US 20020180742A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2002/0180742 A1

Hamid (43) Pub. Date: Dec. 5, 2002

(54) GRAPHICS MACROS FOR A FRAME BUFFER

(76) Inventor: **Hammad Hamid**, Berkshire (GB)

Correspondence Address:
C. DOUGLAS MCDONALD ESQ.
CARLTON FIELDS ET AL
P.O. BOX 3239
TAMPA, FL 33601-3239 (US)

(21) Appl. No.: **09/772,523**

(22) Filed: **Jan. 29, 2001**

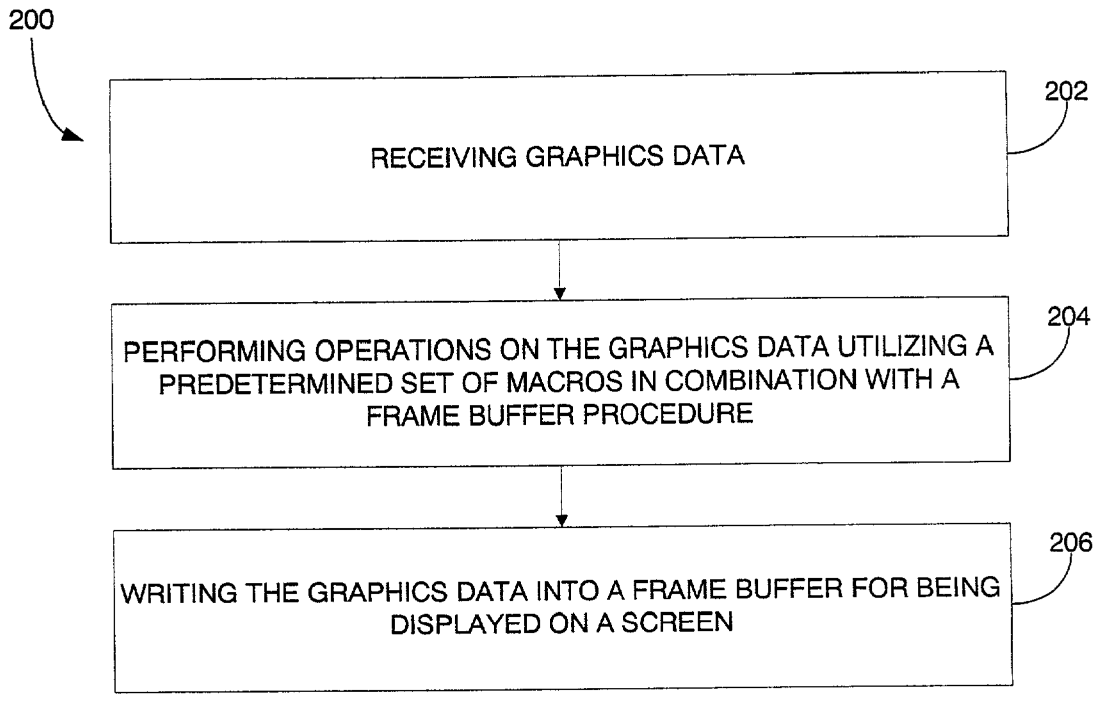
Publication Classification

(51) **Int. Cl.**^{7} **G09G 5/39**; G09G 5/36

(52) **U.S. Cl.** **345/531**; 345/545

(57) **ABSTRACT**

A system, method and article of manufacture are provided for enhanced handling of graphics data in a frame buffer. Initially, graphics data is received. Thereafter, operations are performed on the graphics data utilizing a predetermined set of macros in combination with a frame buffer procedure. In use, the graphics data is written into a frame buffer for being displayed on a screen.


200
RECEIVING GRAPHICS DATA
202
PERFORMING OPERATIONS ON THE GRAPHICS DATA UTILIZING A PREDETERMINED SET OF MACROS IN COMBINATION WITH A FRAME BUFFER PROCEDURE
204
WRITING THE GRAPHICS DATA INTO A FRAME BUFFER FOR BEING DISPLAYED ON A SCREEN
206

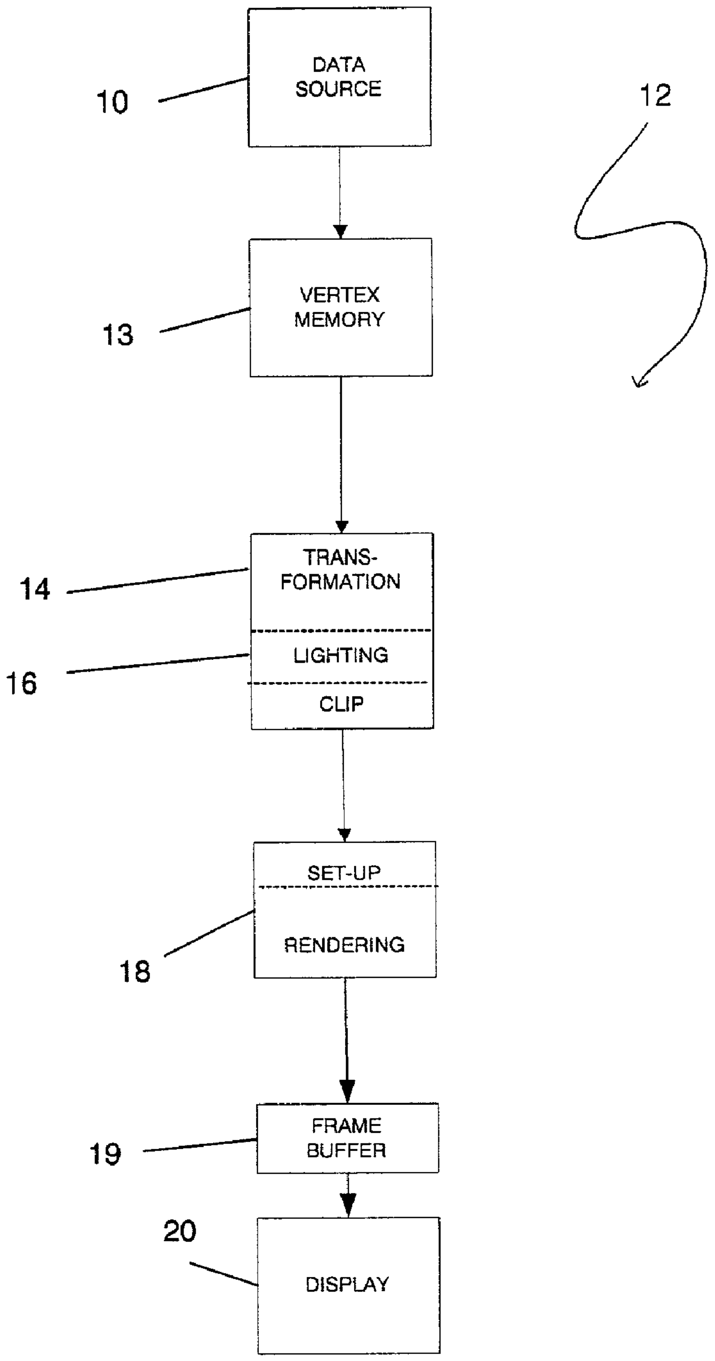


Figure 1
(PRIOR ART)

