A DSP-Based Three-Dimensional Graphics System

APPLICATION REPORT: SPRA402

Nat Seshan
Digital Signal Processor Products
Semiconductor Group
Texas Instruments

Digital Signal Processing Solutions

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

An All-Digital Automatic Gain Control

Abstract

One way to improve the performance of a graphics system is to place an advanced digital signal processor between the host processor and the video controller. This book, the author's bachelor's thesis at MIT, discusses the suitability of the TMS320C30 Digital Signal Processor to that task.

The book begins with a listing of the factors that make the TMS320C30 the digital signal processor of choice. It is followed by a real world scenario that shows the type of need this type of system could meet and how this thesis could be tested. Beginning with an example of a mechanical engineer's need to create, store and adjust 3D objects and render the images on a video display, the books explores the instructional model of how a graphics system can be designed using an advanced digital signal processor.

Topics covered include:

- An overview of the implementation
- Representation of graphics elements
- DPS command execution
- The rendering subsystem (including information on the TMS34010 Graphics System Processor and the TMS34010 Software development board)
- Possible improvements
- References

There are several supporting graphics and tables including:

- An example of a full scale graphics pipeline

- A simple three-processor graphics pipeline
- Modified TMS34010 software development board block diagram
- Transformation matrices and equations
- Comparison of Intel and TMS320C30 32-bit floating-point formats
- TMS34010 signals controlling host port interface
- TMS34010 host control register fields
- Mapping of TMS34010 host control register fields

Product Support

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to dsph@ti.com. Questions receive prompt attention and are usually answered within one business day.

This application report is based on the author's bachelor's thesis at the Massachusetts Institute of Technology.

The placement of a high-performance computational engine, such as an advanced digital signal processor, between the host processor and the video controller in a graphics system can improve performance tremendously. Several factors make the Texas Instruments TMS320C30 Digital Signal Processor well-suited to this task:

- 32-bit floating point arithmetic provides both high-resolution and large dynamic range in calculation.
- Single-cycle, 60-ns instruction execution and parallel bus access greatly improve system throughput.
- A hardware single-cycle multiplier facilitates the matrix arithmetic, which is frequently required in 3D graphics.
- The ease of programmability allows the design of flexible and expandable systems.
- Software tools, such as simulators[1], assembler/linkers[2], and high- level language debuggers/compilers[3], decrease product development time.
- In-circuit scan-path emulators[4], decrease hardware prototyping and debugging time.
- The use of a standard device lowers the overall system cost.

With the use of the TMS320C30, the host processor can request higher-level commands of the rest of the system. Instead of issuing requests for line-draws or screen clears, it can, for example, request that a 3D object be rotated 90 degrees and then be redrawn. In addition, a rendering element (usually a video controller or graphics system processor) can devote its resources solely to screen management rather than doing some portion of the computationally intensive processing. The following pages provide a description of how a 3D graphics system used the TMS320C30 to compute object transformations.

The digital signal processor resides on the TMS320C30 Application Board (C30AB) designed for the IBM PC/AT or compatible. The PC's 80x86 acts as the host processor and communicates to the C30AB through an 8-bit bus slot. Also resident on the bus is a Texas Instruments TMS34010 Software Development Board (SDB)[5,6]. The SDB contains a TMS34010 Graphics System Processor (GSP) [7], which manages the screen memory and drives the video display. Overall, this system is meant to serve as an instructional model of how a graphics system can be designed using an advanced digital signal processor.

The Potential for Graphics Pipelines

A mechanical engineer for an automobile manufacturer wants to design a robot arm for plant automation. Before building a prototype machine, he wishes to compare the ways in which various designs can pick up and assemble components. To do this,the engineer needs a CAD system capable of creating, storing, and adjusting representations of 3D objects and then rendering the images on a video display. The CAD system has four basic aspects:

1) A user interface for command entry.
2) A data management system to store objects and their screen representations.
3) One or more computational engines to perform high-speed calculations for applications such as transformations, clipping, lighting/shading, and fractal graphics.

4) A rendering engine to control the video memory and to drive the video display.

These four tasks are common to many graphics systems, whether they be intended for CAD/CAM, fractal graphics, heads-up displays in fighter aircraft, or Postscript printer control. If one or more processors are assigned to each function, the resulting pipeline will achieve greatly improved system throughput.

In a single-processor system, the CPU is directly responsible for all computations. It must write to video memory, perform all necessary computations, interface to the user, and manage all data storage and recovery. Although additions to the system, such as a video-memory controller or a floating-point coprocessor, may speed up the system, the CPU remains overly burdened as the only intelligent component of the system.

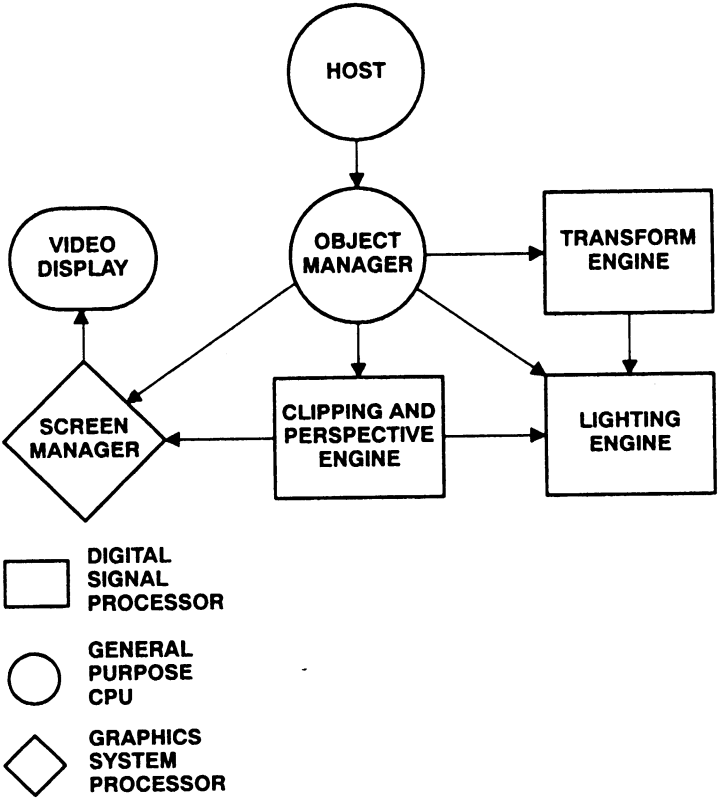
Independent Screen Management

A two-processor system can use a GSP to drive the CRT and to control the video memory. To control the display, the GSP either must interface to an analog monitor through a color palette or must directly drive a digital monitor. If the video memory is volatile, the processor needs a refresh controller that runs in parallel with other processor actions. Special hardware can be developed for screen clears and polygon fills. For flexibility of data representation, the processor should to be able to access pixels of varying bit-widths. At the instruction level, specialized operations could be created to speed pixel processing. Libraries of subroutines for windowing, drawing, and text management enable the rendering engine to execute higher-level commands. Overall, these features allow the CPU to send more powerful directives to the GSP.

A Multiprocessor Pipeline

Adding more links in the graphics pipeline can further relieve the CPU of burdensome tasks. Performance improvements result from each stage being optimized for a particular function. In addition, throughput increases with the number of stages. The pipeline may also contain multiple processors running in parallel at a particular stage to further improve the latency of that stage. Figure 1 shows a full-scale implementation of a graphics pipeline for 3D graphics.

Figure 1. A Full Scale Graphics Pipeline



In a large-scale graphics pipeline, the host processor runs the applications program. The user may be trying to use a CAD program, model the formation of galaxies, animate 3D objects, etc. The host runs these programs at the top level, provides the user interface, and communicates to all I/O devices, including mass storage systems. For numerically intensive applications it may be appropriate to have a digital signal processor as this host. For example, modeling the formation of galaxies requires numerical solutions to systems of differential equations. But even in such a case, it would be reasonable to have a more general-purpose CPU act as a user front end to the digital signal processor.

The purpose of the object manager is to communicate with the host by receiving data and transferring it to other processors in the system. It manages the global representation of all screen parameters and objects. A Reduced Instruction Set Computer (RISC) processor would be well-suited as either the host or the object manager because of its high-performance general-purpose architecture.

Because a DSP has a highly parallel architecture, a fast execution cycle time, an instruction set optimized for numerical processing, and several development tools, it would perform well as any of the computational stages in a graphics pipeline. For example, a DSP could act as a transform manager that calculates the new universal coordinates of globally stored objects according to rota-

tion, translation, and scaling commands from the object manager. Also, the DSP could act as a lighting manager that accepts parameters of environmental lighting settings from the object manager and applies them to the transformed objects. For example, the user may set ambient intensities as well as other sources of varying geometries, intensities, and colors. The lighting manager then applies these light sources to the surfaces of the objects, which may have varying degrees of specular or diffuse reflection, to compute the necessary shading.

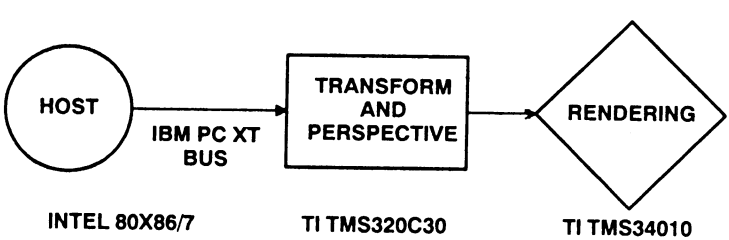
Although the perspective and clipping stage of the system is represented in Figure 1 by a single processing unit, the task may be further partitioned to several DSPs working in series. The perspective calculation takes viewing parameters from the object manager, such as direction of view, location of viewer, and zoom, and produces a two-dimensional projection for the screen. Objects that are too high, too low, or too far right or left can be clipped automatically because the resulting two-dimensional coordinates are off screen. However, clipping objects fully or partially obscured by other objects may require additional stages. Also, objects behind the viewer and those too far away for the user to recognize should be clipped appropriately.

Although digital signal processors are well-suited to be the computational stages of a graphics pipeline, a processor optimized to be a rendering engine might serve better to drive the video display and manage the video memory. Such a processor could also help with the clipping tasks described above. A z-buffer could hold the transformed z-coordinate of each pixel that is projected on to the x-y plane of the screen to facilitate hidden surface removal. A device such the Texas Instruments TMS34010 or the recently introduced TMS34020 could serve as the rendering engine in a full scale system. Both these processors have 32-bit general-purpose architectures with instruction sets and external memory interfaces optimized for graphics.

An Overview of This Implementation

The system shown in Figure 2 is not intended to be a marketable product. Rather, it is targeted toward those who have the intention of designing products in the graphics market. Firms having experience in graphics will be able to resolve the tougher issues of graphics system design without presentation of the described system. The system shown in this report illustrates an attractive option for designing a fast, reliable, portable graphics system with quick turn-around time.

Figure 2. A Simple Three-Processor Graphics Pipeline



One strength of this system is its complete use of standard, commercially available parts. In general, use of standard parts allows for faster design and manufacturing, as well as a more reliable, easier-to-support product. Even the three hardware subsystems can be found on the market:

1) The IBM PC compatible host
2) The TMS320C30 Application Board object manager and transform engine subsystem
3) The TMS34010 Software Development Board rendering subsystem

Another strength of this system is the complete use of portable software. Use of portable software often speeds design times because system software can be mostly debugged before the actual target hardware is available. All software for this system was written in Kernigan and Ritchie C. The command and rendering routine was first debugged on the PC and GSP with the intermediary stage removed. Once debugged, the computationally intensive portion of the software was ported to the DSP, which then assumed control of the GSP. The software on the TMS34010 SDB used many of the graphics routines in the TMS34010 Graphics/Math Library. These routines have been used in many other graphics systems using the TMS34010.

