Motorola Mobility, Inc. v. Microsoft Corporation Doc. 127 Att. 8

EXHIBIT 10

Dockets.Justia.com

11-14-00 A/PROV

Practitioner's Docket No. 03797.00072 PATENT

Preliminary Classification

Proposed Class:
Subclass:

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Keely, Leroy B.

For: Simulating Gestures of a Mouse Using a Stylus and Providing Feedback Thereto

Box Provisional Patent Application
Assistant Commissioner for Patents
Washington, D.C. 20231

COVER SHEET FOR FILING PROVISIONAL APPLICATION (37 C.F.R. SECTION 1.51(c)(1))

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 C.F.R. section 1.51(c)(1)(i). The following comprises the information required by 37 C.F.R. Section 1.51(c)(1):

CERTIFICATION UNDER 37 C.F.R. SECTION 1.10*

*(Express Mail label number is **mandatory**.)*
(Express Mail certification is optional.)

I hereby certify that this correspondence and the documents referred to as attached therein are being deposited with the United States Postal Service on November 10, 2000, in an envelope as "EXPRESS MAIL POST OFFICE TO ADDRESSEE" service under 37 C.F.R. Section 1.10, Mailing Label Number EL566709730US Addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

Jordan N. Bodner

(type or print name of person mailing paper)

Signature of person mailing paper

WARNING: *Certificate of mailing (first class) or facsimile transmission procedures of 37 C.F.R. Section 1.8 cannot be used to obtain a date of mailing or transmission for this correspondence.*

***WARNING:** Each paper or fee filed by "Express Mail" **must** have the number of the "Express Mail" mailing label placed thereon prior to mailing. 37 C.F.R. Section 1.10(b).*
*"Since the filing of correspondence under [Section] 1.10 without the Express Mail mailing label thereon is an oversight that can be avoided by the exercise of reasonable care, requests for waiver of this requirement will **not** be granted on petition." Notice of Oct. 24, 1996, 60 Fed. Reg. 56,439, at 56,442.*

MOTM_24063-1356736

Practitioner's Docket No. 03797.00072

1. The name of the inventor is (37 C.F.R. Section 1.51(c)(1)(ii)):

1. Leroy B. Keely

2. Residence address of the inventor, as numbered above (37 C.F.R. Section 1.51(c)(1)(iii)):

1. 210 Gabarda Way
Portola Valley, CA 94028

3. The title of the invention is (37 C.F.R. Section 1.51(c)(1)(iv)):

Simulating Gestures of a Mouse Using a Stylus and Providing Feedback Thereto

4. The name, registration, customer and telephone numbers of the practitioner are (37 C.F.R. Section 1.51(c)(1)(v)):

Name of practitioner: Pamela I. Banner
Reg. No. 33,644
Tel. No. 202-508-9100

5. The docket number used to identify this application is (37 C.F.R. Section 1.51(c)(1)(vi)):

Docket No. 03797.00072

6. The correspondence address for this application is (37 C.F.R. Section 1.51(c)(1)(vii)):

Pamela I. Banner
1001 G Street, N.W.
Washington, DC 20001

7. Statement as to whether invention was made by an agency of the U.S. Government or under contract with an agency of the U.S. Government. (37 C.F.R. Section 1.51(c)(1)(viii)).

This invention was NOT made by an agency of the United States Government, or under contract

MOTM_24063-1356737

Practitioner's Docket No. 03797.00072

with an agency of the United States Government.

8. Identification of documents accompanying this cover sheet:

A. Documents required by 37 C.F.R. Section 1.51(c)(2)-(3):

Specification:	No. of pages	16 Pages (including claims)
Drawings:	No. of sheets	3 (Figs. 1, 2, and 3)

B. Additional documents:

Claims:	No. of claims	3

9. Fee

The filing fee for this provisional application, as set in 37 C.F.R. Section 1.16(k), is $150.00 for other than a small entity.

10. Fee payment

Fee payment in the amount of $150.00 is being made at this time.

11. Method of fee payment

Charge Account No. 19-0733, in the amount of $150.00.

A duplicate of this Cover Sheet is attached.

Please charge Account No. 19-0733 for any fee deficiency.

Date: November 10, 2000

Reg. No. 42,338

Signature of Practitioner

Reg. No.: 33,644
Tel. No.: 202-508-9100

Pamela I. Banner
Banner & Witcoff, Ltd.
1001 G Street, N.W.
Washington, DC 20001

MOTM_24063-1356738

Express Mail Certificate EL566709730US

SIMULATING GESTURES OF A MOUSE USING A STYLUS AND PROVIDING FEEDBACK THERETO

PROVISIONAL PATENT APPLICATION

Cross-Reference to Related Applications

The present application is related to application Serial No. (Atty docket 3797.00066), entitled Method and Apparatus For Improving the Appearance of Digitally Represented Handwriting, filed concurrently with the present application; to application Serial No. (Atty docket 3797.00067), entitled Highlevel Active Pen Matrix, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00069), entitled Selection Handles in Editing Electronic Documents, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00070), entitled Insertion Point Bungee Space Tool, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00074), entitled System and Method for Accepting Disparate Types of User Input, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00075), entitled In Air Gestures, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00076), entitled Mouse Input Panel Windows Class List, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00077), entitled Mouse Input Panel and User Interface, and filed concurrently with the present application; to application Serial No. (Atty docket 3797.00079), entitled System and Method for Inserting Implicit Page

MOTM_24063-1356739

Breaks, and filed concurrently with the present application; each of which is incorporated by reference herein as to their entireties.

Field of the Invention

The present invention is directed generally to apparatus and methods for simulating a various gestures of a mouse in a computer system and for providing feedback thereto, and more specifically to simulating gestures such as the left click of a mouse, the right click of a mouse, and mouse dragging by manipulation of a stylus in conjunction with a touch-sensitive computer display, as well as generating appropriate visual or other feedback in response to certain gestures.

Background of the Invention

Typical computer systems, especially computer systems using graphical user interface (GUI) systems such as Microsoft WINDOWS, are optimized for accepting user input from two discrete input devices: a keyboard and for entering text, and a pointing device such as a mouse with two or more buttons for driving the user interface. Virtually all software applications designed to run on Microsoft WINDOWS are optimized to accept user input in the same manner. For instance, many applications make extensive use of the right mouse button (a "right click") to display context-sensitive command menus. It is noted that other operating systems incorporate right click operability as well. The user may generate other gestures using the mouse such as by clicking the left button of the mouse (a "left click"), or by clicking the left or right button of the mouse and

MOTM_24063-1356740

moving the mouse while the button is depressed (either a "left click drag" or a "right click drag").

While such mouse gestures are easily done with a mouse, it is not always convenient for a user to utilize a mouse with a computer even though the computer may have a GUI system. Once such type of system may be that which utilizes a touch-sensitive display screen and a stylus. The stylus tip is placed upon the display screen at certain locations to control the objects on the display screen. The stylus is limited in the types of actions that may be easily performed as compared with a mouse. For instance, the typical stylus can thus perform only three actions: placing the stylus tip onto the screen, moving the stylus tip across the screen, and removing the stylus tip from the screen. While a user may operate a remote toolbar to select a different tool, the excessive movement renders the excess movement laborious, inconvenient, and slow. There is presently no convenient way to simulate, for example, a right click as opposed to a left click using a stylus-type input device.

In order to make the full range of interface features accessible to users of such a stylus-based computer, there is a need for an intuitive way of simulating two-or-more button mouse gestures with the stylus. Moreover, it is preferable that any new way of simulating mouse gestures be compatible with existing software applications that conventionally are used with a mouse.

There is also a need for providing helpful feedback to the user of the stylus to indicate whether gestures made with the stylus are those that are intended by the user.

Summary of the Invention

MOTM_24063-1356741

As discussed in the various copending patent applications incorporated herein by reference, aspects of the present invention are directed to a tablet-like computer that may be used for directly writing on a touch-sensitive display surface using a stylus. The computer may allow the user to write and to edit, manipulate, and create objects through the use of the stylus. Many of the features discussed in these copending applications are more easily performed by use of the various aspects of the present invention discussed herein.

An aspect of the present invention is directed to methods and apparatus for simulating gestures of a mouse by use of a stylus on a touch-sensitive display surface. For example, the left click of a mouse may be simulated where the user holds the stylus down on the touch-sensitive display surface without substantial movement and then removes the stylus from the display surface before the expiration of a threshold amount of time. Also, the right click of a mouse may be simulated where the user instead waits until at least the threshold amount of time before removing the stylus. Where the stylus is moved along the display surface during these above stylus gestures, left or right click dragging may thereby be performed.

Another aspect of the present invention is directed to methods and apparatus for providing feedback to a user in a stylus-based touch-sensitive display surface computer system. The user may accordingly be given some indication that they have performed a stylus gesture correctly or incorrectly by, e.g., displaying something on the display surface to indicate a particular status. Such feedback may be responsive to certain stylus gestures such as pressing and holding the stylus against the display surface for at least a certain minimum amount of time. Also, the feedback may be in the form of a "feedback

MOTM_24063-1356742

indicator," which may be visual on the display surface such as an icon, bitmap, or other visual indicator, and/or auditory such as a sound emitted from the computer's speakers. The feedback indicator may change with stylus hold time and/or may be animated. Significant advantages of this aspect of the invention are realized especially in conjunction with the various features discussed in the copending patent applications incorporated herein by reference.