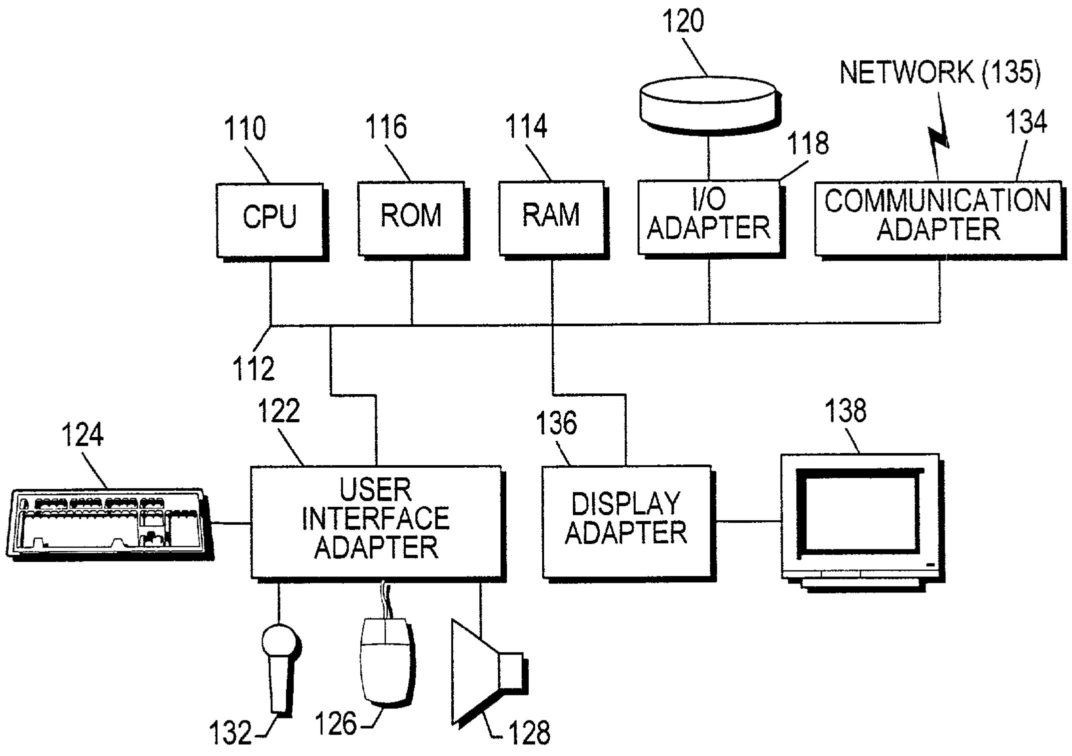

120
NETWORK (135)
110
116
114
118
134
CPU
ROM
RAM
I/O ADAPTER
COMMUNICATION ADAPTER
112
124
122
136
138
USER INTERFACE ADAPTER
DISPLAY ADAPTER
132
126
128

Fig. 1A

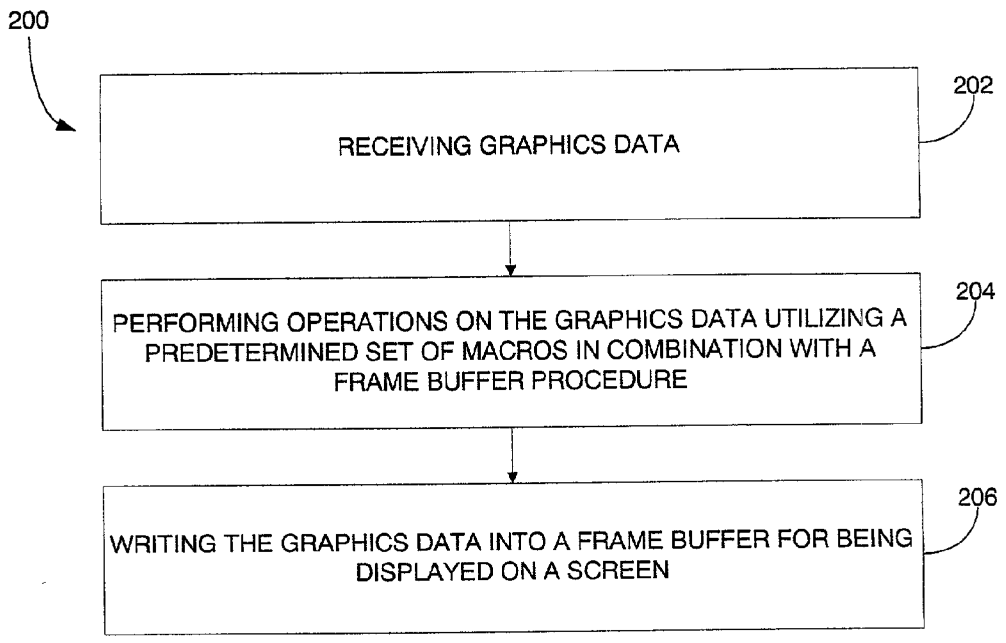

200
RECEIVING GRAPHICS DATA
202
PERFORMING OPERATIONS ON THE GRAPHICS DATA UTILIZING A PREDETERMINED SET OF MACROS IN COMBINATION WITH A FRAME BUFFER PROCEDURE
204
WRITING THE GRAPHICS DATA INTO A FRAME BUFFER FOR BEING DISPLAYED ON A SCREEN
206

Fig. 2

Filename	Purpose
lcddriver.h	Driver for the LCD display.
framebuf.h	Runs the frame buffer.

300

Fig. 3

400

Macro Name	Type	Purpose
print_shape	proc	Called from gfx_drawRect to set the parameters to initiat drawing a rectangle
gfx_interpret_Line	proc	Interprets the shape edges when drawing a triangle.

402

Macro Name	Type	Purpose
gfx_printShapeProcess	proc	Processes all the shape drawing function calls, and contains the actual shape drawing code. This must be run in the main par loop.
gfx_drawRect	proc	Sets the parameters to initiate drawing a rectangle.
gfx_drawTriangle	proc	Sets the parameters to initiate drawing a triangle
gfx_drawCircle	proc	Sets the parameters to initiate drawing a circle
gfx_drawLine	proc	Draws a line between two points
gfx_clearScreen	proc	Sets the parameters to clear the screen to a certain colour.
gfx_draw1BitIcon	proc	Draws a bitmap to the screen from a 1-bit icon rom. (Widths of rom index may need adjusting for different sized roms).
gfx_draw24BitIcon	proc	Draws a bitmap to the screen from a 24-bit icon rom. (Widths of rom index may need adjusting for different sized roms).

Fig. 4

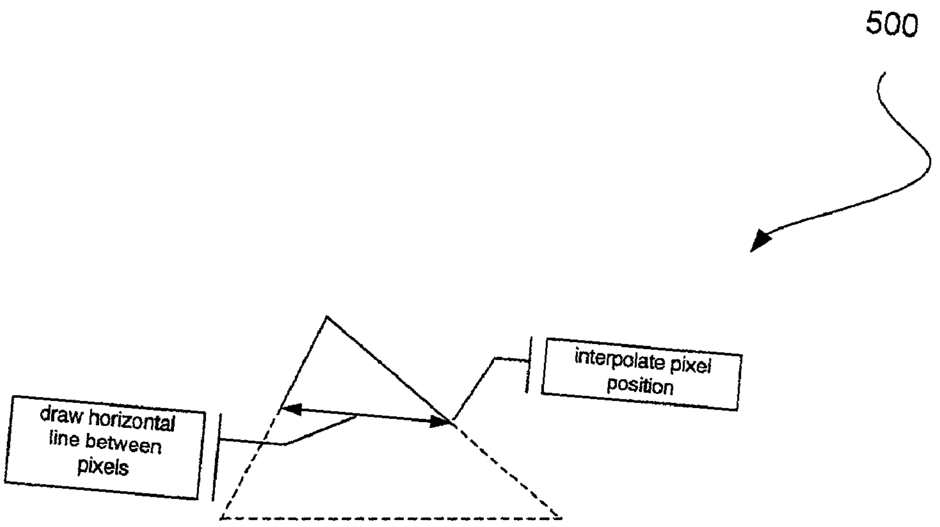

500
draw horizontal line between pixels
interpolate pixel position

Fig. 5

600

Macro	**Clock Speed Estimate (MHz)**
gfx_drawRect	28.9
gfx_drawTriangle	25.6
gfx_drawCircle	15.7

Fig. 6

GRAPHICS MACROS FOR A FRAME BUFFER

FIELD OF THE INVENTION

[0001] The present invention relates to graphics processing systems and more particularly to improving frame buffering in graphics processing systems.