System Hardware

The IBM PC was chosen as the host because of its extensive support by TI development tools. In addition, a large amount of documentation is available concerning interfacing to the PC bus. The system described in this report is designed to run best on an 80386-based IBM PC compatible with an AT power supply and an 80387 floating-point coprocessor. However, either Intel 8086 or 80286 general-purpose microprocessors can also act as the host to the computational engine. The host computer sends commands to

- Load and delete objects
- Target an object for adjustment
- Adjust a particular object
- Recalculate the perspective or
- Redraw the screen.

The 80X87 floating-point coprocessor is not absolutely necessary but greatly improves the time to generate floating-point parameters for the next stage.

This graphics demonstration was the first application developed using the TMS320C30 Application Board (C30AB). Since that time, the C30AB has been included as a part of the XDS1000 emulation system for the TMS320C30 Digital Signal Processor. The TMS320C30's features include

- 60-ns single-cycle execution time (more than 33 MFLOPS)
- 2K x 32-bit dual-access RAM
- 4K x 32-bit dual-access ROM
- 64 x 32-bit instruction cache
- Two 32-bit external memory expansion buses
- Single-cycle floating-point multiply/accumulate
- Two external 32-bit memory ports

- On-chip DMA controller
- Zero-overhead loops and single-cycle branches
- Two on-chip timers and two serial ports
- Floating-point/integer and logical 32/40-bit ALU
- 16M-word memory space
- Register-based CPU
- Development tools, including a simulator, assembler/linker, optimizing C compiler, C-source debugger, and an in-circuit emulator/debugger
- On-chip scan-path emulation logic
- Low-power CMOS technology

The TMS320C30 executes commands from the 80X86 to transform objects, load objects into or delete objects from the system, and compute the projection of 3D objects on the 2D screen. When given a directive to draw the screen, it sends a command to the rendering engine to clear the current screen. Then, the TMS320C30 transfers lists of lines, points, and polygons for the next stage to render.

The TMS34010 Software Development Board (SDB) has been used in TMS34010 development support since 1987. It is configurable for a variety of monitors. The board supports the TMS34010 Graphics/Math Function Library [8] (a library of high-level routines callable from any C program). This board was slightly modified to receive commands from the C30AB as well as from the PC host. Program loaders, C compilers [9], assemblers, and C language standard I/O library support have been developed for this board, as well as for the C30AB. Both cards interface to an IBM PC through an 8-bit slot on the AT bus. The TMS34010 GSP on the SDB is an advanced high-performance CMOS 32-bit microprocessor optimized for graphics display systems. Its key features include:

- 160-ns instruction cycle time
- Fully programmable 32-bit general-purpose processor with a 128M-byte address range
- Pixel processing, X-Y addressing, and window clip/pick built into the instruction set
- Programmable pixel size with 16 boolean and 6 arithmetic pixel processing options (Raster-Ops)
- 31 general purpose 32-bit registers
- 256-byte LRU on-chip instruction cache
- Direct interfacing to both conventional DRAM and multiport video RAM
- Dedicated 8/16-bit host processor interface and HOLD/HLD interface
- Programmable CRT control (HSYNC, VSYNC, BLANK)
- Full line of hardware and software development tools, including a C compiler

The TMS34010 GSP receives commands from the TMS320C30, along with arrays of points, lines, and filled polygons to be drawn. It then uses library routines to render these images on the video display.

System Limitations

The system described here is an instructional system built in a limited development time. Aspects of the system could be optimized for speed and for memory usage. A high-speed 3D graphics system has many features that were not implemented.

This design is non-optimal in several ways. The C routines could be hand-coded to execute faster. A 32-bit host bus interface would allow word-at-a-time data transfers to the TMS320C30. The GSP could be interfaced to faster video memory. At the time of this writing, the TMS34020 second-generation graphics system processor is available. The entire TMS320C30 program could be configured to run from internal memory. Many of these optimizations were not realized because of the limited time available for developing the system.

Many operations that an advanced digital signal processor could easily perform were not designed into this system. These tasks include curved and textured surface generation, lighting, shading, and front and back clipping. For demonstrative purposes, only the endpoint transformation and perspective calculations were implemented.

Similarly, the capabilities of the GSP are clearly underutilized in this pipeline. The GSP is adept at managing multiple windows for display. It can also display text in various fonts. The presented system simply requires that the GSP manage a single graphics-only (no text) window.

Representation of Graphics Elements

Any graphics system must have a method of representing the image to be portrayed on the screen. This method requires a system that is able to store and display primitive elements. These elements could range in complexity from three coordinates describing a point to a set of parametric equations representing an irregular three-dimensional surface. However, simply defining a set of primitive drawing structures does not result in an adequate graphics data representation. The engineer designing the robot does not think of the system as several sheet-metal polygons welded together. He more likely conceives of the arm as a clamp attached to a hand, which, in turn, is attached to an arm, etc. A powerful graphics system must not only describe the primitives to be rendered on the CRT, but also how the primitives are organized or related.

Frames of reference play the central role in the organization of graphics primitives. Any set of graphics primitives rigid with respect to each other can be said to exist in the same, constant frame. When the primitives move, they move as a single unit and remain in the same orientation with respect to each other. In this system, any such set of primitives is called an object. The transformational state of any object is determined by three sets of three parameters each. These sets of the object correspond to the

- Translation
- Scale
- Rotation

Translation of an object within its frame simply amounts to moving all locations in that frame a specified distance along the x-, y-, and z-axes. Thus, each object must hold a set of translation factors, denoted in this system's software by **dx**, **dy**, and **dz** (See Listing 1 in the Appendix). Simi-

larly, **sx**, **sy**, and **sz** determine the scale of an object. These factors determine how many units of the untransformed object's coordinates are represented by one unit of the transformed object's coordinates. The three parameters shown in Appendix Listing 1 that represent all possible orientations of an object (**theta**, **phi**, and **omega**) are described in Table 1.

Table 1. Angles of Rotation

Angle	Axis Rotation is Around	Direction of Positive Rotation	Zero Value
θ	z	x to y	Positive x-axis
ω	x	y to z	Positive y-axis
ϕ	y	z to x	Positive z-axis

The Object Data Structure

Every object contains one or more sets of locations, which are referenced by the drawing primitives within the object. The **locnum** field of the object structure (see Listing 1) represents the number of locations available to be referenced by primitives within the object. This and other array sizes are kept for end points in For/Next-type loops and to allocate the appropriate space for the array contained within an object. Every **location** (see Appendix Listing 2) contains three floating-point numbers representing a coordinate in 3D space: **x**, **y**, and **z**. Their integer x-y locations on screen are also saved: **a**, **b**. To reference a location, a primitive needs only to know the index in the locs array. This allows many primitives to reference the same location.

Three different primitives were implemented to be rendered on the screen:

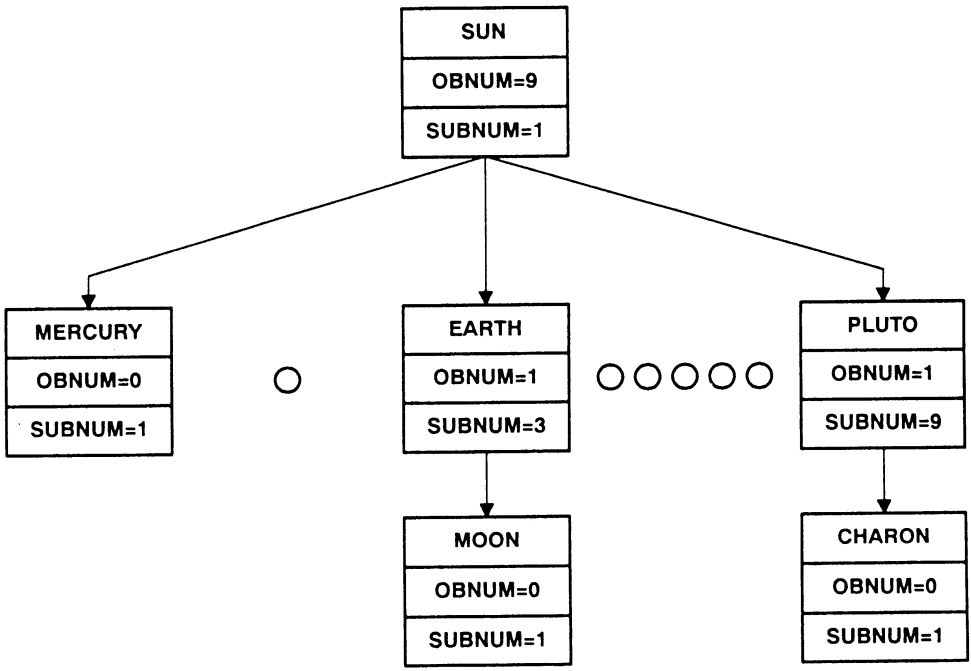
- Points
- Line segments
- Filled polygons

Points are rendered as single pixels on the screen. The **point** structure shown in Listing 3 of the Appendix contains the **color** to draw the point and the index to the location (**locn**) that is referenced by that point. The line structure in Listing 4 of the Appendix contains a **color** and two indices (**startlocn** and **endlocn**) to two end-points of the segment. Finally, the filled polygon shown in Listing 5 of the Appendix contains, in addition to the **color**, the number of vertices (**vertnum**) for the polygon, and a pointer (***vertlocn**) to an array of vertex location indices listed in the order in which they are connected). The last location in the vertex array is connected back to the first, closing the polygon.

Hierarchy

The final array contained within an object (the *parent object*) is a list of pointers to *child objects* defined with respect to the transformed frame of the parent. The number of potential internal objects, **MAXOB**, sets the static size of the array of pointers to child objects. (In this implementation, **MAXOB** = 10.) In addition, the parameter **obnum** keeps track of how many of these potential child objects are utilized. The final bookkeeping parameter is **subnum**. If **subnum** equals *n*, then the object was the *n*th object pointed to in its parent object's child-object array.

Figure 3. Hierarchical Representation of the Solar System



The solar system (Figure 3) represents a classical example of a hierarchical structure. The sun slowly revolves around the galaxy. Wherever the sun travels, the planets follow in the same frame. In turn, each planet may have satellites that revolve around them. The planet is defined with a certain offset (radius of orbit) from the sun, and the satellite is defined similarly with an offset from the planet. To describe the movement of the earth over a period of time, you need only to adjust for its revolution around the sun and the revolution of the moon around the earth. You do not need to describe the rotation of the moon around the sun because when a planet is moved, its satellites automatically move with it.

Transformation parameters are referenced to the frame of the object's parent. Thus, to fully describe a planet orbiting the sun, one must define an empty frame revolving about the sun at some offset, and then define a planet within that frame rotating about some axis. The levels of abstraction within this hierarchy give this data representation its power.

The flexibility of the **object** structure permits the system to model the viewer. The viewer is considered to be at the absolute origin of the system. At system initialization, the first object loaded is the universal object ***universe**. An appropriate choice for such an object would be a set of axes. The view is then adjusted by modifications to the parameters of the ***universe**:

dx, dy, dz - Object translation (viewing position)

sx, sy, sz - Object scale (zoom)

theta, phi, omega - Object orientation (pan)

These three sets of parameters respectively represent the position of the origin of the universe with respect to the viewer (viewing position), how much the view is magnified to the user (zoom), and where the origin is with respect to the user (pan).

Transformations

Transformations of locations in 3D space can be reduced to four-dimensional matrix arithmetic[10]. A location in space can be represented by a four-dimensional row vector (*x y z 1*). When this vector left-multiplies any 4-by-4 transformation matrix, the resulting row vector represents the transformed point. Tables 2, 3, and 4 illustrate the 4-by-4 transformation matrices for rotation around each axis.