These and other features of the invention will be apparent upon consideration of the following detailed description of preferred embodiments. Although the invention has been defined using the appended claims, these claims are exemplary in that the invention is intended to include the elements and steps described herein in any combination or subcombination. Accordingly, there are any number of alternative combinations for defining the invention, which incorporate one or more elements from the specification, including the description, claims, and drawings, in various combinations or subcombinations. It will be apparent to those skilled in the relevant technology, in light of the present specification, that alternate combinations of aspects of the invention, either alone or in combination with one or more elements or steps defined herein, may be utilized as modifications or alterations of the invention or as part of the invention. It is intended that the written description of the invention contained herein covers all such modifications and alterations.

Brief Description of the Drawings

The foregoing summary of the invention, as well as the following detailed description of preferred embodiments, is better understood when read in conjunction with

MOTM_24063-1356743

the accompanying drawings, which are included by way of example, and not by way of limitation with regard to the claimed invention. In the accompanying drawings, elements are labeled with three-digit reference numbers, wherein the first digit of a reference number indicates the drawing number in which the element is first illustrated. The same reference number in different drawings refers to the same element.

Fig. 1 is a schematic diagram of a general-purpose digital computing environment that can be used to implement various aspects of the invention.

Fig. 2 is a plan view of a tablet computer and stylus that can be used in accordance with various aspects of the present invention.

Fig. 3 is a flow chart showing an exemplary set of steps that may be performed in order to simulate a right click of a mouse according to aspects of the present invention.

Detailed Description of Preferred Embodiments

FIG. 1 shows an exemplary embodiment of a general-purpose digital computing environment that can be used to implement various aspects of the invention. In this embodiment, computer 100 includes a processing unit 110, a system memory 120, and a system bus 130 that couples various system components including the system memory to the processing unit 110. The system bus 130 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory includes read only memory (ROM) 140 and random access memory (RAM) 150.

A basic input/output system 160 (BIOS), containing the basic routines that help to transfer information between elements within the computer 100, such as during start-up, is stored in ROM 140. Computer 100 also includes a hard disk drive 170 for

MOTM_24063-1356744

reading from and writing to a hard disk (not shown), a magnetic disk drive 180 for reading from or writing to a removable magnetic disk 190, and an optical disk drive 191 for reading from or writing to a removable optical disk 192 such as a CD ROM or other optical media. The hard disk drive 170, magnetic disk drive 180, and optical disk drive 191 are connected to the system bus 130 by a hard disk drive interface 192, a magnetic disk drive interface 193, and an optical disk drive interface 194, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for the personal computer 100. It will be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs), and the like, may also be used in the example operating environment.

A number of program modules can be stored on the hard disk, magnetic disk 190, optical disk 192, ROM 140 or RAM 150, including an operating system 195, one or more application programs 196, other program modules 197, and program data 198. A user can enter commands and information into the computer 100 through input devices such as a keyboard 101 and pointing device 102. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 110 through a serial port interface 106 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or a universal serial bus (USB). A monitor 107 or other type of display device is also connected to the system bus 130 via an interface, such as a video adapter 108. In addition to the

MOTM_24063-1356745

monitor, personal computers typically include other peripheral output devices (not shown), such as speakers and printers.

The computer 100 can operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 109. Remote computer 109 can be a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to computer 100, although only a memory storage device 111 has been illustrated in FIG. 1. The logical connections depicted in FIG. 1 include a local area network (LAN) 112 and a wide area network (WAN) 113. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the computer 100 is connected to the local network 112 through a network interface or adapter 114. When used in a WAN networking environment, the personal computer 100 typically includes a modem 115 or other means for establishing a communications over the wide area network 113, such as the Internet. The modem 115, which may be internal or external, is connected to the system bus 130 via the serial port interface 106. In a networked environment, program modules depicted relative to the personal computer 100, or portions thereof, may be stored in the remote memory storage device.

It will be appreciated that the network connections shown are example and other means of establishing a communications link between the computers can be used. The existence of any of various well-known protocols such as TCP/IP, Ethernet, FTP, HTTP and the like is presumed, and the system can be operated in a client-server configuration to permit a user to retrieve web pages from a web-based server. Any of

MOTM_24063-1356746

various conventional web browsers can be used to display and manipulate data on web pages.

Figure 2 illustrates a tablet PC 201 that can be used in accordance with various aspects of the present invention. Any or all of the features, subsystems, and functions in the system of Figure 1 can be included in the computer of Figure 2. Tablet PC 201 includes a large display surface 202, e.g., a digitizing flat panel display, preferably, a liquid crystal display (LCD) screen, on which a plurality of windows 203 is displayed. Using stylus 204, a user can select, highlight, and write on the digitizing display area. Examples of suitable digitizing display panels include electromagnetic pen digitizers, such as the Mutoh or Wacom pen digitizers. Other types of pen digitizers, e.g., optical digitizers, may also be used. Tablet PC 201 interprets marks made using stylus 204 in order to manipulate data, enter text, and execute conventional computer application tasks such as spreadsheets, word processing programs, and the like.

A stylus could be equipped with or without buttons or other features to augment its selection capabilities. In one embodiment, a stylus could be implemented as a "pencil" or "pen", in which one end constitutes a writing portion and the other end constitutes an "eraser" end, and which, when moved across the display, indicates portions of the display are to be erased. Other types of input devices, such as a mouse, trackball, or the like could be used. Additionally, a user's own finger could be used for selecting or indicating portions of the displayed image on a touch-sensitive or proximity-sensitive display. Consequently, the term "user input device", as used herein, is intended to have a broad definition and encompasses many variations on well-known input devices.

MOTM_24063-1356747

Referring now to Fig. 3, an example of how a left click drag of a mouse is simulated is now discussed. Any or all of steps set forth herein and in Fig. 3 may be performed by special software that runs in conjunction with, or is part of, the operating system of the computer 201. To simulate a left click drag, the user may touch the stylus 204 or other similar user input device to the display surface 202, an action referred to hereafter as "placing the stylus down" (step 301). Responsive to detecting the stylus being placed down, the computer 201 may begin counting time, e.g., by using a timer, up to a threshold amount of time. A timeout condition is defined responsive to the threshold amount of time having passed. In a preferred embodiment, the threshold amount of time is about 600 milliseconds. However, the threshold amount of time may be any amount of time, such as between 300 and 600 milliseconds, or between 600 milliseconds and 1 second, or greater then 1 second. If the stylus moves prior to the timeout condition occurring (step 302), then in response the computer 201 may generate a standard Microsoft WINDOWS LeftMouseButtonDown event in the original position where the stylus 204 was first brought down (step 314), which may be sent to the software application presently in use. At this point, any subsequent movement of the stylus 204 across the display surface 202 may be detected (step 315), an in response the computer 201 may generate one or more Microsoft WINDOWS MouseMove events (step 316). When the stylus 204 is eventually removed from the display surface 202 (hereafter referred to as "bringing the stylus up") (step 317), the computer 201 in response may generate a Microsoft WINDOWS LeftMouseButtonUp event (step 318). Thus, via steps 301, 302, and 314-318, a left click drag of a mouse is simulated.

MOTM_24063-1356748

To simulate a left click of a mouse without dragging, if the stylus 204 is not moved prior to the timeout condition (step 302), but if the stylus 204 is instead brought up prior to the timeout condition (step 303), then in response the computer 201 may generate first a LeftMouseButtonDown event and then a LeftMouseButtonUp event (steps 304 and 318). Thus, via steps 301-304 and 318, a slightly delayed left click of a mouse is simulated.

To simulate a right click of a mouse without dragging, if the stylus 204 is not moved either prior to the timeout condition or after the timeout condition (steps 302 and 305), and if the stylus 204 is not brought up until after the timeout condition (steps 303 and 306), then in response the computer 201 may generate first a Microsoft WINDOWS RightMouseButtonDown event (step 307), then delay for a period of time (step 308), preferably about 20 milliseconds, and then generate a Microsoft WINDOWS RightMouseButtonUp event (step 313). The computer 201 may also generate and display a feedback indicator that indicates to the user that the stylus 204 has been placed down for a certain minimum amount of time (step 319), as will be discussed below in more detail. Note that step 319 may be in a variety of places within the flowchart of Fig. 3 other than the one shown, such as between steps 305 and 306. Steps 301-303, 305-308, and 313, thus simulate a delayed right click of a mouse.

To simulate a right click drag of a mouse, if the stylus 204 is not moved until after the timeout condition (step 305), then in response the computer 201 may generate a RightMouseButtonDown event (step 309). Upon the computer 201 detecting any subsequent movement of the stylus 204 along the display surface 202 (step 310), the computer 201 may in response generate one or more MouseMove events (step 311).

MOTM_24063-1356749

Once the stylus 204 is brought up (312), in response the computer 201 may generate a RightMouseButtonUp event (step 313). This simulates, via steps 301-303, 305, and 309-313, a right click drag of a mouse.

As a further feature of the right click simulation, a special hold event may be defined wherein the stylus 204 is held against the display screen 202 without substantial movement for at least another threshold amount of time, say about 2 seconds, or about 5 seconds or more, the computer 201 may detect such a hold event and in response act by immediately interpreting any subsequent holding of the stylus 204 as holding the right mouse button down. Thus, the computer 201 may, after 600 milliseconds of stylus holding, go on to steps 305 and 319, and where the stylus 204 is released prior to, say, 5 seconds, then steps 306-308 and 313 may be implemented. However, if instead the stylus 204 is further held at least 5 seconds, then the special subroutine may, in response, immediately generate a RightMouseButtonDown event. Then, when the stylus is eventually released, a RightMouseButtonUp event would be generated by the special subroutine. If the stylus 204 is moved along the display surface 202, then the computer 201 may immediately in response generate one or more MouseMove events. When the stylus 204 is eventually removed from the display surface 202, all pending RightMouse events that would normally have been generated in accordance with the method shown in Fig. 3 would be canceled.

