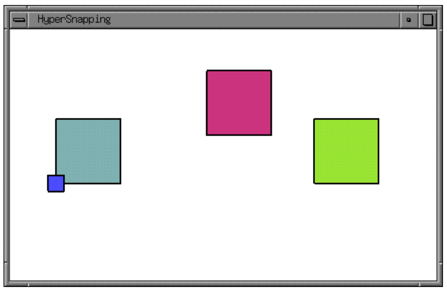

Figure 8. Three rectangles before snapping.

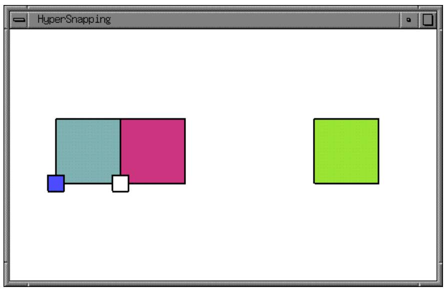

Figure 9. Moving the center rectangle and snapping it to the left rectangle.

In Figure 8, when the rectangle at the center is snapped to the left rectangle, the snapped vertex becomes the subanchor and a small white rectangle is displayed at that point. (Figure 9)

If the user moves the right rectangle and snaps it to the rectangle in the middle, another subanchor point is created as shown in Figure 10. Linear constraints are automatically established between the three objects, and if the user moves the center rectangle or the right rectangle, the position of other rectangles also changes accordingly, just as shown in Rectangle 11.

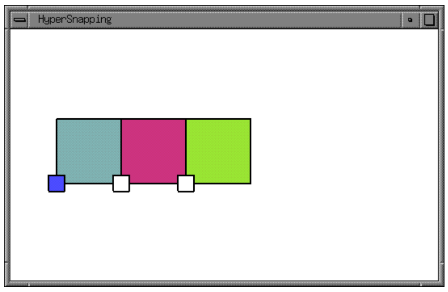

Figure 10. Snapping the right rectangle to the middle rectangle.

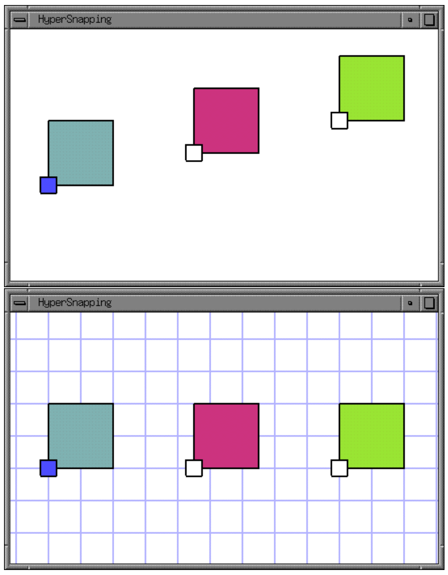

Figure 11. Dragging the right rectangle.

If the user moves the anchor object, all the subanchor objects also move accordingly. (Figure 12.)

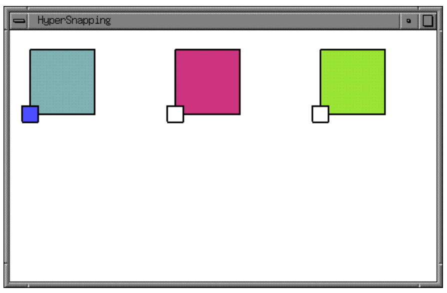

Figure 12. Dragging the anchor object.

If the user rotates the subanchor object, all the anchor and subanchor objects rotate around the anchor point. (Figure 13)

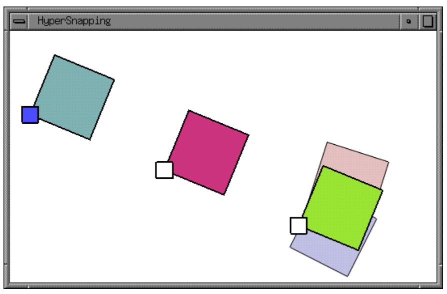

Figure 13. Rotating a subanchor object.

If the anchor object is rotated or enlarged, all the subanchor objects are rotated or enlarged by the same amount.

When the user clicks at a point with no object, all the anchor, subanchor, and constraints are cleared.

Creating constraints from repetitive operations.

Figure 14 shows the display after a user copies a rectangle, pastes it, and snaps it to the original rectangle. If he performs the same operation again, the display will change to Figure 15.