Table 2. Z-Axis Rotation Matrix

cos	sine	0	0
-sin	cos	0	0
0	0	1	0
0	0	0	1

Table 3. Y-Axis Rotation Matrix

cos	0	-sin	0
0	1	0	0
sin	0	cos	0
0	0	0	1

Table 4. X-Axis Rotation Matrix

1	0	0	0
0	cos	sin	0
0	-sin	cos	0
0	0	0	1

It can be shown that these matrices can be used to account for a rotation about any arbitrary axis passing through the origin. The transformation matrix shown in Table 5 corresponds to scaling a location by (**sx**, **sy**, and **sz**) and then moving it by (**dx**, **dy**, and **dz**).

Table 5. Translation and Scaling Matrix

sx	0	0	0
0	sy	0	0
0	0	sz	0
dx	dy	dz	1

The arbitrary transformation of a frame can be defined by a matrix resulting from a multiplication of a subset of the above transformation matrices. However, this multiplication is in general, not commutative. That is, rotating around the x-axis and then translating is not the same as translating and then rotating about the x-axis. By sending values for the nine parameters, the host can request the adjustment of an object. However, *this* system defines these operation as always taking place in the order below:

1) Scale object by (**sx**, **sy**, and **sz**)
2) Translate object by (**dx**, **dy**, and **dz**)
3) Rotate object around z-axis by **theta**.
4) Rotate object around x-axis by **omega**.
5) Rotate object around y-axis by **phi**.

When the matrices shown in Tables 2 through 5 are multiplied, the resulting matrix always contains (*0 0 0 1*)T as its final column. Thus, to denote an arbitrary transformation, you need only remember the first three columns of the composite matrix. If you were to apply the transformations in the order stated previously, the resulting equations in Table 6 would determine the element of the transformation matrix R.

Table 6. Transformation Equations

$r_{12} = s_y \sin\theta$	(2.2)
$r_{13} = s_z \sin\Omega$	(2.3)
$r_{14} = \cos\Omega\ (d_x \cos\theta - d_y \sin\theta) + d_z \sin\Omega$	(2.4)
$r_{21} = s_x(\sin\theta\ \cos\phi + \cos\theta\ \sin\Omega\ \sin\phi)$	(2.5)
$r_{22} = s_y(\cos\theta\ \cos\phi - \sin\theta\ \sin\Omega\ \sin\phi)$	(2.6)
$r_{23} = -s_z \cos\Omega\ \sin\phi$	(2.7)
$r_{24} = \sin\phi\ (\sin\Omega\ (d_x \cos\theta - d_y \sin\theta) - d_z \cos\Omega) + \cos\phi\ (d_x \sin\theta + dy \cos\theta)$	(2.8)
$r_{31} = s_x(\sin\theta\ \sin\phi - \cos\theta\ \sin\Omega\ \cos\phi)$	(2.9)
$r_{32} = s_y(\cos\theta\ \cos\phi + \sin\theta\ \sin\Omega\ \cos\phi)$	(2.10)
$r_{33} = s_z \cos\Omega\ \cos\phi$	(2.11)
$r_{34} = \cos\phi\ (\sin\Omega\ (-d_x \cos\theta + d_y \sin\theta + d_z \cos\Omega) + \sin\phi\ (d_x \sin\theta + dy \cos\theta)$	(2.12)

Note that there also exists a matrix **p[3][4]** (see Listing 1 in the Appendix) that represents the product of all the ancestral transform matrices of an object and that object's R matrix. This matrix represents the object's transformation from the absolute origin of the system.

The Host Processor's Access to Objects

The 80X86 host can exert its control over objects in the following ways:

1) Target Objects - The host can set the target object for adjustment, deletion, or insertion of a child object by either targeting the parent object or a particular child object of the currently targeted object.
2) Load and Delete Objects - The host has the ability to add objects to the system with initial transform parameters. In addition, it can remove objects from the system (including all objects within the deleted objects). When the targeted object is deleted, the new target object defaults to being the object's parent.
3) Adjust Objects - By specifying the nine transform parameters, the host can adjust an object in its parent's frame.
4) Change Perspective - To change the viewing perspective, the host must request that the ***universe** be adjusted.
5) Update Screen Representation - The host can request that the targeted object and its child objects have their location array's screen representations updated.
6) Redraw View - Once all adjustments and updates of screen coordinates are re-specified, the host can request that the view be updated.

Overall, the **object** structure serves well as a data representation for 3D graphics. A single set of locations is available to be referenced by the points, line segments, and filled polygons to be rendered on the screen. Each **object** contains parameters and matrices that specify the transformed state of the object. Thus, at any time these matrices could be applied to the original co-ordinates

loaded into the system to calculate the transformed location of the point. Therefore, as the transformation and the projection on to two-dimensional co-ordinates are done in one step, the original 3D coordinates can be retained and only the final modified two-dimensional screen representation need be updated. The point of view can simply be modified by adjusting the ***universe** as one would adjust any other object. Overall, the hierarchical **object** structure provides a powerful and flexible way to manage graphical data.

DSP Command Execution

The digital signal processor assumes the role of the object manager and keeps track of the representations. Before examining the precise manner in which the TMS320C30 processes the commands from the host, one needs to understand the underlying hardware of this subsystem. A description of the TMS320C30 Application Board can be found in the application report *TMS320C30 Application Board Functional Description*, located in this book. The report describes the avenues of communication between the C30AB and the PC over the PC's bus. An examination of how the TMS320C30 receives and processes data and commands from the 80X86/7 follows.

Initialization

As its first initialization task, the PC maps the dual-port SRAM of the C30AB into its address space by writing the 8 MSBs of address to the mapping register. It then brings the C30AB out of reset by writing a 1 to the **SWRESET** in the C30AB's control register. The PC then loads the TMS320C30 application program into the dual-port SRAM. Loader support software on the C30AB EEPROM moves the code to the proper location in the TMS320C30's address space. Finally, the PC switches the TMS320C30's memory map into run mode to start program execution. The first part of the **main** routine initializes the system (see Listing 8 in the Appendix).

For the system software to run properly, the DSP software must initialize several different items.

1) It enables the on-chip instruction cache.
2) It sets the external flag bit on the C30AB target connector to transfer control of the rendering system from the PC to the C30AB (This assumes that the PC loaded the rendering software before it started up the C30AB).
3) It configures both the primary and the expansion bus with zero software wait-states. Thus, all wait states are generated by the address-decoding PALs on the C30AB.

In addition, the linker configures

1) Primary bus SRAM as program storage
2) Expansion bus SRAM as heap memory allocation
3) Zeroth page of internal RAM as space for system constants
4) First page of internal RAM as the system stack. This configuration maximizes the potential for parallel data and instruction accesses

The initialization procedure then appropriates several local variables for system use, including

1) Two registered looping variables, **i** and **j**
2) The constant 2 PI
3) Registered pointers to the communication registers of the rendering subsystem, ***hstdata** and ***hstcntl**

The TMS320C30 initially sets the contents of these GSP registers to indicate that the computational stage does not have any requests of the rendering stage.

The TMS320C30 system software contains the global variables shown in Listing 7 of the Appendix. The dual-port SRAM pointer **dual_port** is initialized to point to the lowest location on the I/O expansion bus. This pointer points to an integer array that contains all data and command from the PC. Another pointer to the currently targeted object (***to**) is set to reference the **universe**. The ***universe** is set as its own parent with an obnum of 0, indicating no internal objects are loaded.

During the final part of initialization, the C30AB software waits for the PC to load the static ***universe** object. To understand how the PC loads objects into the system, you must comprehend the general communications protocol between the TMS320C30 and the 80X86.

Host to DSP Communication

A two-way polling scheme arbitrates access of the dual-port SRAM. The software allocates the first two words of the SRAM as **COMMAND** and **ACKNOWLEDGE** signals, respectively (see Listing 6 in the Appendix). Remember that the TMS320C30 must mask off the 24 MSBs of dual-port data to receive the proper 8-bit value. The processors poll and write to these two words in order to send requests and acknowledgments. During initialization, the TMS320C30 clears both the **COMMAND** and **ACKNOWLEDGE** locations of the dual-port SRAM. The PC graphics application software must run after this point to ensure that this phase of the initialization does not clear a command from the PC. Once the system software starts executing on both the PC and the TMS320C30, the following sequence enables the PC to send a command to the C30AB:

1) The PC waits for the dual-port SRAM to become free by polling the **ACKNOWLEDGE** word for a zero.
2) The PC loads all command parameters into the dual-port SRAM.
3) The PC then loads the appropriate command byte into **COMMAND**.
4) Once the TMS320C30 returns to its command detection loop, it acknowledges a received command by writing the same byte into the **ACKNOWLEDGE** word.
5) The PC sees that the TMS320C30 has acknowledged the command and writes 00h into COMMAND to withdraw its command. The PC thereby relinquishes control of the dual-port SRAM.
6) The TMS320C30 reads all necessary parameters into its main memory.
7) The TMS320C30, by writing a zero to the **ACKNOWLEDGE** word, indicates that the PC can request another command. This returns the sequence to step (1).

The TMS320C30 treats all of its data types as 32-bit values, but it can read only one byte of valid data from the dual-port SRAM. Thus, the TMS320C30 must mask and concatenate the bytes that the PC maps into contiguous locations to form multibyte words. In addition, since Intel and

the TMS320C30 have different standards, floating-point values from the PC must be converted before the TMS320C30 can use them.

The TMS320C30 can receive either unsigned 8-bit **chars** or unsigned 16-bit short integers from the PC. The macros shown in Listing 6 of the Appendix are used to access these data types from the dual-port SRAM. The **DPLONG** macro takes a certain location in the dual-port, finds the short integer located there, and concatenates it into a 32-bit value for the TMS320C30. The word **LONG** in the macro indicates all integers whether **chars**, **shorts**, or **longs** are represented as 32-bit values by the TMS320C30.

Table 7. Comparison of Intel and TMS320C30 32-Bit Floating-Point Formats

Standard	Exponent Field Bits	Exponent Format	Sign Bit	Mantissa Field	Mantissa Format
TMS320C30	31-24	Two's Complement	23	22-0	Two's Complement
Intel	30-23	Offset Binary	31	22-0	Magnitude

Table 7 illustrates the differences between the TMS320C30 and the Intel single-precision floating-point formats. For every floating-point value that the TMS320C30 receives, it must extract the appropriate fields, convert the fields to the appropriate numerical representation, and then reassemble the fields in TMS320C30 floating-point format. The **dpfloat** routine shown in Listing 9 of the Appendix uses the union structure **fllong** shown in Listing 6 of the Appendix to allow manipulations normally available only for integers on the floating-point value. The program first concatenates the four-byte value in the dual-port SRAM into a single 32-bit integer and then converts this word to TMS320C30 format.

Computational Subsystem Software

Using the communication techniques described in the last section, the TMS320C30 processes the graphics command from the PC. After performing C30AB initialization, the program **main** enters a command detection/execution loop. For each valid value of the **COMMAND** byte, a C **case** statement executes the appropriate code. Since these routines are, in general, too long to be discussed in exhaustive detail, the rest of this section merely summarizes how they work.

When the PC wants to load an object, it first loads the initial nine floating-point transformation parameters into the dual-port SRAM. It then loads the number of

1) Locations
2) Drawn points
3) Lines
4) Filled polygons

These values are limited to 16 bits, thereby allowing for only 65,535 primitives of each type. The size of the dual-port SRAM further limits the array sizes in this implementation. Then the PC loads three floating-point parameters, (x,y, and z), for each location. The size of the dual port limits the number of locations to 377. Once these parameters are loaded into the memory, the host places the command byte for an object load into **COMMAND**. Upon reception of these parameters, the TMS320C30 allocates space for the object as a child of the current target object and also allocates

space for the location, point, and line arrays. Because the size of each polygon varies, space is allocated as each polygon is read.