In the above and remaining discussion, it is assumed that a Microsoft WINDOWS operating system is being used with the computer 201. However, any operating system, especially a GUI operating system, may be used in accordance with the present invention. Further, although the above embodiments are the preferred methods for simulating mouse

MOTM_24063-1356750

gestures using a stylus, there are many variations that are encompassed by the present invention. For instance, the stylus gestures required for simulation a left click and a right click might be reversed, or a user may not need to always keep the stylus 204 down on the display surface 202 during the entire time that a drag is executed, such that a first touch/touch-and-hold of the stylus 204 at a first location on the display surface 202, and a second touch/touch-and-hold at a second location, may thereby simulate a drag from the first location to the second location.

As discussed in the copending patent applications incorporated by reference herein, the computer 201 according to embodiments of the present invention may have the ability to select ink when in pen mode. A user can press and hold, then drag over ink, text, or other objects, within a document to select those objects. A press and hold may switch into selection mode. Because the user is using the stylus 204 or other similar user input device, there are different considerations as compared with when a mouse is used. For instance, while the user of a mouse may hear and feel the click of a mouse button to ensure that a click has been performed, the user of the stylus 204 may not receive clear enough feedback that the intended gesture has been made properly. Also, it is desirable to indicate to the user that a particular requested mode has really been switched to, such as selection mode. Further, where the user must hold down the stylus 204 for at least a certain threshold amount of time to simulate, for example, a right click, feedback to the user indicating that the threshold time has passed would be useful.

Such feedback may be in the form of a feedback indicator, which may be visual (e.g., a graphical icon, a symbol, a bitmapped or other image, etc., displayed on the display surface 202 such as feedback indicator 205 in Fig. 2), auditory (e.g., a click, a

MOTM_24063-1356751

beep, tone, etc. emanating from the computer 201 and/or stylus 204), and/or tactile (e.g., vibration of the computer 201 and/or stylus 204). For example, where the user selects the pen icon in the application toolbar (as described in at least one of the copending patent applications incorporated herein by reference), the user may drag the stylus 204 over a document displayed on the display surface 202 such that digital ink is generated on the display surface 202. At this point, the user may switch to selection mode by pressing and holding the stylus 204 for at least a threshold amount of time that may be the same as the threshold amount of time or that may be different. For instance, the threshold minimum amount of time that the user must hold the stylus 204 against the display surface 202 may preferably be about 400 milliseconds, or between about 400 milliseconds and about 1 second, or more than about 1 second. Responsive to the user holding without substantial movement the stylus 204 against the display surface 202 for the minimum threshold amount of time, the computer 201 may generate a feedback indicator on the display surface 202 which may preferably be located at or near the location of the tip of the stylus 204.

In some embodiments, the feedback indicator may be animated and/or may otherwise change over time. For instance, the feedback indicator 205 is shown as a partially completed clock-like loop that may become more complete over time in, e.g., a clockwise direction. In such embodiments with animation, the feedback indicator may actually begin prior to the threshold amount of time passing, as an indication to the user that the threshold amount of time will soon pass. For instance, where a first threshold amount of time is 1 second, holding the stylus 204 against the display surface 202 for 500 milliseconds may cause an icon (a feedback indicator) in the shape of a wand tip to

MOTM_24063-1356752

appear on the display surface 202 at or near the location of the stylus 204. As the stylus 204 is continued to be held, the icon may animate and begin to glow to indicate that a gesture is beginning to form. At the first threshold amount of time, the animation may have reached the point where the wand tip is fully glowing. Moreover, at the passing of the first threshold of time, the user may thereby be placed in selection mode which is indicated by the status of the feedback indicator.

As a further possibility in connection with the above example, at a second threshold amount of time of, e.g., 2 seconds, the wand tip may further animate and/or change to indicate that a right click will now be simulated when the stylus 204 is removed from the display surface 202.

Where the user removes the stylus 204 from the display surface 202 prior to the threshold of time passing, then any subsequent stylus movement may be interpreted by the computer 201 as normal stylus input. Also, in this situation any feedback indicator and/or animation thereof may be immediately canceled and removed from the display surface 202.

While exemplary systems and methods embodying the present invention are shown by way of example, it will be understood, of course, that the invention is not limited to these embodiments. Modifications may be made by those skilled in the art, particularly in light of the foregoing teachings. For example, each of the elements of the aforementioned embodiments may be utilized alone or in combination with elements of the other embodiments.

MOTM_24063-1356753

We claim:

1. A method for simulating a gesture such as a right click of a mouse using a stylus on a touch-sensitive display surface.

2. A method for simulating a right click of a mouse comprising the steps of:

touching and holding a stylus against a touch-sensitive display surface for at least a threshold amount of time; and

then releasing the stylus from the touch-sensitive display surface.

3. A method for providing a feedback indicator to a user responsive to a stylus being held down on a display surface of a computer for at least a threshold amount of time, thereby indicating a status.