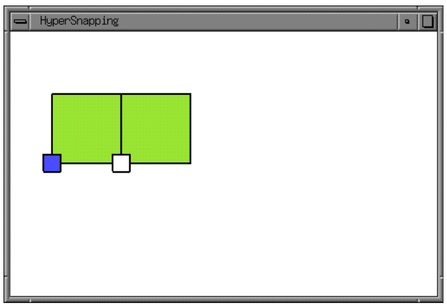

Figure 14. Duplicating a rectangle and snapping it to the original rectangle.

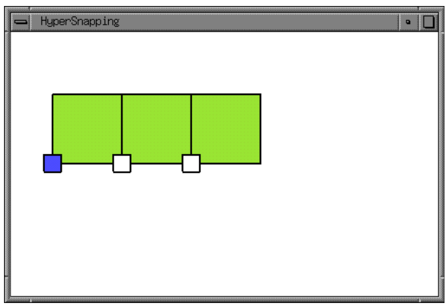

Figure 15. After performing the same operation again.

If a user performs the same operation more than once, the next operation can be predicted from the history of operations, using the prediction method used in the Dynamic Macro system[7]. If the user asks the system to perform the operation again, the repetitive operation is automatically retrieved from the operation history and executed, resulting in Figure 16.

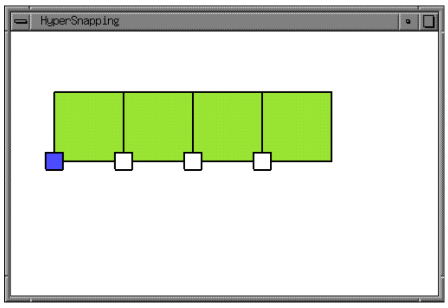

Figure 16. Performing repetitive operations automatically.

Constraints are kept between all the objects, and moving any of the subanchor objects results in Figure 17.

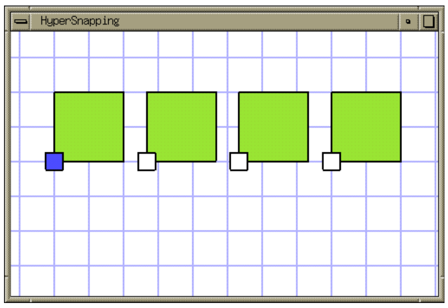

Figure 17. Using the constraints defined by repetitive operations.

Other drawing examples

Using the features of HyperSnapping, diagrams like Figure 18 can very easily be drawn.

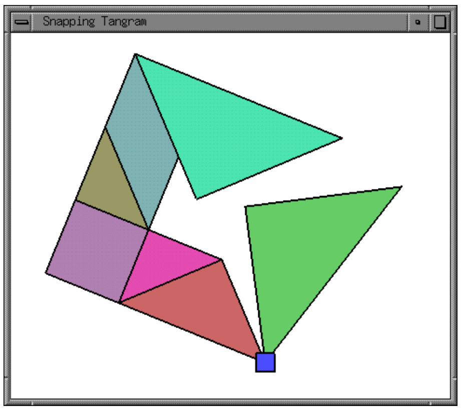

Figure 18. Tangram.

A user is trying to place four aligned rectangles at the edge of a triangle in Figure 19. It is very difficult to draw this kind of figure with a drawing editor which does not support snapping and constraints definition. Using HyperSnapping, users can very easily draw this kind of figures without invoking special commands or changing modes.

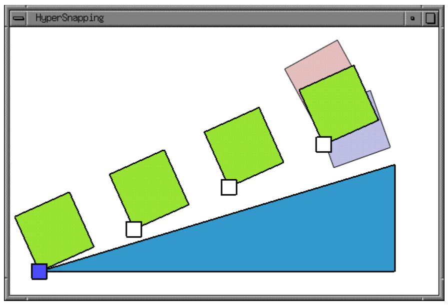

Figure 19. Drawing aligned rectangles at the edge of a triangle.

4. Evaluations

The advantages of using HyperSnapping are as follows:

- Users can control the level of snapping only by dragging objects.
- Users can define a group of objects and constraints between objects without using extra operations.

With these features, users can very easily draw aligned figures like Figure 19.

Also, HyperSnapping has the following advantage.

- Simple PBE methods can be applied based on the operation history.

It is usually difficult to apply PBE techniques on graphical editors, since in many cases user's intention is unclear to the system and the system needs to infer the real intention of the user's operation. For example, when a user moves an object to the left of the screen, the system cannot tell whether 1) he wanted to move the object to the left of the screen, 2) he wanted to move the object closer to another object, or 3) he wanted to move the object by certain amount. In this case, the system cannot create any program only from this single example. With HyperSnapping, the user's intention is always clear to the system if the user sets the anchor and subanchor points properly. Anchor and subanchor points can be specified only by small mouse movement, and users can easily tell their intentions to the system and take advantage of the constraints using the anchor and subanchor points.

