Dannenberg and Amon, "A Gesture Based User Interface Prototyping System," in Proceedings of the 2nd annual ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, WV, November 13-15 1989. New York: ACM, 1989. pp. 127-132.

A Gesture Based User Interface Prototyping System

Roger B. Dannenberg and Dale Amon

School of Computer Science
Carnegie Mellon University
email: Roger.Dannenberg@cs.cmu.edu

Abstract

GID, for Gestural Interface Designer, is an experimental system for prototyping gesture-based user interfaces. GID structures an interface as a collection of "controls": objects that maintain an image on the display and respond to input from pointing and gesture-sensing devices. GID includes an editor for arranging controls on the screen and saving screen layouts to a file. Once an interface is created, GID provides mechanisms for routing input to the appropriate destination objects even when input arrives in parallel from several devices. GID also provides low level feature extraction and gesture representation primitives to assist in parsing gestures.

1. Introduction

Gestures, which can be defined as stylized motions that convey meaning, are used every day in a variety of tasks ranging from expressing our emotions to adjusting volume controls. Gestures are a promising approach to human-computer interaction because they often allow several parameters to be controlled simultaneously in an intuitive fashion. Gestures also combine the specification of operators, operands, and qualifiers into a single motion. For example, a single gesture might indicate "grab this **assembly** and **move** it to **here, rotating** it **this much**." Previous work on gesture based systems [1, 2, 6, 4, 12] has only begun to explore the potential of gestural input. We need a better understanding of how to construct gestural interfaces, and we need systems that allow us to prototype them rapidly in order to learn how to take advantage of gestures. Our work is a step toward these goals.

Building interactive systems based on gesture recognition is not a simple task. As we designed and implemented our system, we encountered several problems which do not arise in more conventional mouse-based systems. One problem is supporting multiple input devices, each of which might have many degrees of freedom. Unlike most mouse-based systems which can only engage in one interaction at a time, our system supports, for example, turning a knob and flipping a switch simultaneously.

Another problem is how to parse input into recognized gestures. We assume that gestures are specific to various interactive objects. For example, a switch displays an image of a toggle on the screen and can be "flipped" by a fingertip, but only if the finger travels across the image in the right direction. In this case, finger motion must be interpreted in the context of the interactive object, and a path (as opposed to instantaneous positions) defines the gesture.

Beyond these problems, we were also interested in making our prototyping environment easy to use, modular and extensible. Thus, we have been concerned with the issues of how to combine interactive objects in a screen-based interface, how to edit the layout and appearance of the interface, and how to encapsulate the behaviors of interactive objects and isolate them from other aspects of the system.

A final issue is the question of debugging support to aid in the implementation of new interactive objects. We use input logging to make bugs more reproducible and a combination of interpreted and compiled code to speed development.

We have completed a system, named GID for Gestural Interface Designer, in which one can interactively create and position instances of interactive objects such as menus, knobs and switches. One can interactively attach semantic actions to these objects. GID supports input from both a mouse and a free-hand sensor that can track multiple fingers. We are far from having the ultimate gesture based interface support environment, but we have developed interesting new techniques that are applicable to future gesture-based systems.

In section 2 we describe the structure of our prototyping system, and section 3 describes the handling of input from multiple devices. In section 4 we describe our general technique for processing input in order to recognize gestures. Section 5 describes in greater detail our development techniques and the current implementation. Conclusions are presented in section 6 along with suggestions for future work.

2. The Interface Designer

This project extends an earlier effort called Interface Designer, or ID. The goal of ID was to provide a small, practical and portable system for creating screen-based interfaces by direct manipulation. ID was inspired by Jean-Marie Hullot's work at INRIA, a precursor to Interface Builder [5, 9]. A typical use of ID might be the following: by selecting a menu item, the user creates an instance of an object which displays a 3-D database. In order to manipulate the image, the user creates a few instances of sliders. A short Lisp expression is typed to supply an action for each slider, and labels of "azimuth", "altitude" and "pitch" are entered. Now, moving a slider causes a message to be sent to the display object and the image is updated accordingly.

The basic internal structure of ID introduces no significant improvements over other object-oriented event-driven interface systems such as MacApp [11] or Cardelli's user interface system [3]. It will be described here, however, for clarity.