BACKGROUND OF THE INVENTION

[0002] Rendering and displaying three-dimensional graphics typically involves many calculations and computations. For example, to render a three dimensional object, a set of coordinate points or vertices that define the object to be rendered must be formed. Vertices can be joined to form polygons that define the surface of the object to be rendered and displayed. Once the vertices that define an object are formed, the vertices must be transformed from an object or model frame of reference to a world frame of reference and finally to two-dimensional coordinates that can be displayed on a flat display device. Along the way, vertices may be rotated, scaled, eliminated or clipped because they fall outside the viewable area, lit by various lighting schemes, colorized, and so forth. Thus the process of rendering and displaying a three-dimensional object can be computationally intensive and may involve a large number of vertices.

[0003] A general system that implements a graphics pipeline system is illustrated in Prior Art **FIG. 1**. In this system, data source **10** generates a stream of expanded vertices defining primitives. These vertices are passed one at a time, through pipelined graphic system **12** via vertex memory **13** for storage purposes. Once the expanded vertices are received from the vertex memory **13** into the pipelined graphic system **12**, the vertices are transformed and lit by a transformation module **14** and a lighting module **16**, respectively, and further clipped and set-up for rendering by a rasterizer **18**, thus generating rendered primitives that are stored in a frame buffer **19** and then displayed on display device **20**.

[0004] During operation, the transform module **14** may be used to perform scaling, rotation, and projection of a set of three dimensional vertices from their local or model coordinates to the two dimensional window that will be used to display the rendered object. The lighting module **16** sets the color and appearance of a vertex based on various lighting schemes, light locations, ambient light levels, materials, and so forth. The rasterization module **18** rasterizes or renders vertices that have previously been transformed and/or lit. The rasterization module **18** renders the object to a rendering target which can be a display device or intermediate hardware or software structure that in turn moves the rendered data to a display device.

[0005] The frame buffer **19** has an address for each pixel component of the display **20**. The frame buffer **19** is also accessed by another mechanism that reads the contents of the frame buffer **19** to create the corresponding pixel by pixel image upon a the display **20**. Typically, the display **20** will be a color CRT with red, green and blue (RGB) electron guns whose intensities are varied by discrete steps to produce a wide range of colors. Accordingly, the frame buffer **19** is divided into portions containing multi-bit values for each color of every pixel. The preferred way to do this is to organize the frame buffer **19** into "planes" which each receive the same address. Each plane holds one bit at each address. Planes are grouped together to form multi-bit values for the attributes of the pixels they represent. Attributes include the RGB intensities, and in many systems ON and OFF for pixels in an "overlay" plane that is merged with data in other planes. For instance, an overlay plane might contain a cursor, and the presence of a bit in the overlay plane might force saturation intensity for all three electron guns, regardless of the actual RGB values for that pixel. In graphics systems with two-dimensional displays that are intended for use with solid modeling of three-dimensional objects, there is frequently another attribute that is stored for each pixel: its depth. Hardware storage of depth values greatly facilitates hidden surface removal, as it allows the hardware to automatically suppress pixels that are not upon the outer surface facing the viewer.

[0006] In accordance with what has been described above, it is not unusual to find graphics systems with between twenty-four and forty planes of frame buffer memory: perhaps three sets of eight for RGB values and sixteen or more for Z, or depth, values. Considering that the monitor could easily be 1280 pixels wide and 1024 pixels high, and that refreshing the display at a power line frequency of 60 Hz is a requirement, it can be concluded that a new pixel of twenty -four or more bits (and possibly qualified for depth) must be obtained for the monitor from the frame buffer at a rate of approximately one pixel every nine nanoseconds. To some extent the advent of so called "video display RAM'S" has made this easier to do. They have special high speed ports that read blocks of data at high speed for use by a shifter that, when grouped with the shifters of other planes for the same color, produce the multi-bit values for color intensity. These multi-bit values are applied to digital-to-analog converters (DAC's) that in turn generate the signals that actually drive the electron guns.

[0007] Despite the video RAM's, formidable problems remain concerning the task of getting the data into the frame buffer in the first place. In the long run, the graphics system will not be able to manipulate an image (draw it, rotate it, cut a hole in it, etc.) any faster than the image can be put into the frame buffer. The speed with which this can be done is one important aspect of "high performance" in a graphics system. Recalling the purpose for caching in a conventional computer system, it will be noted that there is a certain similarity. It would be desirable if a way could be found to cache pixels into a high speed memory and reduce the number of write operations made into the frame buffer. If this could be done without sacrificing other desirable features it would significantly increase the rate with which data could be put into the frame buffer. This is indeed desirable, since much work has been done to develop and perfect dedicated hardware to generate at high speed pixel values from a more abstract description of the image to be rendered.

[0008] Each plane of frame buffer memory is equipped with a corresponding plane of a pixel cache. The pixel rendering hardware stores computed pixel values into the frame buffer by way of the cache. Those familiar with pixel rendering mechanisms will appreciate that the order in which pixels are calculated is not necessarily related to the order they are accessed for use in driving the monitor, which is typically vertically by horizontal rows for a raster scanned CRT. Instead, pixels are apt to be generated in an order that makes sense in light of the techniques being used to represent the object. A wire frame model would rely heavily on the drawing of arbitrarily oriented vectors, while shaded polygons would rely heavily upon an area fill based on successive horizontal lines of pixels. For a curved surface the successive horizontal lines are apt to be fairly short, may be of varying lengths, and might not line up exactly above or beneath each other. Clearly, the preferred pixel rendering techniques are no respecters of sequentially addressed memory spaces. Yet the sequence of generated pixels are still strongly related by just more than being consecutive members in some order of pixel generation; their locations in the final image are physically "close" to each other. That is, sequentially generated pixels are apt to posses a shared "locality." That this is so has been noticed by others, and has been termed the "principle of locality." It seems clear that to maximize the number of hits, a cache for a frame buffer ought to operate in view of the principle of locality. But it is also clear that a different type of locality obtains for area fill operations than does for arbitrary vectors.

[0009] A "tile" is a rectangular collection of pixels. Various schemes for manipulating pixels in groups as tiles have been proposed. It would seem that what a pixel cache for a frame buffer ought to do, at least in part, is cache a tile. But again, the tile shape best suited for area fill operations would be one that is one pixel high by some suitably long number of pixels. The optimum tile shape for the drawing of arbitrary vectors can be shown to be a square. So what is needed then, is a pixel cache whose "shape" is adjustable according to the type of tile best suited for use with the type of pixel rendering to be undertaken.