DRAFT - 4-16-2005

MOTM_24063-1356754

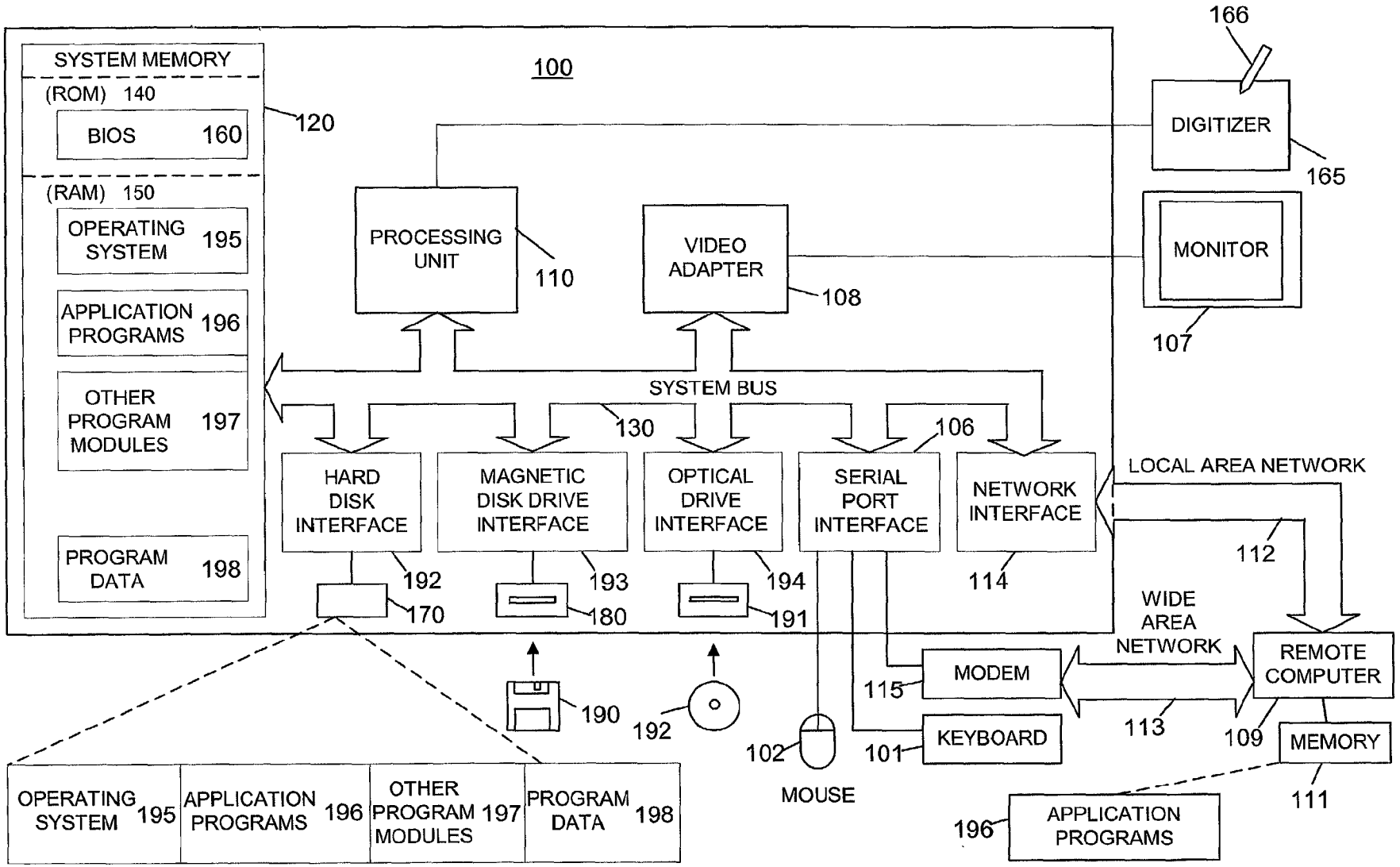


FIG. 1

MOTM_24063-1356755

03797.00072 2 of 3

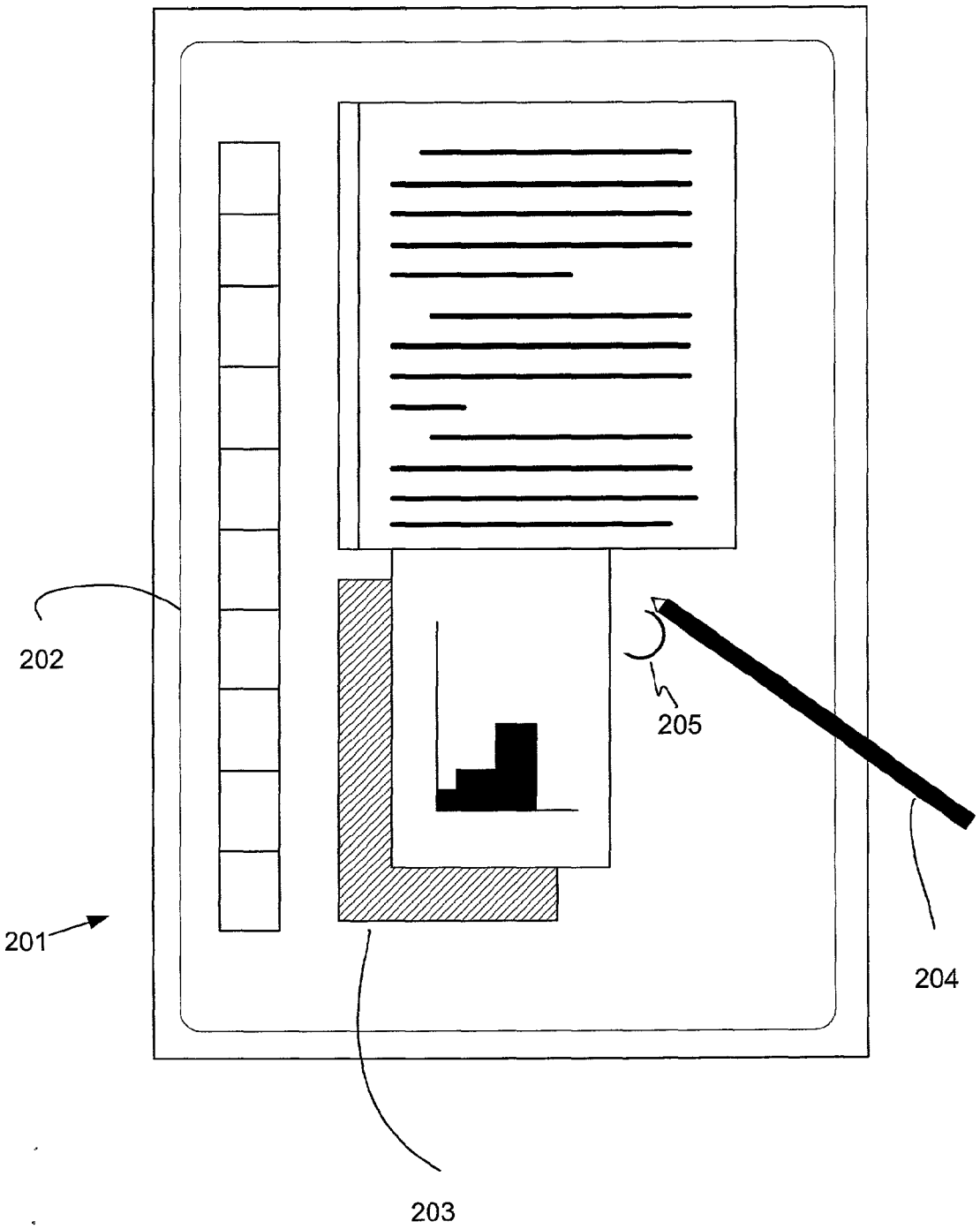


FIG. 2

03797.00072 3 of 3

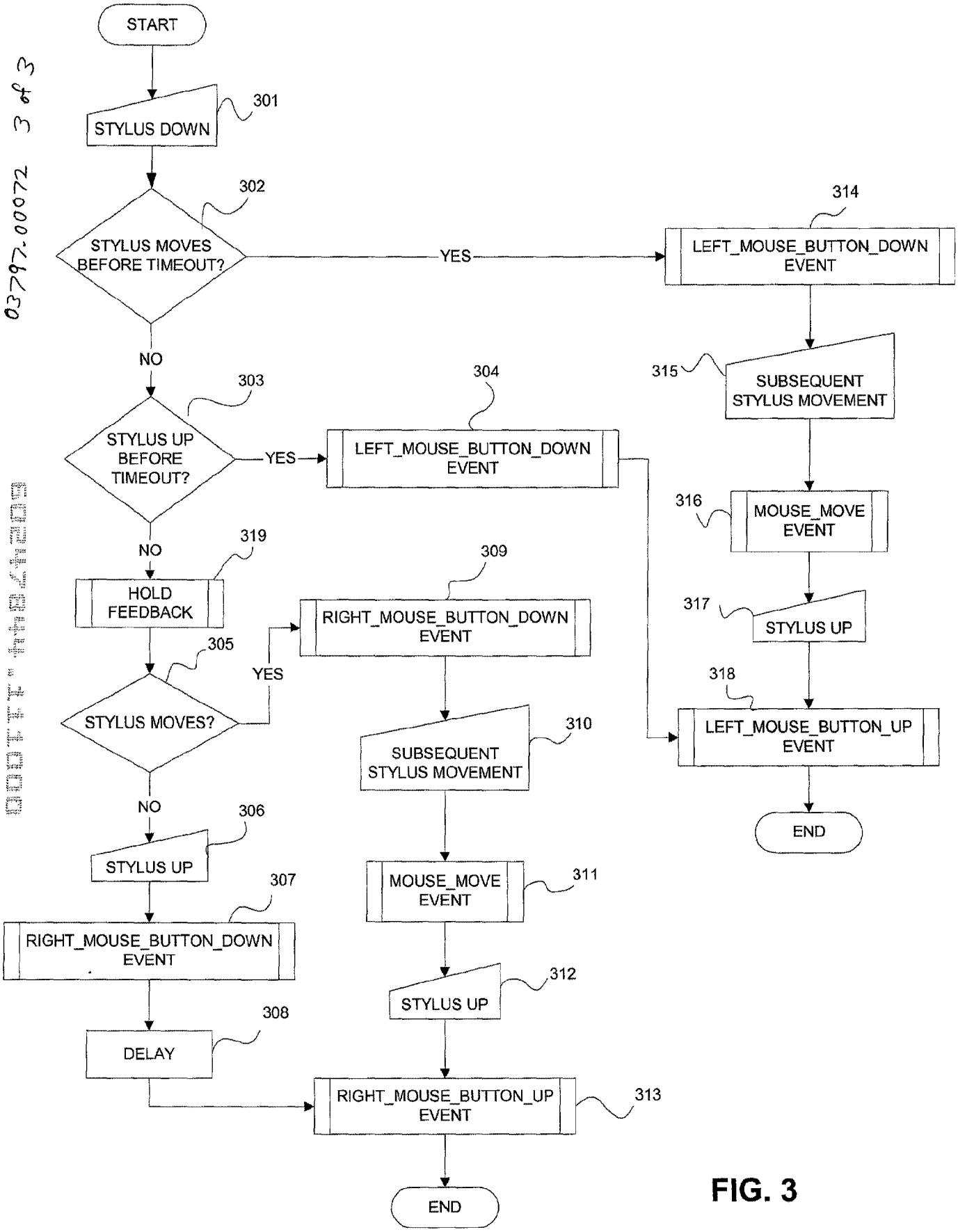


FIG. 3

About Direct2D

This topic introduces Direct2D, a new 2-D graphics API for Windows 7. Direct2D provides Win32 developers with the ability to perform 2-D graphics rendering tasks with superior performance and visual quality.

What is Direct2D?

Direct2D is a hardware-accelerated, immediate-mode 2-D graphics API that provides high performance and high-quality rendering for 2-D geometry, bitmaps, and text. The Direct2D API is designed to interoperate with existing code that uses GDI, GDI+, or Direct3D.

Direct2D is designed primarily for use by the following classes of developers:

- Developers of large, enterprise-scale, native applications.
- Developers who create control toolkits and libraries for consumption by downstream developers.
- Developers who require server-side rendering of 2-D graphics.
- Developers who use Direct3D graphics and need simple, high-performance 2-D and text rendering for menus, user interface (UI) elements, and Heads-up Displays (HUDs).

Why Direct2D?

The primary motivations for creating a new 2-D graphics API in Microsoft Windows include the following:

- To keep pace with the increasing level of visual richness that Windows users are accustomed to.
- To enable developers to write 2-D rendering code that scales directly with the graphics processing hardware of the PC it is running on.
- To enable developers to write code for rendering 2-D graphics that can run in the context of a service.

In recent years, end users have begun to expect greater visual fidelity in digital experiences. This trend is reflected in consumer electronics. GPS devices, media playback devices, mobile phones, and digital cameras deliver consistently richer experiences year after year. This trend can also be seen in the diversity of graphical content in film, television, video games, and on the Web. To keep pace with these changes, developers are consistently asked to take their existing Windows applications to the next level of visual richness.

Graphics processors in modern Windows PCs have also been evolving steadily, driven by advances in video game graphics and parts of the Windows experience like Windows Media Center and Aero. Some Windows applications can take advantage of modern GPUs by using Microsoft Direct3D and Windows Presentation Foundation (WPF). While Direct3D serves high-end 3-D graphics applications and WPF addresses the needs of .NET developers, there are gaps for developers who have large existing codebases of rendering code based on GDI and GDI+ or who want to incorporate high-quality 2-D graphics in their Direct3D-based applications.