After allocating global space for the new object and loading the locations, the TMS320C30 requests more data from the PC. It first requests the points, then the lines, then each polygon. The dual-port SRAM limits the primitive arrays to 2047 points and 1364 lines. In addition, each polygon is limited to 4092 vertices. The TMS320C30 makes a data request by replacing the current **COMMAND** byte that it wrote in **ACKNOWLEDGE** with 127, the flag for the PC to load more data. Although the roles of **ACKNOWLEDGE** and **COMMAND** are reversed in this case, the TMS320C30 requests data in much the same way the PC requests commands. Once the TMS320C30 completes loading the object, it selects the object as the new target object. Finally, using the equations in Table 6, the TMS320C30 calculates the initial value of the object's transformation matrix.

The target object is the object in the hierarchy selected for adjustment, deletion, or calculation of screen coordinates. The PC can either target an object's parent or one of the object's child objects. The command to target a child requires the PC to specify either the child object's sibling number or **subnum**. Thus, when selecting objects for adjustment, the PC must remember where it loaded objects into the hierarchy.

To adjust the transformation parameters of a given object, the PC simply loads the new parameters into the dual-port SRAM. The TMS320C30 adds the values of the new angles of rotation and translation factors to the previous ones. In addition, the TMS320C30 multiplies the old scaling factors by the new ones. Then, the TMS320C30 calculates the transformation matrix of the object by using the equations in Table 6. It does not recalculate screen locations, however, until this is specifically requested by the PC. The TMS320C30 can thus avoid calculating screen coordinates until all adjustments have been made.

Once the PC requests all the changes for a frame on the display, it requests recalculation of screen coordinates at each node it changed. The PC can request recalculation for a particular object and thus update its internal objects as well. This allows the TMS320C30 to avoid recalculating screen coordinates of unchanged locations. For maximum efficiency, the PC must request recalculation in the highest node that it adjusted along any particular path. Thus, in the planetary example given earlier, if, in a period of time, only Pluto and its moon Charon were moved (the other bodies miraculously standing still), only Pluto would need to be targeted for recalculation.

To calculate transformations, the TMS320C30 multiplies the object's transformation matrix by *its parent's* parent transformation matrix to obtain *its own* parent transformation matrix, **p[3][4]**. The TMS320C30 right-multiplies all locations within that object by this matrix to achieve the transformation from the absolute origin of the system. The computational engine calculates perspective by dividing the transformed x- and y-coordinate by the transformed z-coordinate so that locations farther away appear closer together. The plane $z=0$ is defined to be the plane of the screen. This also has the feature that objects behind the viewer appear upside-down in front of the viewer because the objects' z-coordinates are negative. Thus, the program running on the PC must maintain all objects in front of the viewer. Then, the TMS320C30 recursively executes this procedure for each object within the targeted object.

Unlike the recalculation of screen coordinates, the redrawing of objects is done for all objects within the system. Thus, the **draw_object** routine is called with the ***universe** as the argument. The

precise manner in which the TMS320C30 uses this program to redraw the screen is described in the TMS320C30 Drawing Routine Section found later in this report.

Summary of DSP Command Execution

The dual-port SRAM on the C30AB provides all means of communication between the PC and the TMS320C30. A two-way polling scheme arbitrates the TMS320C30's and the PC's access to this SRAM. Using this protocol, the PC can request object loading, deletion, or adjustment, but can request only modification of the object currently targeted for these changes. Also, at the host's request, the computational engine may recalculate the screen representation of all locations within the targeted object. Once all updates for a particular view are made, the PC may request a redrawing of the display. The description of the rendering subsystem, presented next, facilitates a better understanding of how the TMS320C30 requests rendering commands of the GSP.

The Rendering Subsystem

A modified version of the TMS34010 Software Development Board serves as the rendering stage of this graphics pipeline. A complete overview of this PC-based card can be found in the *TMS34010 Software Development Board User's Guide* [2]. Because only minor modifications were made to the commercially available SDB, the hardware aspects of the rendering subsystem are discussed in less detail than the computational stage. The same holds true for many software routines taken from the *TMS34010 Math/Graphics Function Library*.[8] After presenting overviews of the TMS34010 and the SDB, this section focuses on the C30AB/SDB interface and the communications protocol used for command and data transfer between the TMS320C30 and the GSP.

The TMS34010 Graphics System Processor

The TMS34010 combines the best features of general-purpose processors and graphics controllers in one powerful and flexible Graphics System Processor. Key features of the TMS34010 are its speed, high degree of programmability, and efficient manipulation of hardware-supported data types, such as pixels and two-dimensional pixel arrays.

The TMS34010's unique memory interface reduces the time needed to perform tasks such as bit alignment and masking. The 32-bit architecture supplies the large blocks of continuously-addressable memory that are necessary in graphics applications. TMS34010 system designs can take advantage of video RAM technology to facilitate applications such as high-bandwidth frame buffers; this circumvents the bottleneck often encountered when using conventional DRAMs are used in graphics systems.

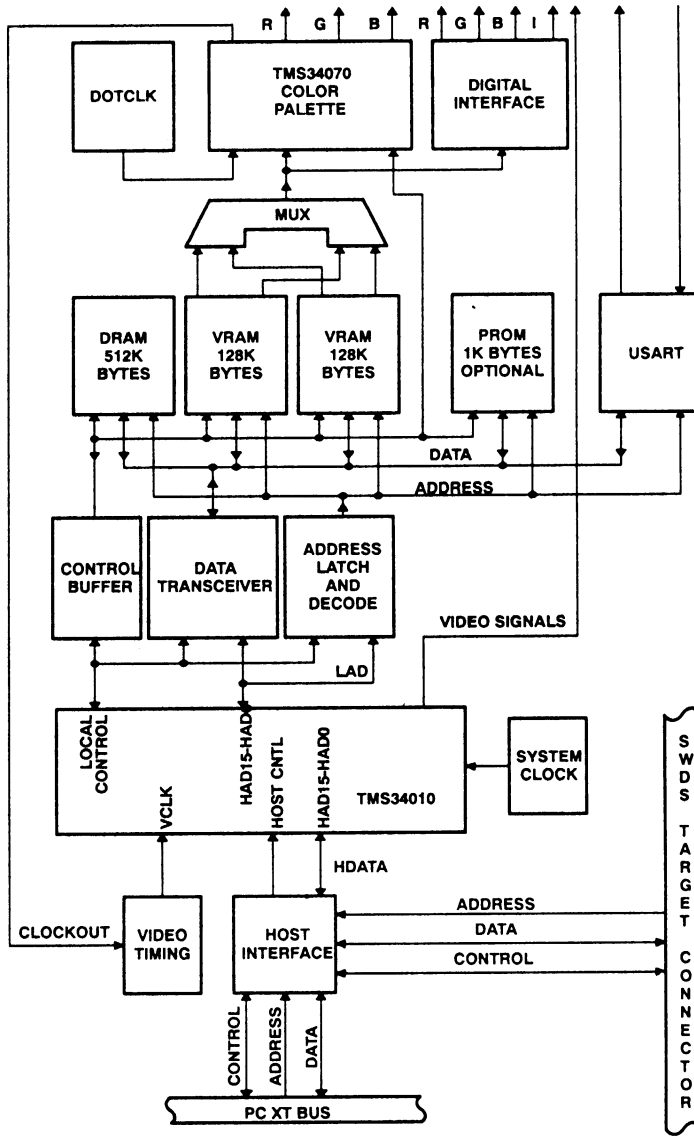
The TMS34010's instruction set includes a full complement of general-purpose instructions, as well as graphics functions from which you can construct efficient high-level functions. The instructions support arithmetic and Boolean operations, data moves, conditional jumps, plus subroutine calls and returns.

The TMS34010 architecture supports a variety of pixel sizes, frame buffer sizes, and screen sizes. On-chip functions have been carefully selected so that no functions tie the TMS34010 to a particular display resolution. This enhances the portability of graphics software and allows the TMS34010 to adapt to graphics standards such as MIT's X, CGI/CGM, GKS, NAPLPS, PHIGS, and other evolving industry and display management standards.

TMS34010 Software Development Board

Figure 4 shows the block diagram of the modified TMS34010 SDB. The graphics SDB is a single card designed around the IBM PC/XT Expansion Bus and serves as a software development tool for programmers writing application software for the TMS34010 Graphics System Processor. The development of a high-performance bit-mapped graphics display in this application report demonstrates the simplicity of hardware design using the TMS34010 SDB.

Figure 4. Modified TMS34010 Software Development Board Block Diagram



This board comes with interactive debug software. Its features include software breakpoints, software single-step and run with count. At the same time, current machine status is displayed on the top half of the host monitor.

The SDB contains 512K bytes of program RAM for the TMS34010 to execute drawing functions, application programs, and displays. Both the program RAM and the frame buffer are accessible to the host through the TMS34010's memory-mapped host port.

The frame buffer consists of eight SIP memory modules organized into four color planes. This allows 16 colors per frame from the digital monitor. The TMS34070 color palette incorporates a 12-bit color lookup table to give you a choice of 16 colors in a frame from a 4096-color palette. Furthermore, the palette incorporates a variety of unique line load features to allow the color lookup table to be reloaded on every line; this means that 16 of 4096 colors can be displayed per line.

The TMS34010 Host Interface

The GSP has two 16-bit buses: one interfaces with the video and program memory, and a second interfaces to a host processor. The host can access the GSP by writing and reading four internal memory-mapped GSP 16-bit registers:

- **HSTADRL** and **HSTADRH** together form a 32-bit pointer to a location in the GSP's address space.
- **HSTCNTL** contains several programmable fields that control host interface functions.
- **HSTDATA** buffers data that is transferred through the host interface between the GSP's local memory and the host processor.

Several signals are available for communications between the host and the GSP.

- **HD15** through **HD0** are the actual data lines.
- **HCS** is the interface select signal strobe from the host.
- **HSF1** and **HSF0** select which host register is being addressed.
- **HREAD** and **HWRITE** are, respectively, the read and write strobes from the host.

Table 8 shows how the above signals address the four host registers.

- **HLDS** and **HUDS** signals, respectively, select the low byte or the high byte of the host interface registers.
- **HRDY** informs the host when the GSP is ready to complete a transaction.
- **HINT** is the interrupt signal from the host to the GSP.

Table 8. TMS34010 Signals Controlling Host Port Interface

Host Interface Control Signals				
HCS	**HSF1 & HSF0**	**HREAD**	**HWRITE**	**Operation**
1	XX	X	X	No Operation
0	00	0	1	HSTADRL read
0	00	1	0	HSTADRL write
0	01	0	1	HSTADRH read
0	01	1	0	HSTADRH write
0	10	0	1	HSTDATA read
0	10	1	0	HSTDATA write
0	11	0	1	HSTCNTL read
0	11	1	0	HSTCNTL write

The fields in **HSTCNTL** control host interrupt processing, auto-incrementing of the host address register, and protocol in byte-at-a-time accesses to the 16-bit host port (whether the lower or the higher byte comes first). **HSTCNTL** also contains the status of interrupts from the host to the GSP and from the GSP to the host and a three-bit message word in either direction. These control bits are shown in Table 9.