ID represents the screen as a tree of objects. At the root is a screen object that contains a set of window objects. Each window object may contain a set of control objects. One type of control object is the control group, which serves to collect a set of control objects into an aggregate. Other types of control objects include sliders, buttons and switches of various styles. (See figure 2-1.)

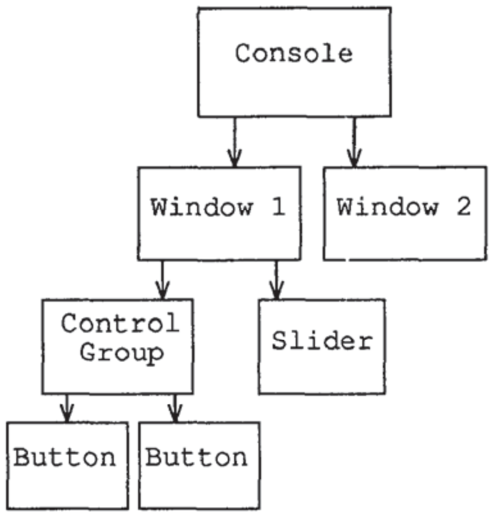

Figure 2-1: An ID control object tree.

In addition to the hierarchy implied by this tree, there is also a class hierarchy arranged so that classes can inherit much of their behavior. (See figure 2-2.) The InputControl class encapsulates generic behavior of objects that handle input from the user and manage some sort of image on the screen. PictureControls, a subclass of InputControls, actually draw images. These include classes such as Switch and Slider. Another subclass of InputControl is ControlGroup, which implements the search for an input handler. New interactive controls are typically created by subclassing PictureControl or one of its subclasses. Output-only "controls" have also been defined as subclasses of Control. For example, class 3dPict draws a wire-frame rendering of a 3-D data base which is loaded from a file.

```
Object
    Control
        InputControl
            PictureControl
                Button
                Switch
                Slider
                FontDev
            ControlGroup
                Console
                Window
                Menu
                MenuCard
                PictureGroup
            MenuItem
        OutputOnlyControl
```

Figure 2-2: Interface Designer class hierarchy.

In normal operation, ID has a single main loop that waits for input and delivers it to the appropriate destination. Each input event is represented by a window identifier, a device type (e.g. mouse or keyboard), coordinates (if any), and other data. This event is passed to the root of the tree where a search for a recipient begins. Typically, each node which is not a leaf node (a PictureControl) passes the event to each of its children until one of them accepts the input event.

To make this recursive search reasonably efficient, a ControlGroup object rejects mouse input which falls outside of its bounding box, and windows reject input unless the event's window identifier matches. Even with these optimizations, it is too inefficient to search the object tree from the root for each mouse-moved event during a dragging operation. Instead, a context mechanism is used.

In ID, the handler for input is found at the top of a *context stack*. An object can grab future input events by pushing a new context onto the stack to direct future input to the object. For example, a dragging operation would start with a mouse-down event that would be handled in the normal way. Upon receiving the mouse-down event, the object that handles the dragging operation pushes the context stack and becomes the target of future input. All successive mouse-move events go directly to the object. When mouse-up is received, the object pops the current context to restore input processing to normal.

The context stack has two uses in addition to temporarily grabbing mouse input. The context stack is used for nested pop-up windows and also for implementing an ''edit'' mode in which control objects can be created, moved, copied, and deleted. In edit mode, we want to be able to select controls without invoking their normal operations. This is accomplished by pushing a special ''edit context'' which routes all input to an editor that can manipulate the on-screen objects.

3. Parallel Input Handling

We used ID as the basis for GID, our gesture-based system. GID was designed to be used with a Sensor Frame [7] as the gesture sensing device. The Sensor Frame tracks multiple objects (normally fingers) in a plane positioned just above the face of a CRT display. The ''plane'' actually has some thickness, so three coordinates are used to locate each visible finger. When a finger enters the field of view, it is assigned a unique identifier called the *finger identifier* Each time the finger moves, the new coordinates of the finger and the finger identifier are transmitted from the Sensor Frame to the host computer. Ideally, when a finger enters the field of view of the Sensor Frame, it is assigned a number which it retains for the entire time it remains in view. Since the Sensor Frame may be tracking multiple fingers in parallel, coordinate changes for several fingers may be interleaved in time.