Finally, the need for a graphics API that can be used in a service has become an emerging requirement for developers working in enterprise and Web-development scenarios. Existing rendering APIs focus on client-side rendering in a single user session. As such, they can fall short in areas of robustness and scalability when used in a service context. A new API is required to address this.

High Performance with Maximum Availability

Direct2D is a user-mode library that is built using the Direct3D 10.1 API. This means that Direct2D applications benefit from hardware-accelerated rendering on modern mainstream GPUs. Hardware acceleration is also achieved on earlier Direct3D 9 hardware by using Direct3D 10-level-9 rendering. This combination provides excellent performance on graphics hardware on existing Windows PCs.

The following diagram shows the layered architecture of Direct2D.

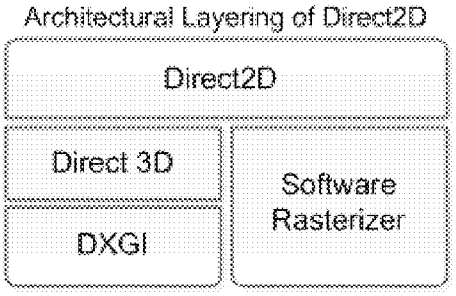


Diagram of the Direct2D layered architecture

For scenarios where the use of hardware acceleration is not feasible, Direct2D includes a high-performance software rasterizer. When rendering in software, applications that use Direct2D experience substantially better rendering performance than with GDI+ and with similar visual quality. The software rasterizer is also designed for use in a service context.

Content that is rendered by using Direct2D can also be displayed remotely by using the Remote Desktop Protocol (RDP) infrastructure in the Windows 7 operating system. Developers can select whether rendering is handled by the GPU on the display computer or rendered locally and transmitted as bitmaps. This choice can be made based on the required fill-rate and the quantity of graphics primitives that are rendered. When the display computer is running an operating system earlier than Windows 7, remote display rendering is performed by transmitting bitmaps over the network.

By providing a single API that combines the performance of Direct3D and high availability by providing software fallback, remote desktop, and service rendering, Direct2D enables developers to have a single implementation for high-performance rendering in many different scenarios.

Visual Quality

Applications that use Direct2D for graphics can deliver higher visual quality than what can be achieved using GDI. Direct2D uses per-primitive antialiasing to deliver smoother looking curves and lines in rendered content. There is also full support for transparency and alpha blending when rendering 2-D primitives. The following images compare aliased content rendered by using GDI (on the left) with antialiased content rendered by Direct2D (on the right).

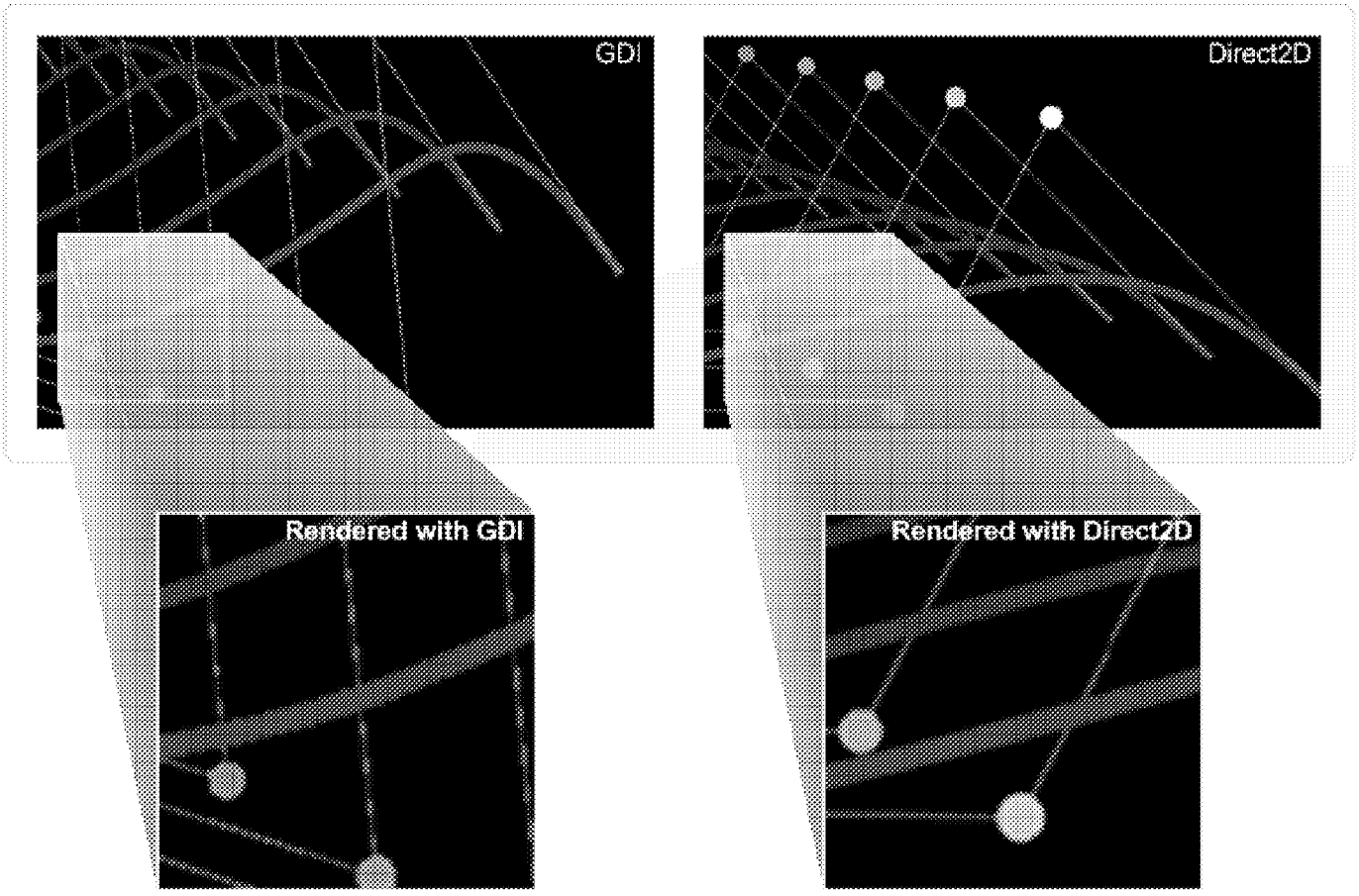


Illustration of curves and lines that are rendered in GDI and in Direct2D

Developers can specify aliased rendering of vector graphics for scenarios where it is required. This is used in scenarios where snapping to hard pixel boundaries is required, such as UI elements like pointers or rulers, if the GDI style of output must be matched, or if antialiasing will be performed downstream in the rendering process via Multisample Antialiasing or some other mechanism.

Interoperability

Integrating Direct2D-based rendering is made easier for developers through surface-level interoperability with GDI and Direct3D. Applications that render content primarily with GDI, GDI+, or Direct3D, can begin by using Direct2D to render specific areas of their application, and over time move to a model where rendering is performed primarily via Direct2D, using GDI primarily for plug-ins or legacy extensibility.

Direct2D also makes it easy to use DirectWrite[1], the new text API, and the advanced imaging features of the Microsoft Windows Imaging Component (WIC)[2].

For more information about Direct2D interoperability, see the Interoperability section of the Direct2D SDK.[3]

Summary

Microsoft Direct2D enables developers to build 2-D graphics features in their applications that deliver improved visual quality over GDI, and performance characteristics that scale with modern GPUs. The Direct2D interoperability model enables developers to selectively migrate portions of their application at a time alongside GDI, GDI+, or Direct3D-based rendering.

Related Topics

Getting Started with Direct2D[4]
Direct2D API Overview[5]

Send comments about this topic to Microsoft

Build date: 7/8/2010

Links Table

[1]*http://msdn.microsoft.com/en-us/library/dd368038(v=VS.85).aspx*

[2]*http://msdn.microsoft.com/en-us/library/ms737408.aspx*

[3]*http://msdn.microsoft.com/en-us/library/dd756743(v=VS.85).aspx*

[4]*http://msdn.microsoft.com/en-us/library/dd756670(v=VS.85).aspx*

[5]*http://msdn.microsoft.com/en-us/library/dd317121(v=VS.85).aspx*

Community Content

I agree with TerryLicia

I'm just an old user who was knowledgeable back in DOS days. I stopped teaching before Windows came in and I only know a little about a lot of computer things nowadays. I find that the "...for Dummies" books are what I can understand now! Please help us understand the basics of this.

5/10/2011
marjit2

Direct2D? API? WTH?

I'm just a user. A friendly end user. And 9/10th's of the time I have NO idea what the *** the "more information" means after I've read it! From sentence ONE, I'm lost. I'm not a developer, I'm do not write code. WHY DOESNT MS HAVE A TINY ONE OR TWO SENTENCES AT THE VERY BEGINNING OF ALLLLLLL THESE UPDATES AND TELL US IF WE, THE USERS, NEED TO HAVE THIS STUFF OR NOT?! If we don't even know what the dickens the EXPLANATIONS are, then why should be bother downloading ANYTHING? Isnt' what you sold us good enough - ohhh for a month or so? I started getting "updates" that would do zetherlating and then some bodundoing, and ended with mystifying the xbargodiddle to the point it just would go lightspeed into utter prweblogerdgumdo! Yeah ... and that is EXACTLY what all these updates mean to me, a friendly end user.