Table 9. TMS34010 Host Control Register Fields

Field	Name	Purpose	Write Access
0 – 2	MSGIN	Input Message Buffer	Host Only
3	INTIN	Input Interrupt Bit	Host Only
4 – 6	MSGOUT	Output Message Buffer	GSP Only
8	INTOUT	Output Interrupt Bit	GSP Only
8	NMI	Nonmaskable Interrupt	Host Only
9	NMIN	Nonmaskable Interrupt	GSP and Host
10	Unused	Unused	Neither
11	INCW	Increment Pointer Address on Write	GSP and Host
12	INCR	Increment Pointer address on Read	GSP and Host
13	LBL	Lower Byte Last	GSP and Host
14	CF	Cache Flush	GSP and Host
15	HLT	Halt TMS34010 Processing	GSP and Host

TMS320C30 Application Board Interface

In its unmodified form, the SDB communicates to the PC host through a single transceiver. A PAL decodes the PC address into the appropriate register selection signals. The registers are mapped redundantly into blocks of PC memory address space, as shown in Table 10. The board was modified by the addition of a connector to a cable from the C30AB's target connector. The TMS320C30 sends to the modified SDB the following:

- The TMS320C30s expansion bus address
- The TMS320C30s data signals
- I/O address space access strobe
- Expansion bus read and write strobes

These signals map the GSP's host interface registers in the TMS320C30's address space (also shown in Table 10). The TMS320C30 mapping is actually replicated in four-word blocks until location 8057FFh.

Table 10. Mapping of TMS34010 Host Control Registers

Register	PC Mapping	TMS320C30 Mapping
HSTDATA0	C7000h - C7CFFh	805002h
HSTCNTL	C7D00h - C7DFFh	805003h
HSTADRL	C7E00h - C7EFFh	805000h
HSTADRH	C7F00h - C7FFFh	805001h

The modified SDB board must be able to select either the PC or the C30AB as its host. The C30AB target connector makes the two external flag bits **XF0** and **XF1** available to the SDB. The TMS320C30 can configure these flags as either input or output pins. Upon leaving reset, these pins default to inputs and remain in the high-impedance state. **XF0** is pulled low on the SDB to appear off when the TMS320C30 is in reset. After the PC loads the rendering software into the GSP, it activates the C30AB and loads the TMS320C30's software. As discussed earlier, the TMS320C30, during initialization, configures **XF0** as an output and loads it with a one. The address-decoding PALs on the SDB use this signal to select the C30AB as the SDB's host. When the TMS320C30 controls the SDB, it communicates through a full 16-bit interface to the GSP. Thus, before the integer screen coordinates are sent in two's-complement form to the GSP, they must be clipped to a range of -32,768 to 32,767. Fortunately, this range is still two orders of magnitude greater than the resolution of most monitors.

In general, the above interface is fairly straightforward. The only complication is that the designers of the GSP expected a relatively slow microcoded general-purpose processor as a host. This allows the GSP to actually assert its **HRDY** line 80 ns before it is actually ready to process a transaction. When interfacing to the TMS320C30, PALs become necessary as state machines to create the appropriate number of wait-states on host reads and writes and thus ensure proper interprocessor communication.

DSP to GSP Communication

The TMS320C30 loads all commands and data into a command buffer contained within a space not usually mapped by the SDB's C compiler configuration. This portion of GSP address space, the Shadow RAM, is normally reserved for optional PROMs. However, by writing a 1 to an RS latch in the GSP's memory space, this area becomes occupied by the topmost portion of program/data DRAM. Before the TMS320C30 starts writing to **HSTDATA** to access this memory, it configures the host address to autoincrement. Once the GSP finishes processing data in the shadow RAM, it resets the value of the address registers to point to the beginning of the shadow RAM in order to allow the TMS320C30 to properly load its next command and data.

The communication protocol between the TMS320C30 and the GSP closely resembles the protocol between the PC and the TMS320C30. The **MSGIN** and **MSGOUT** fields, respectively, replace the **COMMAND** and **ACKNOWLEDGE** words. However, rather than these fields con-

taining a particular value for a command, the value of 3 (binary 011) in either of these fields indicates that a command or an acknowledge exists. Upon reception of a command request, the GSP refers to the first location of the shadow RAM for a command word from the TMS320C30. Thus, the overall command scheme proceeds as follows:

1) The TMS320C30 waits until it sees that the **MSGOUT** field contains a 0.
2) The TMS320C30 stores all command and data into the shadow RAM.
3) The TMS320C30 writes a 3 to the **MSGIN** field and waits for acknowledgment.
4) The GSP acknowledges the reception of a command by writing a 3 to the **MSGOUT** field.
5) The TMS320C30 withdraws its request by writing a 0 to **MSGIN**.
6) The GSP reads the first word of the shadow RAM for the command and jumps to the appropriate case to process it.
7) Once the GSP is finished with all data in the shadow RAM, it resets the values of the host address registers and then writes a 0 to the **MSGOUT** bit, indicating that the TMS320C30 is free to request another command.

The TMS320C30 Drawing Routine

When the TMS320C30 receives a redraw-screen request from the PC, it sends a command to the GSP to clear the screen after the monitor has drawn the bottom line; this ensures that the last view was drawn in its entirety. The TMS320C30 then calls its **draw_object** routine with ***universe** as an argument. For each array of primitives within the **object**, the TMS320C30 sends the size of the array and the array of screen representations of the primitives themselves to the TMS34010. Thus, the TMS320C30 can request the GSP to draw arrays of points, lines, or filled polygons. Once all arrays are drawn, **draw_object** recursively executes for all child objects within the universe. In this manner, all objects defined within the system are drawn.

GSP System Initialization

Several initialization routines are provided in the *TMS34010 Math/Graphics Function Library User's Guide* [8]. The GSP executes these programs to properly configure the system before it begins its command detection loop:

- The call to **init_video** configures the graphics buffer for an NEC Multisync Monitor displaying 640 x 480 resolution.
- The **init_graphics** function initializes the graphics environment by setting up the data structures for the graphics functions and assigning default values to system parameters.
- The **init_screen** command initializes the screen. The entire frame buffer is cleared, and a color lookup table is loaded with the default color palette.
- The **init_vuport** function initializes the viewport data structures and opens viewport 0, the system, or root window.
- The **set_origin** command sets the origin of the system to the center of the screen.

Drawing Routines

Several drawing routines are also provided in the *TMS34010 Math/Graphics Function Library User's Guide* [8]:

- For each primitive in an array sent from the TMS320C30, the GSP sets the proper drawing color with the **set_color** command.
- The TMS320C30 commands the GSP to execute to the **clear_screen** before it starts to request drawing of primitives for the next view.
- The TMS320C30 requests a **wait_scan** execution from the GSP to ensure that the GSP has fully displayed the last view before drawing the current view.
- The GSP uses the **draw_point(x,y)** function to render a point on the display.
- Similarly, it uses the **draw_line(x1,y1,x2,y2)** command to draw a line. The arguments are the screen coordinates of the two end-points of the segment.
- The **fill_polygon(n, linelist, ptlist)** function takes as arguments of the number of vertices, an array of the line segments forming the sides of the polygon, and a list of screen coordinates referenced by the linelist.

Summary

The TMS34010 Software Development board provides a good rendering module for this graphics system. The support hardware has been debugged and used in industry since 1987 and thus makes a reliable rendering subsystem. The target connector to the C30AB provides access to the TMS320C30 as an alternate host. Three PALs and two transceivers allow the TMS320C30 to assume control of the GSP, once both have started running their software. The **draw_object** program on the TMS320C30 can command the GSP to draw graphics primitives. Functions in the *TMS34010 Math/Graphics Function Library User's Guide* [8] allow the GSP to initialize the monitor interface, clear the screen, ensure that an entire screen has been drawn, and draw the graphics primitives. Overall, the TMS34010 development tools provide an easy means to develop a rendering subsystem for this graphics pipeline.

Possible Improvements

Several changes may be incorporated into the system to improve performance. Some simple enhancements involve modifications of the computational subsystem's software to allow faster and more transparent command execution. Restructuring the method in which the data and command pass through the pipeline, a more complex modification, can greatly increase throughput. Additional features such as more complex primitives, lighting, windowing, and text display would require major software modifications to the system. However, any such modifications would not need to change the communication protocols or the command detection loops significantly. Finally, although the TMS320C30 represents the state-of-the-art in digital signal processing, the host processor and the rendering engine may be improved.

Computational Subsystem Software

The drawing routine currently sends the primitive arrays of an object one at a time to the GSP. Instead, it should send all primitive arrays for all objects to be redrawn in a single pass. The GSP should then process the contents of this stack of commands and data.

Currently, as soon as the PC finishes requesting objects adjustments, it must request recalculations of the screen coordinates of location arrays. The screen_object routine must operate on all

objects that have been adjusted directly or indirectly by having their ancestors adjusted. Instead, this routine should be called once with the ***universe** as the argument. The **object** structure should contain a flag that is set when an object is adjusted and reset when it is drawn. Thus, the new **screen_object** procedure would recursively search down the hierarchy of objects until it encounters an object that has been adjusted and then should recalculate all the screen coordinates for it and those of its internal objects. Upon completion, it should search the rest of the hierarchy for adjusted objects. Thus, the host would have to request only adjustment, targeting, and draw commands. Screen representations would be automatically recalculated whenever a draw command is executed.

Rendering Subsystem Software

Rendering subsystem drawing routines could be improved by designing functions coded to handle the primitive arrays rather than individual programming elements. These functions may be able to fit in the GSP's instruction cache and improve execution time.

Improved Data Flow

One problem consistent at all stages of the system is the method of buffering. A single buffer usually contains all data and commands to be transferred from one stage to the next. Thus, during command execution one processor may wait for the other to relinquish control of the command buffer.

The first of two methods to improve the dual-port SRAM connecting the PC and the DSP is to divide the SRAM into two buffers. The PC writes the current command to one buffer, while the TMS320C30 processes commands and data stored in the other. This prevents contention for the dual-port SRAM. The particular buffer which each processor controls is swapped on each command request. Second, adding three more 4K x 8 dual-port SRAMS in parallel would allow the PC to communicate to the TMS320C30 with full 32-bit wide words. Thus, the masking and concatenation necessary to receive larger data types would become unnecessary. On the original design the potential addition of these RAMs consumed a prohibitive amount of board space. Full word size is possible only if space constraints are eased.

The splitting of the command buffer between the TMS320C30 and the GSP allows the GSP to draw the current screen while the TMS320C30 sends the primitive arrays for the next. Similarly, two display buffers allow one buffer to be displayed on the monitor while the GSP draws the next view to the other.

Computational Features

The DSP is suited to perform many other types of computational features. Because these functions are more complex, they were not implemented in the limited design time available. This system truncates objects that are too high, too low, too far right, or too far left by using the GSP's drawing routines that automatically clip coordinates outside the screen boundaries. However, the system cannot determine whether one object is in front of another and draw the objects appropriately. Functions to do this hidden-surface removal require complex algorithms to determine whether

one 3D surface obscures another. Simpler routines could be made to clip objects that are too far away to see or objects that are behind the viewer.

A lighting feature would allow appropriate factors of light intensity and reflection to determine the shading of surfaces. Lighting may be ambient (equal everywhere) or come from several possible source geometries. Reflections could either be diffuse and scatter light equally in all directions, or be specular like those off any shiny surface. With these parameters, the TMS320C30 can compute the appropriate shading of a given pixel. In this scenario, the GSP is reduced to drawing single points with a given color. Thus, any lighting function would slow rendering time.

More complex primitives can be produced by using the TMS320C30 to generate arrays of pixels representing solutions to equations. The PC could dispatch a command to draw a primitive based on a particular type of equation (such as the parametric equations representing a sphere) and then load the appropriate parameters for that equation. The DSP would generate the appropriate set of pixels for that object and send it to the GSP as arrays of points.

Rendering Features

The *TMS34010 Math/Graphics Function Library* [8] permits the user to create and select various windows for display. Once a window is selected the DSP can run the existing system software within that window. Thus, the host would also need to be able to direct the DSP to tell the GSP how to manipulate its windows. The Library also enables the GSP to print text on the screen. This feature also would not be very difficult to implement.