[0010] That object can be achieved by a pixel cache, frame buffer controller and frame buffer memory organization that cooperate to implement a cache corresponding to a tile of adjustable rectangular dimensions. The frame buffer memory organization involves dividing the frame buffer into a number of separately addressable groups. Each group is composed of one or more bits. Along the scan lines of the raster groups repeat in a regular order. Successive scan lines have different starting groups in the pattern of repetition. Thus, whether a tile proceeds horizontally along a scan line, or vertically across successive scan lines, different groups are accessed for the pixels in that tile. This allows the entire tile to be fetched with one memory cycle. In such a scheme adjacent pixel ad dresses do not necessarily map into adjacent frame buffer addresses, as in conventional bit-mapped displays. Instead, an address manipulator within the frame buffer controller converts a pixel address (screen location) into a collection of addresses (one for each group) according to rules determined by the shape of the tile to be accessed.

[0011] Each plane of the frame buffer memory includes a sixteen-bit plane of an RGB pixel cache and a sixteen-bit plane of a Z value cache. (It will be understood, of course, that the number sixteen is merely exemplary, and is not the only practical size of pixel cache.) For each bit in a pixel's RGB values, the pixel's (X, Y) location on the monitor is mapped into the proper location of the plane of the RGB cache associated with that bit. If there is a hit, then the pixel is written to the cache. If there is a miss, then the cache is written out to the frame buffer in accordance with a replacement rule similar to those used with so-called "line movers" or "bitblts." The replacement rule uses sixteen-bit registers named SOURCE, DESTINATION and PATTERN. There is one of these registers for each plane of frame buffer memory. At the time of the preceding miss, each DESTINATION, and not the cache, was loaded with a copy of that region (tile) of the frame buffer that the cache was then to represent. Data was then written to the cache until there was a miss. Then the frame buffer controller simultaneously copied all of the bits of each plane in the cache into each SOURCE; this frees the cache for immediate use in storing new pixel values. The frame buffer controller proceeded to combine each SOURCE with its associated DESTINATION according to the desired rule (OR, AND, XOR, etc.). The result was further modified by the associated PATTERN, which can be used to impose special deviations upon the pixel data. For example, PATTERN might suppress a regular succession of pixels to create "holes" into which might later be placed pixels of another object, thus creating the illusion of transparency. However achieved, the result is written, all sixteen bits in parallel, for each plane, to the frame buffer. The mapping of pixel addresses into the cache and the parallel write into the frame buffer (i.e., the mapping of the cache contents back into frame buffer addresses) are automatically adjusted according to the size and shape of the tile being handled. Thus, one aspect of the invention to be disclosed is a pixel cache memory that accepts programmatically variable tile sizes. It will be further understood as the description proceeds that the tiles may be aligned on selected pixel boundaries, and that those boundaries need not be permanently fixed in advance.

[0012] Up to now, prior art frame buffer procedures such as the foregoing example execute command after command without the use of macros of any sort.

SUMMARY OF THE INVENTION

[0013] A system, method and article of manufacture are provided for enhanced handling of graphics data in a frame buffer. Initially, graphics data is received. Thereafter, operations are performed on the graphics data utilizing a predetermined set of macros in combination with a frame buffer procedure. In use, the graphics data is written into a frame buffer for being displayed on a screen.

[0014] In one embodiment of the present invention, the macros may include a function identifier and parameters associated therewith. Further, the operations may include a wait operation. Moreover, the operations may include a draw rectangle operation, draw circle operation, and/or draw triangle operation.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] The invention will be better understood when consideration is given to the following detailed description thereof. Such description makes reference to the annexed drawings wherein:

[0016] **FIG. 1** illustrates a prior art graphics pipeline;

[0017] **FIG. 1A** is a schematic diagram of a hardware implementation of one embodiment of the present invention;

[0018] **FIG. 2** illustrates a method for enhanced handling of graphics data in a frame buffer;

[0019] **FIG. 3** illustrates the various external files may be may be needed by the present invention;

[0020] **FIG. 4** illustrates internal and external macros associated with the present invention;

[0021] **FIG. 5** illustrates the Draw Triangle Macro, in accordance with one embodiment of the present invention; and

[0022] **FIG. 6** illustrates a chart showing speeds with which the various macros may operate.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0023] A preferred embodiment of a system in accordance with the present invention is preferably practiced in the context of a personal computer such as an IBM compatible personal computer, Apple Macintosh computer or UNIX based workstation. A representative hardware environment is depicted in **FIG. 1A**, which illustrates a typical hardware configuration of a works tation in accordance with a preferred embodiment having a central processing unit **110**, such as a microprocessor, and a number of other units interconnected via a system bus **112**. The workstation shown in **FIG. 1A** includes a Random Access Memory (RAM) **114**, Read Only Memory (ROM) **116**, an I/O adapter **118** for connecting peripheral devices such as disk storage units **120** to the bus **112**, a user interface adapter **122** for connecting a keyboard **124**, a mouse **126**, a speaker **128**, a microphone **132**, and/or other user interface devices such as a touch screen (not shown) to the bus **112**, communication adapter **134** for connecting the workstation to a communication network (e.g., a data processing network) and a display adapter **136** for connecting the bus **112** to a display device **138**. The workstation typically has resident thereon an operating system such as the Microsoft Windows NT or Windows/95 Operating System (OS), the IBM OS/2 operating system, the MAC OS, or UNIX operating system. Those skilled in the art will appreciate that the present invention may also be implemented on platforms and operating systems other than those mentioned.

[0024] The hardware environment set forth in **FIG. 1A** includes a graphic acceleration integrated circuit which offloads graphics processing from the central processing unit **110**. In one embodiment of the present invention, the graphics acceleration integrated circuit may include, at least in part, a field programmable gate array (FPGA) device. Use of such device provides flexibility in functionality, while maintaining high processing speeds.

[0025] Examples of such FPGA devices include the XC2000™ and XC3000™ families of FPGA devices introduced by Xilinx, Inc. of San Jose, Calif. The architectures of these devices are exemplified in U.S. Pat. Nos. 4,642,487; 4,706,216; 4,713,557; and 4,758,985; each of which is originally assigned to Xilinx, Inc. and which are herein incorporated by reference for all purposes. It should be noted, however, that FPGA's of any type may be employed in the context of the prese nt invention. An FPGA device can be characterized as an integrated circuit that has four major features as follows.

[0026] (1) A user-accessible, configuration-defining memory means, such as SRAM, PROM, EPROM, EEPROM, anti-fused, fused, or other, is provided in the FPGA device so as to be at least once-programmable by device users for defining user-provided configuration instructions. Static Random Access Memory or SRAM is of course, a form of reprogrammable memory that can be differently programmed many times. Electrically Erasable and reProgrammable ROM or EEPROM is an example of nonvolatile reprogrammable memory. The configuration-defining memory of an FPGA device can be formed of mixture of different kinds of memory elements if desired (e.g., SRAM and EEPROM) although this is not a popular approach.

[0027] (2) Input/Output Blocks (IOB's) are provided for interconnecting other internal circuit components of the FPGA device with external circuitry. The IOB's' may have fixed configurations or they may be configurable in accordance with user-provided configuration instructions stored in the configuration-defining memory means.

[0028] (3) Configurable Logic Blocks (CLB's) are provided for carrying out user-programmed logic functions as defined by user-provided configuration instructions stored in the configuration-defining memory means.