MAKE IT FOR US OR FOR DEVELOPERS ... BUT DONT SEND IT OUT TO EVERYONE WITHOUT SAYING, IN THE FIRST SENTENCE, WHICH ONE OF US IT IS FOR! QUIT WASTING MY DAMN TIME, YA CHOWDERHEADS. For being such geniuses, you can be the most stupid kids on the block, I swear!

5/3/2011
TerryLicia

Chet's Coments

Well I'd shure like to download anything that will help me get Windows #7 back

4/29/2011
Chester Phelps

© 2011 Microsoft. All rights reserved.

About Keyboard Input

Applications should accept user input from the keyboard as well as from the mouse. An application receives keyboard input in the form of messages posted to its windows.

This section covers the following topics:

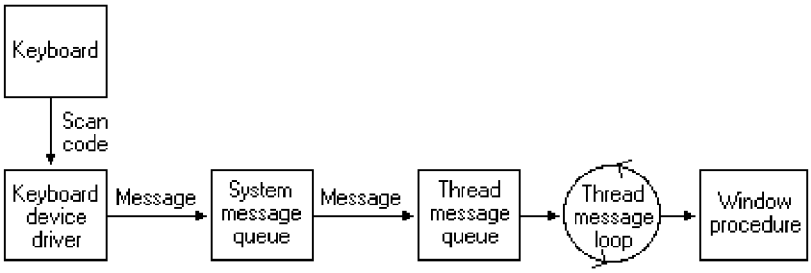
- Keyboard Input Model
- Keyboard Focus and Activation
- Keystroke Messages
  - System and Nonsystem Keystrokes
  - Virtual-Key Codes Described
  - Keystroke Message Flags
- Character Messages
  - Nonsystem Character Messages
  - Dead-Character Messages
- Key Status
- Keystroke and Character Translations
- Hot-Key Support
- Keyboard Keys for Browsing and Other Functions
- Simulating Input
- Languages, Locales, and Keyboard Layouts

Keyboard Input Model

The system provides device-independent keyboard support for applications by installing a keyboard device driver appropriate for the current keyboard. The system provides language-independent keyboard support by using the language-specific keyboard layout currently selected by the user or the application. The keyboard device driver receives scan codes from the keyboard, which are sent to the keyboard layout where they are translated into messages and posted to the appropriate windows in your application.

Assigned to each key on a keyboard is a unique value called a *scan code*, a device-dependent identifier for the key on the keyboard. A keyboard generates two scan codes when the user types a key—one when the user presses the key and another when the user releases the key.

The keyboard device driver interprets a scan code and translates (maps) it to a *virtual-key code*, a device-independent value defined by the system that identifies the purpose of a key. After translating a scan code, the keyboard layout creates a message that includes the scan code, the virtual-key code, and other information about the keystroke, and then places the message in the system message queue. The system removes the message from the system message queue and posts it to the message queue of the appropriate thread. Eventually, the thread's message loop removes the message and passes it to the appropriate window procedure for processing. The following figure illustrates the keyboard input model.



Keyboard Focus and Activation

The system posts keyboard messages to the message queue of the foreground thread that created the

window with the keyboard focus. The *keyboard focus* is a temporary property of a window. The system shares the keyboard among all windows on the display by shifting the keyboard focus, at the user's direction, from one window to another. The window that has the keyboard focus receives (from the message queue of the thread that created it) all keyboard messages until the focus changes to a different window.

A thread can call the **GetFocus**[1] function to determine which of its windows (if any) currently has the keyboard focus. A thread can give the keyboard focus to one of its windows by calling the **SetFocus**[2] function. When the keyboard focus changes from one window to another, the system sends a **WM_KILLFOCUS**[3] message to the window that has lost the focus, and then sends a **WM_SETFOCUS**[4] message to the window that has gained the focus.

The concept of keyboard focus is related to that of the active window. The *active window* is the top-level window the user is currently working with. The window with the keyboard focus is either the active window, or a child window of the active window. To help the user identify the active window, the system places it at the top of the Z order and highlights its title bar (if it has one) and border.

The user can activate a top-level window by clicking it, selecting it using the ALT+TAB or ALT+ESC key combination, or selecting it from the Task List. A thread can activate a top-level window by using the **SetActiveWindow**[5] function. It can determine whether a top-level window it created is active by using the **GetActiveWindow**[6] function.

When one window is deactivated and another activated, the system sends the **WM_ACTIVATE**[7] message. The low-order word of the *wParam* parameter is zero if the window is being deactivated and nonzero if it is being activated. When the default window procedure receives the **WM_ACTIVATE** message, it sets the keyboard focus to the active window.

To block keyboard and mouse input events from reaching applications, use **BlockInput**[8]. Note, the **BlockInput** function will not interfere with the asynchronous keyboard input-state table. This means that calling the **SendInput**[9] function while input is blocked will change the asynchronous keyboard input-state table.

Keystroke Messages

Pressing a key causes a **WM_KEYDOWN**[10] or **WM_SYSKEYDOWN**[11] message to be placed in the thread message queue attached to the window that has the keyboard focus. Releasing a key causes a **WM_KEYUP**[12] or **WM_SYSKEYUP**[13] message to be placed in the queue.

Key-up and key-down messages typically occur in pairs, but if the user holds down a key long enough to start the keyboard's automatic repeat feature, the system generates a number of **WM_KEYDOWN**[10] or **WM_SYSKEYDOWN**[11] messages in a row. It then generates a single **WM_KEYUP**[12] or **WM_SYSKEYUP**[13] message when the user releases the key.

This section covers the following topics:

- System and Nonsystem Keystrokes
- Virtual-Key Codes Described
- Keystroke Message Flags

System and Nonsystem Keystrokes

The system makes a distinction between system keystrokes and nonsystem keystrokes. System keystrokes produce system keystroke messages, **WM_SYSKEYDOWN**[11] and **WM_SYSKEYUP**[13]. Nonsystem keystrokes produce nonsystem keystroke messages, **WM_KEYDOWN**[10] and **WM_KEYUP**[12].

If your window procedure must process a system keystroke message, make sure that after processing the message the procedure passes it to the **DefWindowProc**[14] function. Otherwise, all system operations involving the ALT key will be disabled whenever the window has the keyboard focus. That is, the user won't be able to access the window's menus or System menu, or use the ALT+ESC or ALT+TAB key combination to activate a different window.

System keystroke messages are primarily for use by the system rather than by an application. The system uses them to provide its built-in keyboard interface to menus and to allow the user to control which window is active. System keystroke messages are generated when the user types a key in combination with the ALT key, or when the user types and no window has the keyboard focus (for example, when the active application is minimized). In this case, the messages are posted to the message queue attached to the active window.

Nonsystem keystroke messages are for use by application windows; the **DefWindowProc**[14] function does nothing with them. A window procedure can discard any nonsystem keystroke messages that it does not need.

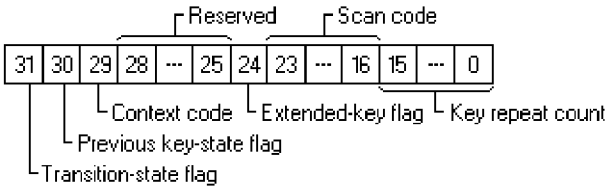
Virtual-Key Codes Described

The **wParam** parameter of a keystroke message contains the virtual-key code of the key that was pressed or released. A window procedure processes or ignores a keystroke message, depending on the value of the virtual-key code.

A typical window procedure processes only a small subset of the keystroke messages that it receives and ignores the rest. For example, a window procedure might process only **WM_KEYDOWN**[10] keystroke messages, and only those that contain virtual-key codes for the cursor movement keys, shift keys (also called control keys), and function keys. A typical window procedure does not process keystroke messages from character keys. Instead, it uses the **TranslateMessage**[15] function to convert the message into character messages. For more information about **TranslateMessage** and character messages, see Character Messages.

Keystroke Message Flags

The **lParam** parameter of a keystroke message contains additional information about the keystroke that generated the message. This information includes the repeat count, the scan code, the extended-key flag, the context code, the previous key-state flag, and the transition-state flag. The following illustration shows the locations of these flags and values in the **lParam** parameter.



An application can use the following values to manipulate the keystroke flags.

Value	Description
KF_ALTDOWN	Manipulates the ALT key flag, which indicates whether the ALT key is pressed.
KF_DLGMODE	Manipulates the dialog mode flag, which indicates whether a dialog box is active.
KF_EXTENDED	Manipulates the extended key flag.
KF_MENUMODE	Manipulates the menu mode flag, which indicates whether a menu is active.
KF_REPEAT	Manipulates the repeat count.
KF_UP	Manipulates the transition state flag.

Repeat Count

You can check the repeat count to determine whether a keystroke message represents more than one keystroke. The system increments the count when the keyboard generates **WM_KEYDOWN**[10] or **WM_SYSKEYDOWN**[11] messages faster than an application can process them. This often occurs when the user holds down a key long enough to start the keyboard's automatic repeat feature. Instead of filling the system message queue with the resulting key-down messages, the system combines the messages into a single key down message and increments the repeat count. Releasing a key cannot start the automatic repeat feature, so the repeat count for **WM_KEYUP**[12] and **WM_SYSKEYUP**[13] messages is always set to 1.