In our gesture-based system, we wanted be able to handle multiple finger gestures acting on a single object, for example, turning a knob. We also wanted to allow users to operate a control with each hand. The stack-based context mechanism described in the previous section, however, does not allow inputs to be directed to several objects. We could simply pass all input to the root of the object tree, but again, the search overhead would be too high.

Our solution is to maintain a more general mapping from input events to objects. Each context contains a list of input templates, each of which has an associated handling object. Input templates consist of a window identifier, device type, and finger identifier. If all elements of the template match corresponding elements of an input event (the template may have ''don't care'' values) then the event is sent to the indicated handling object. If no template matches, then input is sent to a default handling object, also specified in the current context. As a result, we can have:

- two fingers operating a knob (input from either finger is forwarded immediately to the knob object),
- another finger moving toward a switch (input from this finger goes to the root of the object tree as usual. The switch object may change the current context and take future input directly when the finger gets close), and
- a simultaneous mouse click on a button (this input would work its way through the object tree from the root to the button object).

In some cases, one might want to effect a global context change, such as a pop-up dialog box which preempts all controls. This is accomplished by pushing a new context on the stack. This may redirect input from an object with a gesture in progress. We avoid problems here by sending a ''finger up'' event to the old handling object and a ''finger down'' event to the new handling object whenever a finger changes windows.

4. Gesture Representation and Processing

Since individual finger coordinates do not convey any dynamic aspects of gestures, the first stage of processing Sensor Frame input data is to represent the path of each finger by a set of features. The features are then interpreted by controls. The current set of features includes a piece-wise linear approximation of the path, the point where the path first crosses into an ''activation radius'', and the cumulative angular change.

4.1. Initial Processing

The x,y,z coordinates are supplied by the Sensor Frame as integers but are translated to floating point for further processing. The x,y,z portion of the input data is referred to hereafter as a *Raw Data Point* or *RDP*.

Normally, the default handling object for RDP's is the root of the object tree. The tree is searched after each input; however, when the RDP falls within the bounding box of a control object, the object responds by putting a template in the current context that will direct future events with the same finger identifier to the object. Future matching events will arrive at the object where they are added to a table associated with both the object and the finger identifier. This table of RDP's is called an *open vector*.

4.2. Path Decomposition

The next step is to process the open vector of RDP's to obtain a segmented[1] representation. This representation simultaneously provides data reduction and immunity from jitter.

For convenience, we want our approximation to be continuous; that is, each segment begins where the previous one ended, and all endpoints coincide with data points (RDP's). The algorithm for constructing the approximation is straightforward: as each RDP is added to the open vector, and error measure is computed. When the error measure exceeds a constant threshold, a segment from the first to the next-to-last point is added to the path and the open vector is adjusted to contain the last two RDP's. This algorithm can be described as ''greedy without backtracking'' since we pack as many RDP's into each segment as possible (limited by the error threshold) and we

[1]In this discussion, a *segment* is an ordered pair of points, e.g. RDP's, and a point is an x, y, z triple.

never try alternative assignments of RDP's to segments.

Figure 4-1 illustrates the process. The segment from point 1 to point 3 falls below the error threshold, but a segment from point 1 to point 4 exceeds the threshold. Therefore, the segment [point 1, point 3] is added to the path, and a new open vector [point 3, point 4] is started. This is extended to point 5 and then to point 6.

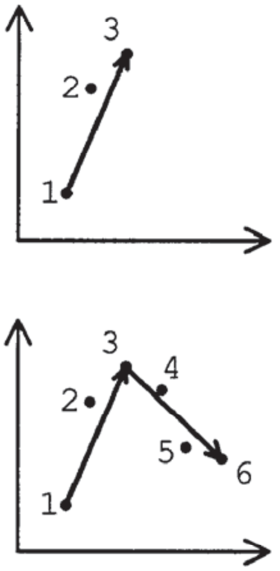

Figure 4-1: Fitting vectors to a set of points.