[0029] Typically, each of the many CLB's of an FPGA has at least one lookup table (LUT) that is user-configurable to define any desired truth table,—to the extent allowed by the address space of the LUT. Each CLB may have other resources such as LUT input signal pre-processing resources and LUT output signal post-processing resources. Although the term 'CLB' was adopted by early pioneers of FPGA technology, it is not uncommon to see other names being given to the repeated portion of the FPGA that carries out user-programmed logic functions. The term, 'LAB' is used for example in U.S. Pat. No. 5,260,611 to refer to a repeated unit having a 4-input LUT.

[0030] (4) An interconnect network is provided for carrying signal traffic within the FPGA device between various CLB's and/or between various IOB's and/or between various IOB's and CLB's. At least part of the interconnect network is typically configurable so as to allow for programmably-defined routing of signals between various CLB's and/or IOB's in accordance with user-defined routing instructions stored in the configuration-defining memory means.

[0031] In some instances, FPGA devices may additionally include embedded volatile memory for serving as scratchpad memory for the CLB's or as FIFO or LIFO circuitry. The embedded volatile memory may be fairly sizable and can have 1 million or more storage bits in addition to the storage bits of the device's configuration memory.

[0032] Modern FPGA's tend to be fairly complex. They typically offer a large spectrum of user-configurable options with respect to how each of many CLB's should be configured, how each of many interconnect resources should be configured, and/or how each of many IOB's should be configured. This means that there can be thousands or millions of configurable bits that may need to be individually set or cleared during configuration of each FPGA device.

[0033] Rather than determining with pencil and paper how each of the configurable resources of an FPGA device should be programmed, it is common practice to employ a computer and appropriate FPGA-configuring software to automatically generate the configuration instruction signals that will be supplied to, and that will ultimately cause an unprogrammed FPGA to implement a specific design. (The configuration instruction signals may also define an initial state for the implemented design, that is, initial set and reset states for embedded flip flops and/or embedded scratchpad memory cells.)

[0034] The number of logic bits that are used for defining the configuration instructions of a given FPGA device tends to be fairly large (e.g., 1 Megabits or more) and usually grows with the size and complexity of the target FPGA. Time spent in loading configuration instructions and verifying that the instructions have been correctly loaded can become significant, particularly when such loading is carried out in the field.

[0035] For many reasons, it is often desirable to have in-system reprogramming capabilities so that reconfiguration of FPGA's can be carried out in the field.

[0036] FPGA devices that have configuration memories of the reprogrammable kind are, at least in theory, 'in-system programmable' (ISP). This means no more than that a possibility exists for changing the configuration instructions within the FPGA device while the FPGA device is 'in-system' because the configuration memory is inherently reprogrammable. The term, 'in-system' as used herein indicates that the FPGA device remains connected to an application-specific printed circuit board or to another form of end-use system during reprogramming. The end-use system is of course, one which contains the FPGA device and for which the FPGA device is to-be at least once configured to operate within in accordance with predefined, end-use or 'in the field' application specifications.

[0037] The possibility of reconfiguring such inherently reprogrammable FPGA's does not mean that configuration changes can always be made with any end-use system. Nor does it mean that, where in-system reprogramming is possible, that reconfiguration of the FPGA can be made in timely fashion or convenient fashion from the perspective of the end-use system or its users. (Users of the end-use system can be located either locally or remotely relative to the end-use system.)

[0038] Although there may be many instances in which it is desirable to alter a pre-existing configuration of an 'in the field' FPGA (with the alteration commands coming either from a remote site or from the local site of the FPGA), there are certain practical considerations that may make such in-system reprogrammability of FPGA's more difficult than first apparent (that is, when conventional techniques for FPGA reconfiguration are followed).

[0039] A popular class of FPGA integrated circuits (IC's) relies on volatile memory technologies such as SRAM (static random access memory) for implementing on-chip configuration memory cells. The popularity of such volatile memory technologies is owed primarily to the inherent reprogrammability of the memory over a device lifetime that can include an essentially unlimited number of reprogramming cycles.

[0040] There is a price to be paid for these advantageous features, however. The price is the inherent volatility of the configuration data as stored in the FPGA device. Each time power to the FPGA device is shut off, the volatile configuration memory cells lose their configuration data. Other events may also cause corruption or loss of data from volatile memory cells within the FPGA device.

[0041] Some form of configuration restoration means is needed to restore the lost data when power is shut off and then re-applied to the FPGA or when another like event calls for configuration restoration (e.g., corruption of state data within scratchpad memory).

[0042] The configuration restoration means can take many forms. If the FPGA device resides in a relatively large system that has a magnetic or optical or opto-magnetic form of nonvolatile memory (e.g., a hard magnetic disk)—and the latency of powering up such a optical/magnetic device and/or of loading configuration instructions from such an optical/magnetic form of nonvolatile memory can be tolerated—then the optical/magnetic memory device can be used as a nonvolatile configuration restoration means that redundantly stores the configuration data and is used to reload the same into the system's FPGA device(s) during power-up operations (and/or other restoration cycles).

[0043] On the other hand, if the FPGA device(s) resides in a relatively small system that does not have such optical/magnetic devices, and/or if the latency of loading configuration memory data from such an optical/magnetic device is not tolerable, then a smaller and/or faster configuration restoration means may be called for.

[0044] Many end-use systems such as cable-TV set tops, satellite receiver boxes, and communications switching boxes are constrained by prespecified design limitations on physical size and/or power-up timing and/or security provisions and/or other provisions such that they cannot rely on magnetic or optical technologies (or on network/satellite downloads) for performing configuration restoration. Their designs instead call for a relatively small and fast acting, non-volatile memory device (such as a securely-packaged EPROM IC), for performing the configuration restoration function. The small/fast device is expected to satisfy application-specific criteria such as: (1) being securely retained within the end-use system; (2) being able to store FPGA configuration data during prolonged power outage periods; and (3) being able to quickly and automatically re-load the configuration instructions back into the volatile configuration memory (SRAM) of the FPGA device each time power is turned back on or another event calls for configuration restoration.

[0045] The term 'CROP device' will be used herein to refer in a general way to this form of compact, nonvolatile, and fast-acting device that performs 'Configuration-Restoring On Power-up' services for an associated FPGA device.

[0046] Unlike its supported, volatilely reprogrammable FPGA device, the corresponding CROP device is not volatile, and it is generally not 'in-system programmable'. Instead, the CROP device is generally of a completely nonprogrammable type such as exemplified by mask-programmed ROM IC's or by once-only programmable, fuse-based PROM IC's. Examples of such CROP devices include a product family that the Xilinx company provides under the designation 'Serial Configuration PROMs' and under the trade name, XC1700D.TM. These serial CROP devices employ one-time programmable PROM (Programmable Read Only Memory) cells for storing configuration instructions in nonvolatile fashion.

[0047] A preferred embodiment is written using Handel-C. Handel-C is a programming language marketed by Celoxica

Limited. Handel-C is a programming language that enables a software or hardware engineer to target directly FPGAs (Field Programmable Gate Arrays) in a similar fashion to classical microprocessor cross-compiler development tools, without recourse to a Hardware Description Language. Thereby allowing the designer to directly realize the raw real-time computing capability of the FPGA.

[0048] Handel-C is designed to enable the compilation of programs into synchronous hardware; it is aimed at compiling high level algorithms directly into gate level hardware.

[0049] The Handel-C syntax is based on that of conventional C so programmers familiar with conventional C will recognize almost all the constructs in the Handel-C language.