A More Advanced Host

A more advanced host could be a high-speed RISC processor such as SPARC. This unit could communicate with the DSP at faster rates, so command transfers would consume less time. In addition, SPARC is a 32-bit machine, which could allow word transfers between host and DSP in a single instruction.

A More Advanced Rendering Engine

The TMS34010's performance as a rendering engine could be improved. If the GSP could be ready to complete a transaction when the **HRDY** line is asserted and not some period of time later, the C30AB to SDB interface would be more straightforward and not require as many wait states. This problem is corrected in the second-generation GSP TMS34020, which was not available at the time of the design of this system. In addition, the TMS34020 also allows the host to transparently access the GSP's bus while the GSP continues processor functions.

Conclusion

Despite its shortcomings, this system still demonstrates the dataflow in a graphics pipeline using a digital signal processor as a computational element. One main benefit of the digital signal

processor is the availability of development tools such as C compilers, assembler/linkers, software development boards, and in-circuit emulators that accelerate design time. The TMS320C30 also provides speeds comparable to many bit-slice processors that require programmers to develop extensive microcode routines. The hardware multiplier, floating-point capability, RISC architecture, and parallel bus access facilitate fast, precise graphics calculations. Overall, a digital signal processor provides an attractive option to the graphics system designer interested in making high-performance systems with quick turnaround time.

References

1) *TMS320C30 Simulator User's Guide* (literature number SPRU017), Texas Instruments, 1989.
2) *Third-Generation TMS320 User's Guide* (literature number SPRU031), Texas Instruments, 1988.
3) *TMS320C30 C Compiler User's Guide* (literature number SPRU034), Texas Instruments, 1988.
4) *TMS320C30 Assembly Language Tools User's Guide* (literature number SPRU035), Texas Instruments, 1989.
5) *TMS34010 Software Development Board Schematics* (literature number SPVU003), Texas Instruments, 1986.
6) *TMS34010 Software Development Board User's Guide* (literature number SPVU002A), Texas Instruments, 1987.
7) *TMS34010 User's Guide* (literature number SPVU001A), Texas Instruments, 1988.
8) *TMS34010 Math/Graphics Function Library User's Guide* (literature number SPVU006), Texas Instruments, 1987.
9) *TMS34010 C Compiler Reference Guide* (literature number SPVU005A), Texas Instruments, 1986.
10) Foley, J.D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison Wesley, 1984.