Scan Code

The scan code is the value that the keyboard hardware generates when the user presses a key. It is a device-dependent value that identifies the key pressed, as opposed to the character represented by the key. An application typically ignores scan codes. Instead, it uses the device-independent virtual-key codes to interpret keystroke messages.

Extended-Key Flag

The extended-key flag indicates whether the keystroke message originated from one of the additional keys on the enhanced keyboard. The extended keys consist of the ALT and CTRL keys on the right-hand side of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; the NUM LOCK key; the BREAK (CTRL+PAUSE) key; the PRINT SCRN key; and the divide (/) and ENTER keys in the numeric keypad. The extended-key flag is set if the key is an extended key.

Context Code

The context code indicates whether the ALT key was down when the keystroke message was generated. The code is 1 if the ALT key was down and 0 if it was up.

Previous Key-State Flag

The previous key-state flag indicates whether the key that generated the keystroke message was previously up or down. It is 1 if the key was previously down and 0 if the key was previously up. You can use this flag to identify keystroke messages generated by the keyboard's automatic repeat feature. This flag is set to 1 for **WM_KEYDOWN**[10] and **WM_SYSKEYDOWN**[11] keystroke messages generated by the automatic repeat feature. It is always set to 0 for **WM_KEYUP**[12] and **WM_SYSKEYUP**[13] messages.

Transition-State Flag

The transition-state flag indicates whether pressing a key or releasing a key generated the keystroke message. This flag is always set to 0 for **WM_KEYDOWN**[10] and **WM_SYSKEYDOWN**[11] messages; it is always set to 1 for **WM_KEYUP**[12] and **WM_SYSKEYUP**[13] messages.

Character Messages

Keystroke messages provide a lot of information about keystrokes, but they do not provide character codes for character keystrokes. To retrieve character codes, an application must include the **TranslateMessage**[15] function in its thread message loop. **TranslateMessage** passes a **WM_KEYDOWN**[10] or **WM_SYSKEYDOWN**[11] message to the keyboard layout. The layout examines the message's virtual-key code and, if it corresponds to a character key, provides the character code

equivalent (taking into account the state of the SHIFT and CAPS LOCK keys). It then generates a character message that includes the character code and places the message at the top of the message queue. The next iteration of the message loop removes the character message from the queue and dispatches the message to the appropriate window procedure.

This section covers the following topics:

- Nonsystem Character Messages
- Dead-Character Messages

Nonsystem Character Messages

A window procedure can receive the following character messages: **WM_CHAR**[16], **WM_DEADCHAR**[17], **WM_SYSCHAR**[18], **WM_SYSDEADCHAR**[19], and **WM_UNICHAR**[20]. The **TranslateMessage**[15] function generates a **WM_CHAR** or **WM_DEADCHAR** message when it processes a **WM_KEYDOWN**[10] message. Similarly, it generates a **WM_SYSCHAR** or **WM_SYSDEADCHAR** message when it processes a **WM_SYSKEYDOWN**[11] message.

An application that processes keyboard input typically ignores all but the **WM_CHAR**[16] and **WM_UNICHAR**[20] messages, passing any other messages to the **DefWindowProc**[14] function. Note that **WM_CHAR** uses 16-bit Unicode Transformation Format (UTF) while **WM_UNICHAR** uses UTF-32. The system uses the **WM_SYSCHAR**[18] and **WM_SYSDEADCHAR**[19] messages to implement menu mnemonics.

The **wParam** parameter of all character messages contains the character code of the character key that was pressed. The value of the character code depends on the window class of the window receiving the message. If the Unicode version of the **RegisterClass**[21] function was used to register the window class, the system provides Unicode characters to all windows of that class. Otherwise, the system provides ASCII character codes. For more information, see Unicode and Character Sets[22].

The contents of the **lParam** parameter of a character message are identical to the contents of the **lParam** parameter of the key-down message that was translated to produce the character message. For information, see Keystroke Message Flags.

Dead-Character Messages

Some non-English keyboards contain character keys that are not expected to produce characters by themselves. Instead, they are used to add a diacritic to the character produced by the subsequent keystroke. These keys are called *dead keys*. The circumflex key on a German keyboard is an example of a dead key. To enter the character consisting of an "o" with a circumflex, a German user would type the circumflex key followed by the "o" key. The window with the keyboard focus would receive the following sequence of messages:

1. **WM_KEYDOWN**[10]
2. **WM_DEADCHAR**[17]
3. **WM_KEYUP**[12]
4. **WM_KEYDOWN**[10]
5. **WM_CHAR**[16]
6. **WM_KEYUP**[12]

TranslateMessage[15] generates the **WM_DEADCHAR**[17] message when it processes the **WM_KEYDOWN**[10] message from a dead key. Although the *wParam* parameter of the **WM_DEADCHAR** message contains the character code of the diacritic for the dead key, an application typically ignores the

message. Instead, it processes the **WM_CHAR**[16] message generated by the subsequent keystroke. The *wParam* parameter of the **WM_CHAR** message contains the character code of the letter with the diacritic. If the subsequent keystroke generates a character that cannot be combined with a diacritic, the system generates two **WM_CHAR** messages. The *wParam* parameter of the first contains the character code of the diacritic; the *wParam* parameter of the second contains the character code of the subsequent character key.

The **TranslateMessage**[15] function generates the **WM_SYSDEADCHAR**[19] message when it processes the **WM_SYSKEYDOWN**[11] message from a system dead key (a dead key that is pressed in combination with the ALT key). An application typically ignores the **WM_SYSDEADCHAR** message.

Key Status

While processing a keyboard message, an application may need to determine the status of another key besides the one that generated the current message. For example, a word-processing application that allows the user to press SHIFT+END to select a block of text must check the status of the SHIFT key whenever it receives a keystroke message from the END key. The application can use the **GetKeyState**[23] function to determine the status of a virtual key at the time the current message was generated; it can use the **GetAsyncKeyState**[24] function to retrieve the current status of a virtual key.

The keyboard layout maintains a list of names. The name of a key that produces a single character is the same as the character produced by the key. The name of a noncharacter key such as TAB and ENTER is stored as a character string. An application can retrieve the name of any key from the device driver by calling the **GetKeyNameText**[25] function.

Keystroke and Character Translations

The system includes several special purpose functions that translate scan codes, character codes, and virtual-key codes provided by various keystroke messages. These functions include **MapVirtualKey**[26], **ToAscii**[27], **ToUnicode**[28], and **VkKeyScan**[29].

In addition, Microsoft Rich Edit 3.0 supports the HexToUnicode IME[30], which allows a user to convert between hexadecimal and Unicode characters by using hot keys. This means that when Microsoft Rich Edit 3.0 is incorporated into an application, the application will inherit the features of the HexToUnicode IME.

Hot-Key Support

A *hot key* is a key combination that generates a **WM_HOTKEY**[31] message, a message the system places at the top of a thread's message queue, bypassing any existing messages in the queue. Applications use hot keys to obtain high-priority keyboard input from the user. For example, by defining a hot key consisting of the CTRL+C key combination, an application can allow the user to cancel a lengthy operation.

To define a hot key, an application calls the **RegisterHotKey**[32] function, specifying the combination of keys that generates the **WM_HOTKEY**[31] message, the handle to the window to receive the message, and the identifier of the hot key. When the user presses the hot key, a **WM_HOTKEY** message is placed in the message queue of the thread that created the window. The *wParam* parameter of the message contains the identifier of the hot key. The application can define multiple hot keys for a thread, but each hot key in the thread must have a unique identifier. Before the application terminates, it should use the **UnregisterHotKey**[33] function to destroy the hot key.

Applications can use a hot key control to make it easy for the user to choose a hot key. Hot key controls are typically used to define a hot key that activates a window; they do not use the **RegisterHotKey**[32] and **UnregisterHotKey**[33] functions. Instead, an application that uses a hot key control typically sends

the **WM_SETHOTKEY**[34] message to set the hot key. Whenever the user presses the hot key, the system sends a **WM_SYSCOMMAND**[35] message specifying SC_HOTKEY. For more information about hot key controls, see "Using Hot Key Controls" in Hot Key Controls[36].

Keyboard Keys for Browsing and Other Functions

Windows provides support for keyboards with special keys for browser functions, media functions, application launching, and power management. The **WM_APPCOMMAND**[37] supports the extra keyboard keys. In addition, the **ShellProc**[38] function is modified to support the extra keyboard keys.

It is unlikely that a child window in a component application will be able to directly implement commands for these extra keyboard keys. So when one of these keys is pressed, **DefWindowProc**[14] will send a **WM_APPCOMMAND**[37] message to a window. **DefWindowProc** will also bubble the **WM_APPCOMMAND** message to its parent window. This is similar to the way context menus are invoked with the right mouse button, which is that **DefWindowProc** sends a **WM_CONTEXTMENU**[39] message on a right button click, and bubbles it to its parent. Additionally, if **DefWindowProc** receives a **WM_APPCOMMAND** message for a top-level window, it will call a shell hook with code HSHELL_APPCOMMAND.

Windows also supports the Microsoft IntelliMouse Explorer, which is a mouse with five buttons. The two extra buttons support forward and backward browser navigation. For more information, see XBUTTONs[40].

Simulating Input

To simulate an uninterrupted series of user input events, use the **SendInput**[9] function. The function accepts three parameters. The first parameter, *cInputs*, indicates the number of input events that will be simulated. The second parameter, *rgInputs*, is an array of **INPUT**[41] structures, each describing a type of input event and additional information about that event. The last parameter, *cbSize*, accepts the size of the **INPUT** structure, in bytes.