[0050] Sequential programs can be written in Handel-C just as in conventional C but to gain the most benefit in performance from the target hardware its inherent parallelism must be exploited.

[0051] Handel-C includes parallel constructs that provide the means for the programmer to exploit this benefit in his applications. The compiler compiles and optimizes Handel-C source code into a file suitable for simulation or a net list which can be placed and routed on a real FPGA.

[0052] More information regarding the Handel-C programming language may be found in "EMBEDDED SOLUTIONS Handel-C Language Reference Manual: Version 3,""EMBEDDED SOLUTIONS Handel-C User Manual: Version 3.0,""EMBEDDED SOLUTIONS Handel-C Interfacing to other language code blocks: Version 3.0," and "EMBEDDED SOLUTIONS Handel-C Preprocessor Reference Manual: Version 2.1," each authored by Rachel Ganz, and published by Embedded Solutions Limited, and which are each incorporated herein by reference in their entirety. Additional information may be found in a co-pending application entitled "SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR INTERFACE CONSTRUCTS IN A PROGRAMMING LANGUAGE CAPABLE OF PROGRAMMING HARDWARE ARCHITECTURES" which was filed under attorney docket number EMB1P041, and which is incorporated herein by reference in its entirety.

[0053] Another embodiment of the present invention may be written at least in part using JAVA, C, and the C++ language and utilize object oriented programming methodology. Object oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design and development, various software solutions require adaptation to make use of the benefits of OOP. A need exists for these principles of OOP to be applied to a messaging interface of an electronic messaging system such that a set of OOP classes and objects for the messaging interface can be provided.

[0054] OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the program. An object is a software package that contains both data and a collection of related structures and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

[0055] In general, OOP components are reusable software modules which present an interface that conforms to an object model and which are accessed at run-time through a component integration architecture. A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each other's capabilities or functions. This is generally done by assuming a common component object model on which to build the architecture. It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

[0056] OOP allows the programmer to create an object that is a part of another object. For example, the object representing a piston engine is said to have a composition-relationship with the object representing a piston. In reality, a piston engine comprises a piston, valves and many other components; the fact that a piston is an element of a piston engine can be logically and semantically represented in OOP by two objects.

[0057] OOP also allows creation of an object that "depends from" another object. If there are two objects, one representing a piston engine and the other representing a piston engine wherein the piston is made of ceramic, then the relationship between the two objects is not that of composition. A ceramic piston engine does not make up a piston engine. Rather it is merely one kind of piston engine that has one more limitation than the piston engine; its piston is made of ceramic. In this case, the object representing the ceramic piston engine is called a derived object, and it inherits all of the aspects of the object representing the piston engine and adds further limitation or detail to it. The object representing the ceramic piston engine "depends from" the object representing the piston engine. The relationship between these objects is called inheritance.

[0058] When the object or class representing the ceramic piston engine inherits all of the aspects of the objects representing the piston engine, it inherits the thermal characteristics of a standard piston defined in the piston engine class. However, the ceramic piston engine object overrides these ceramic specific thermal characteristics, which are typically different from those associated with a metal piston. It skips over the original and uses new functions related to ceramic pistons. Different kinds of piston engines have different characteristics, but may have the same underlying functions associated with it (e.g., how many pistons in the engine, ignition sequences, lubrication, etc.). To access each of these functions in any piston engine object, a programmer would call the same functions with the same names, but each type of piston engine may have different/overriding implementations of functions behind the same name. This ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

[0059] With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. In fact, one's logical perception of the reality is the only limit on determining the kinds of things that can become objects in object-oriented software. Some typical categories are as follows:

[0060] Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control system.

[0061] Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.

[0062] An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.

[0063] An object can represent user-defined data types such as time, angles, and complex numbers, or points on the plane.

[0064] With this enormous capability of an object to represent just about any logically separable matters, OOP allows the software developer to design and implement a computer program that is a model of some aspects of reality, whether that reality is a physical entity, a process, a system, or a composition of matter. Since the object can represent anything, the software developer can create an object which can be used as a component in a larger software project in the future.

[0065] If 90% of a new OOP software program consists of proven, existing components made from preexisting reusable objects, then only the remaining 10% of the new software project has to be written and tested from scratch. Since 90% already came from an inventory of extensively tested reusable objects, the potential domain from which an error could originate is 10% of the program. As a result, OOP enables software developers to build objects out of other, previously built objects.

[0066] This process closely resembles complex machinery being built out of assemblies and sub-assemblies. OOP technology, therefore, makes software engineering more like hardware engineering in that software is built from existing components, which are available to the developer as objects. All this adds up to an improved quality of the software as well as an increased speed of its development.

[0067] Programming languages are beginning to fully support the OOP principles, such as encapsulation, inheritance, polymorphism, and composition-relationship. With the advent of the C++ language, many commercial software developers have embraced OOP. C++ is an OOP language that offers a fast, machine -executable code. Furthermore, C++ is suitable for both commercial-application and systems-programming projects. For now, C++ appears to be the most popular choice among many OOP programmers, but there is a host of other OOP languages, such as Smalltalk, Common Lisp Object System (CLOS), and Eiffel. Additionally, OOP capabilities are being added to more traditional popular computer programming languages such as Pascal.

[0068] The benefits of object classes can be summarized, as follows:

[0069] Objects and their corresponding classes break down complex programming problems into many smaller, simpler problems.

[0070] Encapsulation enforces data abstraction through the organization of data into small, independent objects that can communicate with each other. Encapsulation protects the data in an object from accidental damage, but allows other objects to interact with that data by calling the object's member functions and structures.

[0071] Subclassing and inheritance make it possible to extend and modify objects through deriving new kinds of objects from the standard classes available in the system. Thus, new capabilities are created without having to start from scratch.

[0072] Polymorphism and multiple inheritance make it possible for different programmers to mix and match characteristics of many different classes and create specialized objects that can still work with related objects in predictable ways.

[0073] Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.

[0074] Libraries of reusable classes are useful in many situations, but they also have some limitations. For example:

[0075] Complexity. In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.

[0076] Flow of control. A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The programmer has to decide which functions to call at what times for which kinds of objects.

[0077] Duplication of effort. Although class libraries allow programmers to use and reuse many small pieces of code, each programmer puts those pieces together in a different way. Two different programmers can use the same set of class libraries to write two programs that do exactly the same thing but whose internal structure (i.e., design) may be quite different, depending on hundreds of small decisions each programmer makes along the way. Inevitably, similar pieces of code end up doing similar things in slightly different ways and do not work as well together as they should.

[0078] Class libraries are very flexible. As programs grow more complex, more programmers are forced to reinvent basic solutions to basic problems over and over again. A relatively new extension of the class library concept is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus, windows, dialog boxes, and other standard user interface elements for personal computers.

[0079] Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the flow of control. This was appropriate for printing out paychecks, calculating a mathematical table, or solving other problems with a program that executed in just one way.

[0080] The development of graphical user interfaces began to turn this procedural programming arrangement inside out. These interfaces allow the user, rather than program logic, to drive the program and decide when certain actions should be performed. Today, most personal computer software accomplishes this by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code acc ording to actions that the user performs. The programmer no longer determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing cont rol in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

[0081] Even event loop programs require programmers to write a lot of code that should not need to be written separately for every application. The concept of an application framework carries the event loop concept further. Instead of dealing with all the nuts and bolts of constructing basic menus, windows, and dialog boxes and then making these things all work together, programmers using application frameworks start with working application code and basic user interface elements in place. Subsequently, they build from there by replacing some of the generic capabilities of the framework with the specific capabilities of the intended application.