The error measure is:

$$error = \frac{1}{n}\sqrt{(\sum_{i=1}^{n} |D_x(p_i)|)^2 + (\sum_{i=1}^{n} |D_y(p_i)|)^2 + (\sum_{i=1}^{n} |D_z(p_i)|)^2}$$

where $D_x(p_i)$ is the x-component of the shortest vector from an RDP p_i to the proposed segment $[p_1, p_n]$ from point p_1 to p_n. We elected not to take a sum-of-squares in the innermost summation to save a bit of computation, and the resulting path decomposition seems to work well. The distance from a point to a line can be computed without trigonometric or square root functions as shown in Appendix I.

4.3. The Activation Volume

Gesture analysis is performed if an open vector passes into the volume defined by an activation radius and an activation center. Such processing will continue so long as succeeding RDP's remain within that volume.

An activation center is not necessarily static. For example, the knob on a slider has an activation center that moves along with it. The value associated with the device class is in this case a default initial value for the slider location.

Because we are polling the Sensor Frame from the application program, we cannot guarantee that we will catch all (or any) relevant RDP's within a possibly small *activation volume*. This is particularly true if the finger is traveling quickly. However, by setting the size of the bounding box large enough, we can guarantee we will at least pick up endpoints of a path segment that intersects this volume. The same distance algorithm (see Appendix I) used for path decomposition is then used to see if the point of closest approach of the path to the *activation center* is less than the *activation radius*.

4.4. Gestures

Once an RDP falls within the activation radius, the gesture features are examined by the corresponding object. Response to gestures is programmed procedurally for each type of control.

A toggle switch (or any other control affected by a simple linear motion), can be moved if the direction of travel of a finger path (**A**) matches the preferred axis of travel of the device (**B**). We define a maximum angle (θ_{max}) between the two and see if the actual angle (θ_{act}) is within bounds.

The actual angular error can be found using the definition of the vector dot product:

$$\mathbf{A}\cdot\mathbf{B} = |A||B|\cos(\theta_{act})$$

and rearranging to solve for $\cos(\theta_{act})$:

$$\cos(\theta_{act}) = (\mathbf{A}\cdot\mathbf{B}) / (|A||B|)$$

If the inequality:

$$\cos(\theta_{act}) \le \cos(\theta_{max})$$

holds, then the movement of the finger is close enough to the preferred direction to cause a state change. Note that $\cos(\theta_{max})$ is a constant that can be precalculated, thus we avoid calculating transcendentals at run time by comparing cosines of angles instead of the angles themselves and by using the equation:

$$\mathbf{A}\cdot\mathbf{B} = A_xB_x + A_yB_y + A_zB_z$$

The knob rotation gesture consists of one or two fingers moving within the activation radius of the knob. Once it is determined that a finger path crosses the activation radius, an angle from the center of the knob to the finger is computed and saved. Each location change within the activation radius results in a recalculation of the angle, and the angle of the knob is updated by the angular difference. When there are two fingers within the activation radius, the knob is updated when either finger moves; the overall knob rotation is effectively the average rotation of the two fingers.

5. System Considerations

5.1. Implementation Languages

Our Sensor Frame interface, gesture recognition software, and graphics primitives are all implemented in the C programming language. Graphical and interactive objects, as well as the top-level input handling routines, are implemented in XLISP, a lisp interpreter with built-in support for objects.

Although we would have preferred a compiled lisp, this work was begun at a time when our workstation environment was in a state of rapid change. During the course of the project, we ported XLISP to three machine types and implemented our graphics interface on two window managers. The fact that XLISP is a relatively small C program made it easy to port and to extend with the additional graphics and I/O primitives we needed.

5.2. Input Diagnostics

For diagnostic purposes, input of raw position data points is done through a device-independent module that allows input to come from a Sensor Frame, to be partially simulated by a mouse, or to be played back from a file that was "recorded" on a previous run with a mouse or a Sensor Frame. Bugs that appear only in long runs can be reproduced by playing back the log file during a debugging session.

The interface is implemented in such a way that regardless of which device is being used as the pointing device, the window menu is still available via the mouse. Commands are available to display every RDP as a small box on the screen; to print the results of every Sensor Frame input to a diagnostic window; to select a prerecorded file, a mouse or the Sensor Frame as the source of input; or to begin or end recording data for future playback.