The **SendInput**[9] function works by injecting a series of simulated input events into a device's input stream. The effect is similar to calling the **keybd_event**[42] or **mouse_event**[43] function repeatedly, except that the system ensures that no other input events intermingle with the simulated events. When the call completes, the return value indicates the number of input events successfully played. If this value is zero, then input was blocked.

The **SendInput**[9] function does not reset the keyboard's current state. Therefore, if the user has any keys pressed when you call this function, they might interfere with the events that this function generates. If you are concerned about possible interference, check the keyboard's state with the **GetAsyncKeyState**[24] function and correct as necessary.

Languages, Locales, and Keyboard Layouts

A *language* is a natural language, such as English, French, and Japanese. A *sublanguage* is a variant of a natural language that is spoken in a specific geographical region, such as the English sublanguages spoken in Great Britain and the United States. Applications use values, called language identifiers[44], to uniquely identify languages and sublanguages.

Applications typically use *locales* to set the language in which input and output is processed. Setting the locale for the keyboard, for example, affects the character values generated by the keyboard. Setting the locale for the display or printer affects the glyphs displayed or printed. Applications set the locale for a keyboard by loading and using keyboard layouts. They set the locale for a display or printer by selecting a font that supports the specified locale.

A keyboard layout not only specifies the physical position of the keys on the keyboard but also determines the character values generated by pressing those keys. Each layout identifies the current input language and determines which character values are generated by which keys and key combinations.

Every keyboard layout has a corresponding handle that identifies the layout and language. The low word of the handle is a language identifier. The high word is a device handle, specifying the physical layout, or is zero, indicating a default physical layout. The user can associate any input language with a physical layout. For example, an English-speaking user who very occasionally works in French can set the input language of the keyboard to French without changing the physical layout of the keyboard. This means the user can enter text in French using the familiar English layout.

Applications are generally not expected to manipulate input languages directly. Instead, the user sets up language and layout combinations, then switches among them. When the user clicks into text marked with a different language, the application calls the **ActivateKeyboardLayout**[45] function to activate the user's default layout for that language. If the user edits text in a language which is not in the active list, the application can call the **LoadKeyboardLayout**[46] function with the language to get a layout based on that language.

The **ActivateKeyboardLayout**[45] function sets the input language for the current task. The *hkl* parameter can be either the handle to the keyboard layout or a zero-extended language identifier. Keyboard layout handles can be obtained from the **LoadKeyboardLayout**[46] or **GetKeyboardLayoutList**[47] function. The HKL_NEXT and HKL_PREV values can also be used to select the next or previous keyboard.

The **GetKeyboardLayoutName**[48] function retrieves the name of the active keyboard layout for the calling thread. If an application creates the active layout using the **LoadKeyboardLayout**[46] function, **GetKeyboardLayoutName** retrieves the same string used to create the layout. Otherwise, the string is the primary language identifier corresponding to the locale of the active layout. This means the function may not necessarily differentiate among different layouts with the same primary language, so cannot return specific information about the input language. The **GetKeyboardLayout**[49] function, however, can be used to determine the input language.

The **LoadKeyboardLayout**[46] function loads a keyboard layout and makes the layout available to the user. Applications can make the layout immediately active for the current thread by using the KLF_ACTIVATE value. An application can use the KLF_REORDER value to reorder the layouts without also specifying the KLF_ACTIVATE value. Applications should always use the KLF_SUBSTITUTE_OK value when loading keyboard layouts to ensure that the user's preference, if any, is selected.

For multilingual support, the **LoadKeyboardLayout**[46] function provides the KLF_REPLACELANG and KLF_NOTELLSHELL flags. The KLF_REPLACELANG flag directs the function to replace an existing keyboard layout without changing the language. Attempting to replace an existing layout using the same language identifier but without specifying KLF_REPLACELANG is an error. The KLF_NOTELLSHELL flag prevents the function from notifying the shell when a keyboard layout is added or replaced. This is useful for applications that add multiple layouts in a consecutive series of calls. This flag should be used in all but the last call.

The **UnloadKeyboardLayout**[50] function is restricted in that it cannot unload the system default input language. This ensures that the user always has one layout available for enter text using the same character set as used by the shell and file system.

Send comments about this topic to Microsoft

Links Table

[1] *http://msdn.microsoft.com/en-us/library/ms646294(v=VS.85).aspx*

[2] *http://msdn.microsoft.com/en-us/library/ms646312(v=VS.85).aspx*

[3] *http://msdn.microsoft.com/en-us/library/ms646282(v=VS.85).aspx*

[4] *http://msdn.microsoft.com/en-us/library/ms646283(v=VS.85).aspx*

[5] *http://msdn.microsoft.com/en-us/library/ms646311(v=VS.85).aspx*

[6] *http://msdn.microsoft.com/en-us/library/ms646292(v=VS.85).aspx*

[7] *http://msdn.microsoft.com/en-us/library/ms646274(v=VS.85).aspx*

[8] *http://msdn.microsoft.com/en-us/library/ms646290(v=VS.85).aspx*

[9] *http://msdn.microsoft.com/en-us/library/ms646310(v=VS.85).aspx*

[10] *http://msdn.microsoft.com/en-us/library/ms646280(v=VS.85).aspx*

[11] *http://msdn.microsoft.com/en-us/library/ms646286(v=VS.85).aspx*

[12] *http://msdn.microsoft.com/en-us/library/ms646281(v=VS.85).aspx*

[13] *http://msdn.microsoft.com/en-us/library/ms646287(v=VS.85).aspx*

[14] *http://msdn.microsoft.com/en-us/library/ms633572(v=VS.85).aspx*

[15] *http://msdn.microsoft.com/en-us/library/ms644955(v=VS.85).aspx*

[16] *http://msdn.microsoft.com/en-us/library/ms646276(v=VS.85).aspx*

[17] *http://msdn.microsoft.com/en-us/library/ms646277(v=VS.85).aspx*

[18] *http://msdn.microsoft.com/en-us/library/ms646357(v=VS.85).aspx*

[19] *http://msdn.microsoft.com/en-us/library/ms646285(v=VS.85).aspx*

[20] *http://msdn.microsoft.com/en-us/library/ms646288(v=VS.85).aspx*

[21] *http://msdn.microsoft.com/en-us/library/ms633586(v=VS.85).aspx*

[22] *http://msdn.microsoft.com/en-us/library/dd374083(v=VS.85).aspx*

[23] *http://msdn.microsoft.com/en-us/library/ms646301(v=VS.85).aspx*

[24] *http://msdn.microsoft.com/en-us/library/ms646293(v=VS.85).aspx*

[25] *http://msdn.microsoft.com/en-us/library/ms646300(v=VS.85).aspx*

[26] *http://msdn.microsoft.com/en-us/library/ms646306(v=VS.85).aspx*

[27] *http://msdn.microsoft.com/en-us/library/ms646316(v=VS.85).aspx*

[28] *http://msdn.microsoft.com/en-us/library/ms646320(v=VS.85).aspx*

[29] *http://msdn.microsoft.com/en-us/library/ms646329(v=VS.85).aspx*

[30] *http://msdn.microsoft.com/en-us/library/dd318146(v=VS.85).aspx*

[31] *http://msdn.microsoft.com/en-us/library/ms646279(v=VS.85).aspx*

[32] *http://msdn.microsoft.com/en-us/library/ms646309(v=VS.85).aspx*

[33] *http://msdn.microsoft.com/en-us/library/ms646327(v=VS.85).aspx*

[34] *http://msdn.microsoft.com/en-us/library/ms646284(v=VS.85).aspx*

[35] *http://msdn.microsoft.com/en-us/library/ms646360(v=VS.85).aspx*

[36] *http://msdn.microsoft.com/en-us/library/bb775233(v=VS.85).aspx*

[37] *http://msdn.microsoft.com/en-us/library/ms646275(v=VS.85).aspx*

[38] *http://msdn.microsoft.com/en-us/library/ms644991(v=VS.85).aspx*

[39] *http://msdn.microsoft.com/en-us/library/ms647592(v=VS.85).aspx*

[40] *http://msdn.microsoft.com/en-us/library/ms645601(v=VS.85).aspx#_win32_XBUTTONs*

[41] *http://msdn.microsoft.com/en-us/library/ms646270(v=VS.85).aspx*

[42] *http://msdn.microsoft.com/en-us/library/ms646304(v=VS.85).aspx*

[43] *http://msdn.microsoft.com/en-us/library/ms646260(v=VS.85).aspx*

[44] *http://msdn.microsoft.com/en-us/library/dd318691(v=VS.85).aspx*

[45] *http://msdn.microsoft.com/en-us/library/ms646289(v=VS.85).aspx*

[46] *http://msdn.microsoft.com/en-us/library/ms646305(v=VS.85).aspx*

[47] *http://msdn.microsoft.com/en-us/library/ms646297(v=VS.85).aspx*

[48] *http://msdn.microsoft.com/en-us/library/ms646298(v=VS.85).aspx*

[49] *http://msdn.microsoft.com/en-us/library/ms646296(v=VS.85).aspx*

[50] *http://msdn.microsoft.com/en-us/library/ms646324(v=VS.85).aspx*

Community Content

Previous Key-State Flag

"It is always set to 0 for WM_KEYUP and WM_SYSKEYUP messages." It's incorrect. At least in WinXP it is set to 1 for these two messages.

10/26/2009
dyonis 

© 2011 Microsoft. All rights reserved.