[0082] Application frameworks reduce the total amount of code that a programmer has to write from scratch. However, because the framework is really a generic application that displays windows, supports copy and paste, and so on, the programmer can also relinquish control to a greater degree than event loop programs permit. The framework code takes care of almost all event handling and flow of control, and the programmer's code is called only when the framework needs it (e.g., to create or manipulate a proprietary data structure).

[0083] A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

[0084] Thus, as is explained above, a framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., for menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

[0085] There are three main differences between frameworks and class libraries:

[0086] Behavior versus protocol. Class libraries are es sentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.

[0087] Call versus override. With a class library, the code the programmer instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together.

[0088] Implementation versus design. With class libraries, programmers reuse only implementations, whereas with frameworks, they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

[0089] Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved. A preferred embodiment of the invention utilizes HyperText Markup Language (HTML) to implement documents on the Internet together with a general-purpose secure communication protocol for a transport medium between the client and the Newco. HTTP or other protocols could be readily substituted for HTML without undue experimentation. Information on these products is available in T. Bemers-Lee, D. Connoly, "RFC 1866: Hypertext Markup Language-2.0" (Nov. 1995); and R. Fielding, H, Frystyk, T. Berners-Lee, J. Gettys and J. C. Mogul, "Hypertext Transfer Protocol—HTTP/1.1: HTTP Working Group Internet Draft" (May 2, 1996). HTML is a simple data format used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML has been in use by the World-Wide Web global information initiative since 1990. HTML is an application of

ISO Standard 8879; 1986 Information Processing Text and Office Systems; Standard Generalized Markup Language (SGML).

[0090] To date, Web development tools have been limited in their ability to create dynamic Web applications which span from client to server and interoperate with existing computing resources. Until recently, HTML has been the dominant technology used in development of Web-based solutions. However, HTML has proven to be inadequate in the following areas:

[0091] Poor performance;

[0092] Restricted user interface capabilities;

[0093] Can only produce static Web pages;

[0094] Lack of interoperability with existing applications and data; and

[0095] Inability to scale.

[0096] Sun Microsystem's Java language solves many of the client-side problems by:

[0097] Improving performance on the client side;

[0098] Enabling the creation of dynamic, real-time Web applications; and

[0099] Providing the ability to create a wide variety of user interface components.

[0100] With Java, developers can create robust User Interface (UI) components. Custom "widgets" (e.g., real-time stock tickers, animated icons, etc.) can be created, and client-side performance is improved. Unlike HTML, Java supports the notion of client-side validation, offloading appropriate processing onto the client for improved performance. Dynamic, real-time Web pages can be created. Using the above-mentioned custom UI components, dynamic Web pages can also be created.

[0101] Sun's Java language has emerged as an industry-recognized language for "programming the Internet." Sun defines Java as: "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language. Java supports programming for the Internet in the form of platform-independent Java applets." Java applets are small, specialized applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g., simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible s browser (e.g., Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on C++. Sun's Java literature states that Java is basically, "C++ with extensions from Objective C for more dynamic method resolution."

[0102] Another technology that provides similar function to JAVA is provided by Microsoft and ActiveX Technologies, to give developers and Web designers wherewithal to build dynamic content for the Internet and personal computers. ActiveX includes tools for developing animation, 3-D virtual reality, video and other multimedia content. The tools use Internet standards, work on multiple platforms, and are being supported by over 100 companies. The group's building blocks are called ActiveX Controls, small, fast components that enable developers to embed parts of software in hypertext markup language (HTML) pages. ActiveX Controls work with a variety of programming languages including Microsoft Visual C++, Borland Delphi, Microsoft Visual Basic programming system and, in the future, Microsoft's development tool for Java, code named "Jakarta." ActiveX Technologies also includes ActiveX Server Framework, allowing developers to create server applications. One of ordinary skill in the art readily recognizes that ActiveX could be substituted for JAVA without undue experimentation to practice the invention.

[0103] In the one embodiment of the present invention, the aforementioned Handel-C programming language is capable of executing macros for working in conjunction with a frame buffer procedure for drawing graphics to the screen. **FIG. 2** illustrates a method **200** for enhanced handling of graphics data in a frame buffer. Initially, in operation **202**, graphics data is received. Thereafter, in operation **204**, operations are performed on the graphics data utilizing a predetermined set of macros in combination with a frame buffer procedure. The graphics data is written into a frame buffer for being displayed on a screen. Note operation **206**.

[0104] The Handel-C compiler passes source code through a standard C-programming language preprocessor before compilation allowing the use of "#define" to define constants and macros in the usual manner. There are some limitations to this approach. Since the preprocessor can only perform textual substitution, some useful macro constructs cannot be expressed. For example, there is no way to create recursive macros using the preprocessor. Handel-C provides additional macro support to allow more powerful macros to be defined (for example, recursive macro expressions). In addition, Handel-C supports shared macros to generate one piece of hardware which is shared by a number of parts of the overall program similar to the way that procedures allow conventional C to share one piece of code between many parts of a conventional program.

[0105] By this design, the present invention has the following features:

[0106] Compliments frame buffering.

[0107] Graphics macros for circles, squares, and triangles.

[0108] Line drawing macro.

[0109] Icon drawing macros for 24-bit and monochrome bitmaps.

[0110] **FIG. 3** illustrates the various external files **300** may be may be needed by the present invention. Note Appendix A. **FIG. 4** illustrates internal and external macros, **400** and **402**, associated with the present invention. Such macros are defined in the file "Gfx.h." which is well known in the Handel-C programming language. Such macros are split into two sections: internal macros which are specific to Gfx.h. External macros should be called by a main program.

[0111] One macro, gfx_printShapeProcess, is run as one of the main parallel processes of a graphics application. It consists of four main sections:

[0112] Wait Stage

[0113] When no shape drawing calls are made, print-_shape_run, the shape drawing flag, is set to off and the process waits for initiation.

[0114] Draw Rectangle

[0115] The present macro is run in parallel with the circle and triangle drawing stages. Pixels are drawn directly into the frame buffer at the relevant place, in rows. When each row is completed, the y variable is incremented and the x variable is reset to its original position for the start of the next row. If one of the other shape stages is running, a delay is performed.

[0116] Draw Circle

[0117] The present macro is run in parallel with the rectangle and triangle drawing stages. The program steps through a square of width twice the radius, centered at the same point as the circle. Each pixel is tested for satisfaction of Pythagoras' Theorem. If the inequality of Table 1 is satisfied, the pixel (px, py) is inside the circle and the color is written to the frame buffer.

TABLE 1

$(\text{centre\_x} - px)^2 + (\text{centre\_y} - py)^2 \leqq \text{radius}^2$

[0118] Draw Triangle

[0119] The present macro is run in parallel with draw circle and draw rectangle stages. The first step is to sort the coordinates into ascending order of y-values. Once done, the macro interpolates the border lines pixel by pixel, drawing horizontal lines to fill the triangle as below. The interpolate line function is just a variation on the macro, gfx_drawLine, except it calculates one pixel at a time. **FIG. 5** illustrates the Draw Triangle Macro **500**, in accordance with one embodiment of the present invention.

[0120] Another macro, the gfx_drawLine macro, draws a straight line to the frame buffer in the set color. The drawLine_d variable is set, and converges to zero by subtracting fractions of the x distance or y distance. The sign of drawLine_d determines whether to increment or decrement y or x. The relevant pixel is then written into the frame buffer.