Appendix A

Graphics Programs

```
*********************************************************************

-->Listing 1: TMS320C30 C Structure Representing an Object

struct object
{
   struct object *parent;/* object within who's frame the object is defined */
   long subnum;                              /* sibling number of object   */
   long locnum;                              /* number of locations        */
   long ptnum;                               /* number of points           */
   long lnnum;                               /* number of lines            */
   long pgnum;                               /* number of polygons         */
   long obnum;                               /* number of daughter objects */
   float sx;      float sy;      float sz;   /* scale factors              */
   float dx;      float dy;      float dz;   /* offsets                    */
   float theta;          /* angle of rotation around z-axis (x to y)       */
   float phi;            /* angle of rotation around x-axis (y to z)       */
   float omega;          /* angle of rotation around y-axis (z to x)       */
   float r[3][4];        /* matrix formed by scale, the offset, then rotate */
   float p[3][4];        /* ascending product of all ancestral r matrices  */
   loc      *locs;       /* pointer to location array                      */
   point    *points;                  /* pointer to point array           */
   line     *lines;                   /* pointer to line array            */
   polygon  *polygons;                /* pointer to polygon array         */
   struct object *objects[MAXOB];     /* pointer to array of              */
                                      /*pointers to child objects         */
};

*********************************************************************


*********************************************************************

-->Listing 2: TMS320C30 C Structure Representing a Location

typedef struct
{
   float x;     float y;     float z;              /* world  coordinates */
   long a;      long b;                            /* screen coordinates */
} loc;

*********************************************************************

*********************************************************************

-->Listing 3: TMS320C30 C Structure Representing a Point

typedef struct
{
   long color;
   long locn;                        /* number of location in location array */
} point;

*********************************************************************


*********************************************************************

-->Listing 4: TMS320C30 C Structure Representing a Line

typedef struct
{
   long color;
   long startlocn;                                  /* start loc number */
   long endlocn;                                    /* end loc number   */
} line;

*********************************************************************


*********************************************************************

-->Listing 5: TMS320C30 C Structure Representing a Filled Polygon

typedef struct
{
   long color;
   long vertnum;                          /* number of vertices            */
   long *vertlocn;                        /* array of vertices loc numbers */
} polygon;

*********************************************************************
```

```
**************************************************************************

-->Listing 6: TMS320C30 Communications Macros

/*-------------------------------------------------------------------*/
/*              COMMUNICATIONS MACROS TO GSP                         */
/*-------------------------------------------------------------------*/
#define CTLFREE     0x0800
#define CTLREQ      0x0803
#define CTLACK      0x0833
#define CTLWITH     0x0830
#define HOSTCNTL    (*hstcntl & 0x00FFF)
/*-------------------------------------------------------------------*/
/*              MAXIMUM NUMBER OF INTERNAL OBJECTS                   */
/*-------------------------------------------------------------------*/
#define MAXOB       10
/*-------------------------------------------------------------------*/
/*              PC COMMUNICATION LOCATIONS                           */
/*-------------------------------------------------------------------*/
#define COMMAND     (*dual_port & 0x0FF)
#define ACKNOWLEDGE dual_port[1]
/*-------------------------------------------------------------------*/
/*              DATA RECOVERY FROM THE DUAL PORT                     */
/*-------------------------------------------------------------------*/
#define DP0(a)      dual_port[a]
#define DP1(a)      dual_port[a + 1]
#define DP2(a)      dual_port[a + 2]
#define DP3(a)      dual_port[a + 3]
#define DPLONG(a)   ((long) ((DP1(a) & 0x00FF) << 8 : (DP0(a) & 0x00FF)))

**************************************************************************


**************************************************************************

-->Listing 7: TMS320C30 Global Variables

long k,l;                           /* temporary and looping variables    */
struct object *universe, *to, *no;  /* universe, target object, next object */
unsigned long *dual_port;           /* dual port SRAM                     */
union                               /* variable to construct a c30 format */
{                                   /*  float from intel format allowing  */
float           f;                  /*  bit manipulation on a float       */
unsigned long   l;
} fllong;

**************************************************************************
```

```
**************************************************************************

-->Listing 8: TMS320C30 Main Command Execution Loop

void main()
{
    register float twopi = 6.283185308;
    register long i, j;
    register long *hstdata = (long *) 0x805002; /* 340 host data register   */
    register long *hstcntl = (long *) 0x805003; /* 340 host control register*/
    dual_port = (unsigned long *) 0x804000;
    asm("       OR      0800h,ST");/* enable cache                         */
    asm("       LDI     02h,IOF"); /* set XF0 and assume control of 340SDB */
   /* set for zero internal wait states on both buses                       */
    *((unsigned long *) 0x808060) = 0;
    *((unsigned long *) 0x808064) = 0x1000;
    *hstcntl    = CTLFREE;/* turn off any request to TMS34010               */
    *dual_port  = 0;      /* turn off any request from the PC               */
    ACKNOWLEDGE = 0;      /* turn off any acknowlegement to the PC          */
   /* allocate space for the internal object                               */
    universe = (struct object *) malloc (sizeof(struct object));
    to       = universe;          /* target universe                       */
    to->subnum = 0;               /* set universe sibling number to 0      */
    to->parent = to;              /* universal object is its own parent    */
    while(COMMAND != 1);          /* first command must be a load object   */
    ACKNOWLEDGE = 1;              /* acknowledge that c30 is ready         */
    while(COMMAND != 0);          /* wait for pc to withdraw request       */
    load_object();                /* load universe                         */
    ACKNOWLEDGE = 0;              /* show that dual port is free           */
    matrix();                     /* calculate transformation matrix       */
    for(;;)                       /* infinite loop for PC command detection*/
    {
        while(COMMAND == 0);      /* wait for PC to request service        */
        j = COMMAND;              /* save command                          */
        ACKNOWLEDGE = j;          /* acknowledge request                   */
        while(COMMAND != 0);      /* wait for PC to withdraw request       */
        switch (j)                /* execute requested command number      */
        {
           case 1:                /* LOAD A DAUGHTER OBJECT                */
               if (to->obnum == MAXOB) break; /* abort if > maximum objects */
               j = ++to->obnum;   /* increase number of daughter objects   */
                                  /* allocate space for new object         */
            to->objects[j] = (struct object *) malloc (sizeof(struct object));
               no = to->objects[j]; /* next object is daughter object      */
               no->subnum = j;    /* set sibling number of next object     */
               no->parent = to;   /* assign current object as no's paret   */
               to = no;           /* target daughter object                */
               load_object();     /* load daughter object                  */
               ACKNOWLEDGE = 0;   /* show that dual port is free           */
               matrix();          /* calculate transform matrix            */
               break;

            case 2:               /* TARGET A DAUGHTER OBJECT              */
               j = DPLONG(2);     /* get daughter object number to target  */
```

```
    ACKNOWLEDGE = 0;    /* show that dual port is free          */
    if (j > to->obnum) break; /* can only target existing object */
    to = to->objects[j]; /* target daughter object              */
    break;

 case 3:                /* TARGET PARENT OBJECT                 */
    ACKNOWLEDGE = 0 ;   /* show that dual port is free          */
    to = to->parent;    /* set targeted object to parent        */
    break;

 case 4:                /* DELETE TARGETED OBJECT               */
    ACKNOWLEDGE = 0;    /* show that request dual port is free  */
    if (to == universe) break;/* don't allow deletion of universe*/
    j = to->subnum + 1; /* get number of next sibling           */
    no = to->parent;    /* set next object to parent            */
    delete_object(to);  /* delete current object                */
    to = no;            /* target parent object                 */
    l = to->obnum;      /* find total number of siblings        */

   /* decrement sibling number on all younger siblings          */
    for(i = j; i <= l; ++i) --to->objects[i]->subnum;
    --to->obnum;   /* decrement total number of daughter objects */
    break;

case 5:                 /* ADJUST TARGETED OBJECT               */
    to->sx    *= dpfloat(2); /* adjust scales                   */
    to->sy    *= dpfloat(6);
    to->sz    *= dpfloat(10);
    to->dx    += dpfloat(14); /* adjust offsets                 */
    to->dy    += dpfloat(18);
    to->dz    += dpfloat(22);
    to->theta += dpfloat(26); /* adjust angles                  */
    to->phi   += dpfloat(30);
    to->omega += dpfloat(34);
    ACKNOWLEDGE = 0;          /* show that dual port is free    */

   /* keep angles in the (0,2pi) range)                         */
    to->theta = fmod(to->theta, twopi);
    to->phi   = fmod(to->phi  , twopi);
    to->omega = fmod(to->omega, twopi);
    matrix(to);             /* recalculate transform matrix     */
    break;

case 6:                     /* DRAW UNIVERSE                    */
    ACKNOWLEDGE = 0;        /* show that dual port is free      */

    while(HOSTCNTL != CTLFREE); /* wait for 340 to be free      */
    *hstdata = 4;           /* enter command for a screen clear */
    *hstcntl = CTLREQ;      /* request service from 340         */
    while(HOSTCNTL != CTLACK); /* wait for acknowledgement      */
    *hstcntl = CTLWITH;     /* withdraw request                 */

    draw_object(universe);  /* draw universe                    */
            while(HOSTCNTL != CTLFREE); /* wait for 340 to be free      */
            *hstdata = 6;           /* enter command for a scanline     */
            *hstcntl = CTLREQ;      /* request service from 340         */
            while(HOSTCNTL != CTLACK); /* wait from acknowledgement     */
            *hstcntl = CTLWITH;     /* withdraw request                 */
            break;

          case 7:               /* CALCULATE SCREEN COORDINATES         */
            /* +++WARNING+++ the PC user must execute a screen command to */
            /* screen all objects that have been adusted since the last   */
            /* draw before the next draw. However, if an object is        */
            /* screened all daughter objects are as well.                 */
             ACKNOWLEDGE = 0;     /* show that dual port is free         */
             screen_object(to);   /* calcuclate screen coordinates       */
             break;

          default:
              ACKNOWLEDGE = 0;              /* show that dual port is free */
              break;
        }
    }
}
```

```
*************************************************************************

*************************************************************************

-->Listing 9: TMS320C30 Floating-Point Conversion Routine

float dpfloat (a)
register unsigned long a;            /* offset from start of dual port SRAM */
{
   register unsigned long sign;
   unsigned long mant, ex;

   a =   (DP3(a) << 24              /* concatenate 4-byte value             */
       | (DP2(a) & 0x00FF) << 16
       | (DP1(a) & 0x00FF) << 8
       | (DP0(a) & 0x00FF));
   sign = (a & 0x80000000) >> 8;   /* extract and reposition sign bit      */
   ex   =  ((a & 0x7F800000)       /* extract exponent                     */
          - 0x3f800000) << 1;      /* converts to 2's complement           */
   if (sign)
   {
      mant = (- a) & 0x007FFFFF;   /* takes 2's complement of mantissa     */
      if (mant == 0) ex  -= 0x01000000; /* checks for input mantissa of -2 */
   }
   else mant = a & 0x007FFFFF;     /* otherwise leave mantissa alone       */
   a = sign + mant + ex;           /* reconstruct floating-point fields    */
   fllong.i = a;
   return fllong.f;                /* return reconstructed float           */
}
```

```
*************************************************************************
```

```
*************************************************************************

-->Listing 10: TMS320C30 Object Loading Routine

void load_object()
{
   register long i,j;                    /* temporary and looping variables */
   register struct object *o;            /* pointer to target object        */
   register loc     *temploc;            /* temporary location pointer      */
   register line    *templn;             /* temporary line pointer          */
   polygon          *temppg;             /* temporary polygon pointer       */
   point   *temppt;                      /* temporary point pointer         */
   long lc   = DPLONG(2);                /* number of coordinate locations  */
   long pt   = DPLONG(4);                /* number of points                */
   long ln   = DPLONG(6);                /* number of lines                 */
   long pg   = DPLONG(8);                /* number of polygons              */

    o = to;                          /* set target object as object for loading /

  /* initialize primitive numbers and transform parameters                  */
   o->locnum = lc;
   o->ptnum  = pt;
   o->lnnum  = ln;
   o->pgnum  = pg;
   o->obnum  = -1;
   o->sx     = dpfloat(10);    o->sy  = dpfloat(14);   o->sz    = dpfloat(18);
   o->dx     = dpfloat(22);    o->dy  = dpfloat(26);   o->dz    = dpfloat(30);
   o->theta  = dpfloat(34);    o->phi = dpfloat(38);   o->omega = dpfloat(42);

  /* ALLOCATE SPACE FOR OBJECT PRIMITIVES                                   */
   o->locs   = (loc *) malloc (sizeof (loc) * lc);
   o->points     = (point   *) malloc (sizeof (point  ) * pt);
   o->lines    = (line  *) malloc (sizeof (line ) * ln);
   o->polygons   = (polygon *) malloc (sizeof (polygon) * pg);

  /* LOAD UPTO 377 LOCATIONS PER OBJECT                                     */
   for (i = 0, j=46; i < lc; ++i, j += 12)
   {
       temploc    = &(o->locs[i]);              /* save temporary location */
       temploc->x = dpfloat(j);                 /* load world coordinates  */
       temploc->y = dpfloat(j + 4);
       temploc->z = dpfloat(j + 8);
   }

  /* LOAD UPT 2047 POINTS PER OBJECT                                        */
   if (pt)
   {
       more_data();
       for (i = 0, j=2 ; i < pt; ++i, j += 4)
       {
           temppt       = &(o->points[i]); /* set temporary point location */
           temppt->color = DPLONG(j);      /* get point color              */
           temppt->locn  = DPLONG(j + 2);  /* get point location           */
       }
   }

  /* LOAD UPTO 1364 LINES                                                   */
   if (ln)
   {
       more_data();
       for (i = 0, j=2 ; i < ln; ++i, j += 6)
       {
           templn            = &(o->lines[i]);     /* set temporary line    */
           templn->color     = DPLONG(j);          /* get color             */
           templn->startlocn = DPLONG(j + 2);      /* get starting location */
           templn->endlocn   = DPLONG(j + 4);      /* get ending location   */
       }
   }

  /* LOAD ONE POLYGON AT A TIME                                             */
   if (pg)
   {
       for (i = 0; i < pg; ++i)
       {
           more_data();
           temppg           = &(o->polygons[i]);  /* set temporary polygon  */
           temppg->color    = DPLONG(2);          /* get color              */
           l                = DPLONG(4);          /* get number of verteces */
           temppg->vertnum  = l;                  /* set number of verteces */

          /* allocate space for vertex location list                         */
           temppg->vertlocn =  (long *) malloc (sizeof (long) * l);


           for (k = 0, j = 6;  k < l; ++k, j += 2)         /* load verteces */
           {
               temppg->vertlocn[k] = DPLONG(j);      /* set vertex location */
           }
       }
   }
}

*************************************************************************
```

```
*************************************************************************

-->Listing 11: TMS320C30 Screen Coordinate Calculation Routine

void screen_object(o)
register struct object *o;
{
   register long i,j;                  /* temporary and looping variables    */
   register loc *temploc;              /* temporary location pointer         */
   register struct object *tempob;     /* temporary object pointer           */
   register float x,y;                 /*  co-ordinate floating point values */
   float z,d;                          /*  and perspective constant          */

   tempob = o->parent;              /* set temporary object to parent object  */

  /* COMPUTE PARENT MATRIX                                                    */
  /* if object is universe set parent matrix to transform matrix r           */
   if (o == universe)
   {
       for(i = 0; i < 3; ++i) for(j = 0; j < 4; ++j)  o->p[i][j] = o->r[i][j];
   }

  /* otherwise p matrix is product of r matrix and parent's p matrix         */
   else for(i = 0; i < 3; ++i)
   {
      o->p[i][0] = o->r[0][0] * tempob->p[i][0] + o->r[1][0] * tempob->p[i][1]
                 + o->r[2][0] * tempob->p[i][2];
      o->p[i][1] = o->r[0][1] * tempob->p[i][0] + o->r[1][1] * tempob->p[i][1]
                 + o->r[2][1] * tempob->p[i][2];
      o->p[i][2] = o->r[0][2] * tempob->p[i][0] + o->r[1][2] * tempob->p[i][1]
                 + o->r[2][2] * tempob->p[i][2];
      o->p[i][3] = o->r[0][3] * tempob->p[i][0] + o->r[1][3] * tempob->p[i][1]
                 + o->r[2][3] * tempob->p[i][2] + tempob->p[i][3];
   }

  /* COMPUTE SCREEN COORDINATES                                               */
   j = o->locnum;                             /* get number of locations */
   for (i = 0; i < j; ++i)
   {
       temploc = &(o->locs[i]);                /* set temporary location */

      /* save global coordinates                                            */
       x = temploc->x;          y = temploc->y;          z = temploc->z;

      /* calculate z value, add offset of 5, and invert for perspective     */
  d = 1/(x * o->p[2][0] + y * o->p[2][1] + z * o->p[2][2] + o->p[2][3] + 10);

      /* calculate transformed x and y, add perspective, and scale to screen*/
       k = (long) ((x * o->p[0][0] + y * o->p[0][1]
                + z * o->p[0][2] +     o->p[0][3]) * d * 200);
       l = (long) ((x * o->p[1][0] + y * o->p[1][1]
                + z * o->p[1][2] +     o->p[1][3]) * d * 200);

      /* clip to a 16 bit integer                                           */
       if      (k >  32000) k =  32000; else if (k < -32000) k = -32000;
       if      (l >  32000) l =  32000; else if (l < -32000) l = -32000;

      /* set screen coordinates                                             */
       temploc->a = k;               temploc->b = l;
   }

  /* screen all internal objects                                             */
   j = o->obnum;
   for (i = 0; i <= j; ++i) screen_object(o->objects[i]);


*************************************************************************


*************************************************************************

-->Listing 12: TMS320C30 Transformation Matrix Evaluation Routine

matrix()
{
   register float         cost, sint;          /* transform temporary     */
   float                  coso, sino, cosp, sinp;/* variables             */
   register struct object *o;

   o = to;
   cost = cos(o->theta);
   sint = sin(o->theta);
   coso = cos(o->omega);
   sino = sin(o->omega);
   cosp = cos(o->phi);
   sinp = sin(o->phi);
   o->r[0][0] = o->sx * cost * coso;
   o->r[0][1] = - o->sy * sint * coso;
   o->r[0][2] = o->sz * sino;
   o->r[0][3] = (o->dx * cost - o->dy * sint) * coso + o->dz * sino;
   o->r[1][0] = o->sx * (sint * cosp + cost * sino * sinp);
   o->r[1][1] = o->sy * (cost * cosp - sint * sino * sinp);
   o->r[1][2] = - o->sz * coso * sinp;
   o->r[1][3] =   ((o->dx * cost - o->dy * sint) * sino - o->dz * coso) * sinp
              + (o->dx * sint + o->dy * cost) * cosp;
   o->r[2][0] = o->sx * (sint * sinp - cost * sino * cosp);
   o->r[2][1] = o->sy * (cost * sinp + sint * sino * cosp);
   o->r[2][2] = o->sz * coso * cosp;
   o->r[2][3] =   ((- o->dx * cost + o->dy * sint) * sino + o->dz * coso)
              * cosp          +  (o->dx * sint + o->dy * cost) * sinp;
}

*************************************************************************
```

```
*****************************************************************

-->Listing 13: TMS320C30 Object Deletion Routine

void delete_object (o)
register struct object *o;
{
   register long i, j;                          /* temporary, looping variables */

   free (o->locs);                              /* delete location array        */
   free (o->points);                            /* delete point array           */
   free (o->lines);                             /* delete line array            */
   j = o->pgnum;                                /* get number of polygons       */
   for (i = 0; i <= j; ++i) free (o->polygons[i].vertlocn); /* delete           */
   for (i = 0; i <= j; ++i) free (o->polygons);             /*  polygons        */
   j = o->obnum;                                /* get number of daughter objects */
   for (i = 0; i <= j;++i) delete_object(o->objects[i]); /* delete objects     */
   free (o);                                    /* delete object                */

*****************************************************************


*****************************************************************

-->Listing 14: TMS320C30 Request for Additional Data in Object Load

void more_data()
{
        ACKNOWLEDGE = 127;          /* request more data                  */
        while(COMMAND != 127);      /* wait for more data                 */
        ACKNOWLEDGE = 1;            /* restore old acknowledge            */
        while(COMMAND != 0);        /* wait for PC to resume old command */


*****************************************************************
```

```
*****************************************************************

-->Listing 15: TMS320C30 Object Drawing Routine

void draw_object (o)
register struct object *o;
{
   register long    i;                          /* temporary, looping variable */
   register loc     *temploc;                   /* temporary location pointer  */
   point            *temppt;                    /* temporary point pointer     */
   register line    *templn;                    /* temporary line pointer      */
   polygon          *temppg;                    /* temporary point pointer     */
   register long *hstdata = (long *) 0x805002; /* 340 host data register      */
   register long *hstcntl = (long *) 0x805003; /* 340 host control register   */
   register j = o->lnnum;                       /*-temporary, looping variable */

  /* DRAW ANY LINES                                                          */
   if (j)
   {
       while (HOSTCNTL != CTLFREE);             /* wait till 340 is free       */
       *hstdata = 123;                          /* send command to draw object */
       *hstcntl = CTLREQ;                       /* request service from 340    */
       *hstdata = j;                            /* send number of lines        */
       for(i=0; i < j; ++i)                     /* send lines                  */
       {
           templn  = &(o->lines[i]);                    /* save line pointer */
           *hstdata = templn->color;                    /* send color        */
           *hstdata = o->locs[templn->startlocn].a;     /* send start        */
           *hstdata = o->locs[templn->startlocn].b;     /*   coordinates     */
           *hstdata = o->locs[templn->endlocn].a;       /* send end          */
           *hstdata = o->locs[templn->endlocn].b;       /*   coordinates     */
       }
       while(HOSTCNTL != CTLACK);    /* wait for 340 to acknoluedge request */
       *hstcntl = CTLWITH;           /* withdraw request                    */
   }

   /* DRAW ANY POINTS                                                       */
    j = o->ptnum;                        /* get number of points           */
    if (j)
    {
       while (HOSTCNTL != CTLFREE);          /* wait till 340 is free       */
       *hstdata = 1;                         /* send command to draw object */
       *hstcntl = CTLREQ;                    /* request service from 340    */
       *hstdata = j;                         /* send number of points       */
       for(i=0; i < j; ++i)                  /* send points                 */
       {
           temppt = &(o->points[i]);            /* save point pointer       */
           *hstdata = temppt->color;            /* send color               */
           *hstdata = o->locs[temppt->locn].a;  /* send screen coordinates  */
           *hstdata = o->locs[temppt->locn].b;
       }
       while(HOSTCNTL != CTLACK);    /* wait for 340 to acknoluedge request */
       *hstcntl = CTLWITH;           /* withdraw request                    */
   }
```

```
  /* DRAW ANY POLYGONS                                                */
   l = o->pgnum;
   if (l)
   {
      for(i = 0; i < l; ++i)            /* draw polygons              */
      {
           temppg = &(o->polygons[i]);  /* wait till 340 is free      */
           j = temppg->vertnum;         /* send command to draw object */
           while (HOSTCNTL != CTLFREE); /* request service from 340   */
           *hstdata = 5;                /* send number of points      */
           *hstcntl = CTLREQ;           /* send points                */
           *hstdata = temppg->color;    /* send color                 */
           *hstdata = j;                /* send number of verteces    */

          /* send point connect list (0,1 , 1,2 , 2,3 .... j-2,j-1 , j-1,0 */
           *hstdata = 0;
           for(k = 1; k < j; ++k)
           {
               *hstdata = k;              *hstdata = k;
           }
           *hstdata = 0;

          /* send vertex location list                                */
           for(k = 0; k < j; ++k)
           {
               temploc = &(o->locs[temppg->vertlocn[k]]); /* save point */
               *hstdata = temploc->a;       *hstdata = temploc->b;
           }
          while(HOSTCNTL != CTLACK); /* wait for 340 to acknolwedge request*/
          *hstcntl = CTLWITH;       /* withdraw request                */
      }
   }

  /* DRAW ANY DAUGHTER OBJECTS                                        */
   j = o->obnum;                                  /* get daughter objects */
   for (i = 0; i <= j; ++i) draw_object(o->objects[i]);
```