On the other hand, HyperSnapping has the following shortcomings:

- Users cannot modify the predefined constraints defined to each snapping operations.
- The group and constraints defined by the snapping operation is temporary, and cannot be used later.

5. Related Works

Simple snapping features are available on most of the drawing editors. More sophisticated snapping features like Snap-Dragging[2] are now becoming popular and adopted in several drawing editors[1].

Honda proposed simple and powerful drawing operations called *Integrated Manipulation*[5]. With Integrated Manipulation, users can move/enlarge/rotate an object just by dragging the object and moving it closer to another object. When the dragged object gets closer to another object, a point in the object (called the *pivot*) snaps to the target object, and the dragged object automatically rotates and changes size around the pivot. HyperSnapping does not support moving and rotating at the same time, but similar operation is possible by snapping the anchor point and rotating around it.

The history of constraint-based drawing editors is very long. SketchPad[10], Juno[9], and many other systems have been developed and various algorithms for solving constraints have been proposed. However, none of them has become popular and widely used, maybe because of the difficulty of programming the constraints.

As we mentioned in section 1, the basic idea of Briar[4] is very close to HyperSnapping in that snapping operations are

[1]Snap-Dragging is available on Adobe Illustrator 8.0.

used for defining constraints. In HyperSnapping, the amount of snapping is automatically controlled by the user's dragging operation, and although the programmable constraints are very limited, almost no extra operations are required to set up constraints, while the user must give extra information to tell Briar which constraints should be used. HyperSnapping can also use the operation history for constructing constraints between objects.

Badros' SCWM[1] is a window manager which enables users to define constraints among the windows on the display, using tool buttons and the user's voice. Using SCWM, users can define sophisticate constraints either by using buttons or by writing Scheme scripts. Snapping operations are not used in SCWM, but adding the automatic snapping mechanism of HyperSnapping would be useful for easier manipulation of windows.

To help users automate graphical editing tasks, many PBE systems for graphical drawing have been proposed. MetaMouse[8] is a system which infers the production rules implicit in the user's editing operations. Since it is difficult to get the user's intention only from the user's editing operations, MetaMouse frequently asks the user questions about why the user performed the operation, so that the system does not make wrong guesses and create useless programs. In HyperSnapping, since the constraints are very limited and the user's intentions are clearly specified by using anchor/subanchor points, questions to users are not necessary at all.

In addition to the systems described above, a number of snapping editors, constraint-based drawing systems, and PBE-based drawing systems have been proposed and used. Although HyperSnapping only offers limited features compared to other sophisticated systems, it is a combination of the essences of these approaches, and any end-user can define constraints easily and use them effectively for his drawing tasks.

6. Conclusions

We have developed a simple and powerful drawing technique called HyperSnapping. To be really successful, we believe that end-user programming systems should be useful and easy to use at the same time. Although being very simple, HyperSnapping users can easily define constraints between objects with simple operations, and use the constraints effectively for drawing tasks.

References

[1] G. J. Badros, J. Nichols, and A. Borning. Scwm: An intelligent constraint-enabled window manager. In *AAAI Spring Symposium on Smart Graphics*, March 2000.

[2] E. A. Bier. Snap-dragging. *Computer Graphics*, 20(4):233–240, August 1986.

[3] A. Cypher, editor. *Watch What I Do – Programming by Demonstration*. The MIT Press, Cambridge, MA 02142, 1993.

[4] M. Gleicher and A. Witkin. Drawing with constraints. *The Visual Computer*, 11(1):39–51, 1994.

[5] M. Honda, T. Igarashi, H. Tanaka, and S. Sakai. Integrated manipulation: Context-aware manipulation of 2d diagrams. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'99)*, pages 159–160. ACM Press, November 1999.

[6] H. Lieberman, editor. *Your Wish is My Command – Programming by Example*. Morgan Kaufmann Publishers, 2001.

[7] T. Masui and K. Nakayama. Repeat and predict – two keys to efficient text editing. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, pages 118–123. Addison-Wesley, April 1994.

[8] D. L. Maulsby, I. H. Witten, and K. A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proceedings of SIGGRAPH'89*, volume 23, pages 127–136, Boston, MA, July 1989.

[9] G. Nelson. Juno, a constraint-based graphics system. *Computer Graphics*, 19(3):235–243, July 1985.

[10] I. Sutherland. Sketchpad: A man-machine graphical communication system. *IFIPS Proceedings of the Spring Joint Computer Conference*, January 1963.