[0121] gfx_draw1BitIcon draws a monochrome bitmap from a rom to the screen buffer. The programmer is able to set the two colors to use for the icon, if the bitmap reads one at a pixel, color1 is used, otherwise color2 is used.

[0122] gfx_draw24BitIcon draws a 24-bit (with 8-bit colors values RGB) to the frame buffer. As the frame buffer uses 15-bit color, the 3 least significant bits on each color value are dropped before drawing it to the screen.

[0123] gfx_clearScreen clears the screen to one color by initiating a draw of a screen-sized rectangle into the frame buffer. It sets the parameters so that the gfx_printShapeProcess macro can draw the rectangle. The other shape macros, gfx_drawRect, gfx_drawTriangle, andgfx_drawCircle operate in the same manner-setting the relevant parameters for gfx_drawShapeProcess to process a draw to the screen.

[0124] With the exceptions of icon drawing and line drawing processes, the macro gfx_printShapeProcess contains all the shape drawing code for the shape macros. This may be called from the main par loop. The shape drawing macros simply set various parameters to initiate the required function and these must be called every time a shape is to be drawn. **FIG. 6** illustrates a chart **600** showing speeds with which the various macros may operate.

APPENDIX A

```
/****************************************************************
* File      :     lcddriver.h                                    *
#include "syncgen.h"
unsigned 10 sx, sy;
//adjust a 15 bit colour to an 18 bit colour
macro expr video_adjust(pixel) =   ( (unsigned 6) (pixel[14:10]@0)) @
                                   ( (unsigned 6) (pixel[9:5] @0)) @
                                   ( (unsigned 6) (pixel[4:0] @0));
macro expr buffer_adjust(pixel) = 0@(pixel[17:13])@(pixel[11:7])
@(pixel[5:1]);
macro expr RGB(r, g, b) = ((unsigned 6)r)@((unsigned 6)g)@((unsigned
6)b);
macro expr black = RGB(0,0,0);
macro expr white = RGB(63,63,63);
macro expr red = RGB(63,0,0);
macro expr blue = RGB(0,0,63);
macro expr green = RGB(0,63,0);
macro expr grey = RGB(17,17,17);
macro expr purple = RGB(0, 63, 63);
macro expr cyan = RGB(0, 63, 63);
#define BRIGHT_WIDTH 3
unsigned BRIGHT_WIDTH brightness_level = 4;
macro proc brightness_process()
{
  unsigned 1 bright_pin;
  unsigned BRIGHT_WIDTH wait;
  interface bus_out() br(bright_pin) with {data = {"D11"}};
  while (1)
  {
    while (wait!=brightness_level)
    {
      par
      {
        bright_pin = 1;
        wait++;
      }
    }
    do
    {
      par
      {
        bright_pin = 0;
        wait++;
      }
    }while (wait!=0);
  }
}
macro proc brightness_set (value)
{
  brightness_level = value;
}
/*
 lcd_driver
 run the sync and pixel generators
 define the external interfaces
 needs sx and sy scan positions defined as globals
*/
macro proc lcd_driver3(video1, video2, video3, trans_clr1,
trans_clr2, lcd_enable)
```

APPENDIX A-continued

```
{
/* interface pin definitions */
 unsigned 1 vs, hs, de;
#ifdef SIMULATE
 macro expr pad(col) = col[17:12] @ (unsigned 2)0 @ col[11:6] @
(unsigned 2)0 @ col[5:0] @ (unsigned 2)0;
 interface bus_out() lcd_pix( video1 == trans_clr1 ? (video2 ==
trans_clr2 ? pad(video3) : pad(video2) : pad(video1)) with
lcd_data_pins;
#else
 interface bus_out() lcd_pix(video1 == trans_clr1 ? (video2 ==
trans_clr2 ? video3 : video2) : video1) with lcd_data_pins;
#endif
 par
 {
  brightness_process();
  SyncGen(sx, sy, vs, hs, de, lcd_enable);
 }
}
macro proc lcd_driver2(video1, video2, trans_clr1, lcd_enable)
{
/* interface pin definitions */
  unsigned 1 vs, hs, de;
#ifdef SIMULATE
 macro expr pad(col) = col[17:12] @ (unsigned 2)0 @ col[11:6] @
(unsigned 2)0 @ col[5:0] @ (unsigned 2)0;
 interface bus_out() lcd_pix( video1 == trans_clr1 ? pad(video2)
: pad(video1)) with lcd_data_pins;
#else
 interface bus_out() lcd_pix(video1 == trans_clr1 ? video2 :
video1) with lcd_data_pins;
#endif
 par
 {
  brightness_process();
  SyncGen(sx, sy, vs, hs, de, lcd_enable);
 }
}
macro proc lcd_driver(video, lcd_enable)
{
/* interface pin definitions */
  unsigned 1 vs, hs, de;
#ifdef SIMULATE
 macro expr pad(col) = col[17:12] @ (unsigned 2)0 @ col[11:6] @
(unsigned 2)0 @ col[5:0] @ (unsigned 2)0;
 interface bus_out() lcd_pix( pad(video)) with lcd_data_pins;
#else
 interface bus_out() lcd_pix(video) with lcd_data_pins;
#endif
 par
 {
  brightness_process ();
  SyncGen(sx, sy, vs, hs, de, lcd_enable);
 }
}
```

[0125] While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1) A method for enhanced handling of graphics data in a frame buffer, comprising the steps of:

(a) receiving graphics data;

(b) performing operations on the graphics data utilizing a predetermined set of macros in combination with a frame buffer procedure; and

(c) writing the graphics data into a frame buffer for being displayed on a screen.

2. A method as recited in claim 1, wherein the operations include a wait operation.

3. A method as recited in claim 1, wherein the operations are selected from the group consisting of a draw rectangle operation, draw circle operation, and draw triangle operation.

4. A method as recited in claim 1, wherein the macros include a function identifier and parameters associated therewith.

5. A method as recited in claim 1, wherein the operations are executed at speeds between 15.7 MHz to 28.9 MHz.

6) A computer program product for enhanced handling of graphics data in a frame buffer, comprising:

(a) computer code for receiving graphics data;

(b) computer code for performing operations on the graphics data utilizing a predetermined set of macros in combination with a frame buffer procedure; and

(c) computer code for writing the graphics data into a frame buffer for being displayed on a screen.

7. A computer program product as recited in claim 6, wherein the operations include a wait operation.

8. A computer program product as recited in claim 6, wherein the operations are selected from the group consisting of a draw rectangle operation, draw circle operation, and draw triangle operation.

9. A computer program product as recited in claim 6, wherein the macros include a function identifier and parameters associated therewith.

10. A computer program product as recited in claim 6, wherein the operations are executed at speeds between 15.7 MHz to 28.9 MHz.

11) A system for enhanced handling of graphics data in a frame buffer, comprising:

(a) logic for receiving graphics data;

(b) logic for performing operations on the graphics data utilizing a predetermined set of macros in combination with a frame buffer procedure; and

(c) logic for writing the graphics data into a frame buffer for being displayed on a screen.

12. A system as recited in claim 11, wherein the operations include a wait operation.

13. A system as recited in claim 11, wherein the operations are selected from the group consisting of a draw rectangle operation, draw circle operation, and draw triangle operation.

14. A system as recited in claim 11, wherein the macros include a function identifier and parameters associated therewith.

15. A system as recited in claim 11, wherein the operations are executed at speeds between 15.7 MHz to 28.9 MHz.

* * * * *