6. Results and Conclusions

In the process of building GID, we have encountered several problems which are worth further study. One problem is how to organize prototyping software such as GID to allow controls to be operated in "run" mode and edited in "edit" mode. It seems inappropriate to implement editing within each object (Should a slider contain code for editing its size, placement, label, etc?), but a modular approach is preferable to a monolithic editor that captures all input in edit mode. In GID, we divert input when in "edit" mode, but we have specific editing methods in various subclasses of Control. One alternative is to implement all interactive behavior outside of control objects as in Garnet [8].

Another problem is that we have no high-level procedures for recognizing complex gestures: our recognizers must be hand-coded using fairly low-level representations. A promising alternative is the pattern recognition approach being pursued by Dean Rubine [10].

We know of no window managers that support multiple cursors. Ideally, the window manager should track each finger with a cursor and also determine what window contains each visible finger. Currently, the overhead of cursor tracking and mapping input to windows from outside of the window manager (X11) causes significant performance problems.

The present resolution of the Sensor Frame is only about 160 x 200 points. While this provides plenty of resolution relative to the size of controls displayed on the screen, greater resolution is needed in order to accurately measure the direction of motion and to minimize jitter.

The organization of GID prevents a single gesture from being received by multiple controls simultaneously. We do not feel this is a serious limitation, but it could be avoided by utilizing a more complete mapping from RDP's to objects. Rather than searching the object tree depth first, we could use hashing or a linear search of all objects to locate potentially overlapping bounding boxes which contain each RDP. Input events would then be duplicated and sent to each "interested" object. This technique was tried in an earlier system and allowed, for example, two adjacent switches to be flipped by moving a finger between them.

We note that some window managers might assist in the implementation of controls: if each control is implemented as a sub-window, then the search for a handler could be performed by the window manager. This technique will *not* work if we want input to reach multiple controls because current window managers will map input to only one window even if there is overlap. Furthermore, window managers typically assume a single pointing device, and extensive modification would be required to handle input from the Sensor Frame or some other gesture sensing device.

In conclusion, we have implemented a system for prototyping gesture-based user interfaces. The system is capable of editing its own interface, and applications are typically built by extension. The system allows us to experiment with screen layout and with multiple input devices without programming, and the system is extensible so that new interaction techniques can be integrated and evaluated. We have found piecewise linear approximations to paths to be an appropriate representation for simple gestures, and our vector software can be reused by different control objects.

7. Acknowledgments

We would like to thank Paul McAvinney, who played a crucial role in this research as inventor of the Sensor Frame and promoter of the value of gestures. Portions of this work were supported by the Sensor Frame Corporation as part of a project sponsored by NASA. Workstations used in this study were made available through an equipment grant from IBM. Thanks are also due to to Gerald Agin for the time he spent discussing curve fitting algorithms and analytic geometry.

References

1. R. A. Bolt. *The Human Interface: where people and computers meet.* Lifetime Learning Publications, 1984.

2. Frederick P. Brooks, Jr. Grasping Reality Through Illusion - Interactive Graphics Serving Science. CHI '88 Proceedings, May, 1988, pp. 1-11.

3. Luca Cardelli. Building User Interfaces by Direct Manipulation. Tech. Rept. 22, Digital Equipment Corporation Systems Research Center Research Report, Oct., 1987.

4. S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett. Virtual Environment Display System. ACM Workshop on Interactive 3D Grahpics, Association for Computing Machinery, 1986, pp. 77-87.

5. Jean-Marie Hullot. *Interface Builder.* Santa Barbara, CA, 1987.

6. Myron W. Krueger. *Artificial Reality.* Addison-Wesley, Reading, MA, 1983.

7. Paul McAvinney. U.S. Patent No. 4,746,770; Method and Apparatus for Isolating and Manipulating Graphic Objects On Computer Video Monitor. May 24, 1988.

8. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, SIGCHI'89, Austin, TX, April, 1989, pp. (to appear).

9. NeXT, Inc. *Interface Builder.* Palo Alto, CA, 1988. (online, preliminary documentation).

10. Dean Rubine. The Automatic Recognition of Gestures. (thesis proposal, Carnegie Mellon University School of Computer Science).

11. Kurt J. Schmucker. *Object-oriented programming for the Macintosh.* Hayden Book Co., Hasbrouck Heights, N.J., 1986.

12. David Weimer and S. K. Ganapathy. A Synthetic Visual Environment With Hand Gesturing and Voice Input. CHI'89 Conference Proceedings, Association for Computing Machinery's Special Interest Group on Computer Human Interaction, 1989, pp. 235-240.

I. Minimum Distance Between a Point and Segment

```
                X
                |
                |
A-----------P------------------B
```

Parameterize equation of **AB**:

$$\mathbf{V}(k) = (1-k)\mathbf{A} + k\mathbf{B}$$

for $0 \le k \le 1$, and $\mathbf{A} \le \mathbf{V} \le \mathbf{B}$ so that **V** is any point on the segment between **A** and **B**.

Release the constraint on k for the time being, and let *P* be the point nearest **X** on **AB**: $\mathbf{P} = \mathbf{V}(k_p)$.

This gives us Equation 1:

$$\textit{Eqn 1} \quad \mathbf{P} = (1-k_p)\mathbf{A} + k_p\mathbf{B}$$

or, in expanded form:

$$\mathbf{P} = \mathbf{A} - k_p\mathbf{A} + k_p\mathbf{B}$$

We want a line normal to **AB** that passes through **X**. By definition the dot product is zero if $\angle\mathbf{APX} = 90^o$, so for $\mathbf{XP} \perp \mathbf{AP}$ we have:

$$(\mathbf{P}-\mathbf{X})\cdot(\mathbf{P}-\mathbf{A}) = 0$$

Now substitute for **P**:

$$(\mathbf{A} - k_p\mathbf{A} + k_p\mathbf{B} - \mathbf{X})\cdot(-k_p\mathbf{A} + k_p\mathbf{B}) = 0$$

expand terms:

$$(-k_p+k_p^2)(\mathbf{A}\cdot\mathbf{A}) + (-k_p^2+k_p-k_p^2)(\mathbf{A}\cdot\mathbf{B}) + (k_p^2)(\mathbf{B}\cdot\mathbf{B}) + k_p(\mathbf{A}\cdot\mathbf{X}) - k_p(\mathbf{B}\cdot\mathbf{X}) = 0$$

divide through by k_p and simplify:

$$(-1+k_p)(\mathbf{A}\cdot\mathbf{A}) + (-2k_p+1)(\mathbf{A}\cdot\mathbf{B}) + k_p(\mathbf{B}\cdot\mathbf{B}) + (\mathbf{A}\cdot\mathbf{X}) - (\mathbf{B}\cdot\mathbf{X}) = 0$$

arrange terms for easier reduction:

$$-(1-k_p)(\mathbf{A}\cdot\mathbf{A}) + [(-k_p+1)(\mathbf{A}\cdot\mathbf{B}) - k_p(\mathbf{A}\cdot\mathbf{B})] + k_p(\mathbf{B}\cdot\mathbf{B}) + (\mathbf{A}\cdot\mathbf{X}) - (\mathbf{B}\cdot\mathbf{X}) = 0$$

apply distributive property of dot product:

$$(1-k_p)[\mathbf{A}\cdot(\mathbf{B}-\mathbf{A})] + k_p[(\mathbf{B}-\mathbf{A})\cdot\mathbf{B}] = [\mathbf{X}\cdot(\mathbf{B}-\mathbf{A})]$$

collect terms:

$$k_p[[-\mathbf{A}\cdot(\mathbf{B}-\mathbf{A})] + [(\mathbf{B}-\mathbf{A})\cdot\mathbf{B}]] + [\mathbf{A}\cdot(\mathbf{B}-\mathbf{A})] = [\mathbf{X}\cdot(\mathbf{B}-\mathbf{A})]$$

apply distributive property of dot product again:

$$k_p[(\mathbf{B}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})] = [(\mathbf{X}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})]$$

solve for k_p:

$$\textit{Eqn 2} \quad k_p = \frac{(\mathbf{B}-\mathbf{A})\cdot(\mathbf{X}-\mathbf{A})}{(\mathbf{B}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})}$$

Note that if $k_p < 0$, the nearest point to **X** is **A**. If $k_p > 1$, it is **B**. Otherwise solve Eqn 1 with value of k_p from Eqn 2 to get the nearest point.