****

****

--)Listing 16: TMS34010 Point Structure

```
typedef struct                                              /* POINT */
{
   short color;  /* point color                                      */
   short x;      /* x co-ordinate                                    */
   short y;      /* y co-ordinate                                    */
} point;
```

****

****

--)Listing 17: TMS34010 Line Structure

```
typedef struct                                              /* LINE */
{
    short color; /* line color                                      */
    short x1;    /* x co-ordinate of starting point                 */
    short y1;    /* y co-ordinate of starting point                 */
    short x2;    /* x co-ordinate of end point                      */
    short y2;    /* y co-ordinate of end point                      */
} line;
```

****

****

--)Listing 18: TMS34010 Color Array

```
long color[16] = {
   CC0, CC1, CC2, CC3, CC4, CC5, CC6, CC7, CC8, CC9, CC10, CC11, CC12,
   CC13, CC14, CC15};
```

****

****

--)Listing 19: TMS34010 Color Palette

```
short mypalet[16] = {
   0x0000, 0xF000, 0x00F0, 0xF0F0, 0x0F00, 0xFF00, 0x0FF0, 0xFFF0,
   0x0AF0, 0x0900, 0xFA70, 0xF440, 0x17B0, 0x6660, 0x9990, 0xBBB0 };
```

****

```
*****************************************************************

-->Listing 20: TMS34010 Main Command Execution Routine

main()
{
   register line *templn;              /* temporary line pointer             */
   register point *temppt;             /* temporary point pointer            */
   register short tempint;             /* temporary integer                  */
   register short i;                   /* looping variable                   */
   line  *lines;                       /* pointer to line array              */
   point *points;                      /* pointer to point array             */
   short *hstadrh, *hstadrl, * hstctll, *number, *pgnum, *pointer, adrl, adrh;
   *((short *) 0x04000000) = 0x0001;   /* turn on shadow ram                 */
   *((short *) 0xC00000B0) &= 0x7FFF;  /* enable cache                       */
   hstctll = (short *) 0xC00000F0;     /* host control register low byte     */
   hstadrh = (short *) 0xC00000E0;     /* host address register high word    */
   hstadrl = (short *) 0xC00000D0;     /* host address register low word     */
   pointer = (short *) 0xFFF00000;     /* pointer to beginning of shadow ram */
   lines  = (line  *) (0xFFF00020);    /* starting point of line array       */
   points = (point *) (0xFFF00020);    /* starting point of point array      */
   pgnum  = (short *) (0xFFF00020); /*location of number of polygon verteces*/
   number = (short *) (0xFFF00010);    /* number of primitives to draw       */
   adrl = (short) (((long) pointer)  & 0x0000FFFF);
   adrh = (short) ((((long) pointer) >> 16) & 0x0000FFFF)
   init_video(1);    /* configure for a NEC MULTISYNC, non-interlaced, 60Hz */
   init_grafix();                      /* initialize graphics environment    */
   init_screen();                      /* initialize screen                  */
   init_vuport();                      /* initialize viewing window          */
   set_origin(320,240);                /* place origin at center of screen   */
   *hstadrh = adrh;                    /* reset start data address           */
   *hstadrl = adrl;
   *hstctll = 0;                       /* turn off any command to the 340    */
   for (;;)
   {
       while (*hstctll != 0x0003);     /* wait for request from the C30      */
       *hstctll = 0x0030;              /* acknowledge request                */
       while (*hstctll != 0x0030);     /*wait for c30 to load data & withdraw*/
       switch (*pointer)               /* decode command                     */
       {
           case 123:                                       /* DRAW LINES   */
               tempint = *number;                          /* get number   */
               for (i = 0; i < tempint; ++i)               /*   of lines   */
               {
                   templn = &(lines[i]);                   /*set line point*/
                   set_color1(color[templn->color]);       /* set color    */
                   draw_line( templn->x1,                  /* draw line    */
                              templn->y1,
                              templn->x2,
                              templn->y2);
               }
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               break;
           case 1:                                         /* DRAW POINTS  */
               tempint = *number;                          /* get number of*/
               for (i=0; i < tempint; ++i)                 /*     points   */
               {
                   temppt = &(points[i]);                  /* save point   */
                   set_color1(color[temppt->color]);       /* set colors   */
                   draw_point( temppt->x,                  /* draw point   */
                               temppt->y);
               }
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               break;
           case 3:                                  /* SET SCREEN BACKGROUND */
               new_screen(color[*number],mypalet); /* clear screen          */
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               break;
           case 4:                                  /* SET BACKGROUND BLACK  */
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               new_screen(0,mypalet); /* clear screen                       */
               break;
           case 5:                                  /* DRAW A FILLED POLYGON */
               set_color1(color[*number]);         /* set polygon color     */
               tempint = *pgnum;                   /* get number of verteces*/
               fill_polygon(tempint,               /* fill polygon          */
                            (short *) (pointer + 3),
                            (short *) (pointer + 3 + (tempint << 1)));
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               break;
           case 6:                    /* WAIT FOR COMPLETE SCREEN RESCAN    */
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               wait_scan(0);          /*wait till scan reaches top of screen*/
               wait_scan(479);    /*wait till scan reaches bottom (line 479)*/
               break;
           default:
               *hstadrh = adrh;       /* reset start data address           */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340    */
               break;
       }
   }
}

*****************************************************************
```

```
*********************************************************************

-->Listing 21: PC Object Loading Data Structure

typedef struct
{
   short ptnum;                          /* number of points (location)    */
   short dtnum;                          /* number of drawn dots           */
   short lnnum;                          /* number of lines                */
   short pgnum;                          /* number of filled polygons      */
   float sx;     float sy;   float sz;   /* scale factors                  */
   float dx;     float dy;   float dz;   /* offset factors                 */
   float theta;  float phi;  float omega; /* angles of rotation             */
} trans;

*********************************************************************


*********************************************************************

-->Listing 22: PC Communications Macros

#define DATASHORT(a) *((unsigned short *) (dual_port + a))
#define DATAFLOAT(a) *((float *) (dual_port + a))
#define COMMAND      *dual_port
#define ACKNOWLEDGE  *((unsigned char *) 0xE0008001)

*********************************************************************


*********************************************************************

-->Listing 23: PC Global Variables

char *dual_port;                       /* dual port sram connecting to C30 SWDS*/
trans *data;

*********************************************************************
```

```
*********************************************************************

-->Listing 24: PC Targeted Object Adjustment Routine

void adjust_object(sx, sy, sz, dx, dy, dz, theta, phi, omega)
double sx, sy, sz, dx, dy, dz, theta, phi, omega;
{
   while(ACKNOWLEDGE != 0);
   DATAFLOAT(2)  = sx;     DATAFLOAT(6)  = sy;   DATAFLOAT(10) = sz;
   DATAFLOAT(14) = dx;     DATAFLOAT(18) = dy;   DATAFLOAT(22) = dz;
   DATAFLOAT(26) = theta;  DATAFLOAT(30) = phi;  DATAFLOAT(34) = omega;
   COMMAND = 5;
   while(ACKNOWLEDGE != 5);
   COMMAND = 0;
}

*********************************************************************


*********************************************************************

-->Listing 25: PC Routine to Set Parameters for an Object Load

void set_parameters(sx, sy, sz, dx, dy, dz, theta, phi, omega)
double sx, sy, sz, dx, dy, dz, theta, phi, omega;
{
   while (ACKNOWLEDGE  != 0);          /* wait for C30 to be free           */
   data->sx    = sx;      data->sy    = sy;    data->sz    = sz;
   data->dx    = dx;      data->dy    = dy;    data->dz    = dz;
   data->theta = theta;   data->phi   = phi;   data->omega = omega;
}

*********************************************************************


*********************************************************************

-->Listing 26: PC Routine to Target Parent of Current Target Object

void target_parent()
{
   while(ACKNOWLEDGE  != 0);           /* wait for C30 to be free           */
   COMMAND = 3;                        /* command to target parent object   */
   while(ACKNOWLEDGE  != 3);           /* wait for C30 to acknowlege request*/
   COMMAND = 0;                        /* withdraw request                  */
}

*********************************************************************
```

```
*************************************************************************
```

-->Listing 27: PC Routine to Target a Child of Current Target Object

```
void target_child(x)
int x;
{
      while(ACKNOWLEDGE  != 0);       /* wait for C30 to be free           */
      DATASHORT(2) = x;               /* target 1st daughter object        */
      COMMAND = 2;                    /* command to target daughter object */
      while(ACKNOWLEDGE  != 2);       /* wait for C30 to acknowlege request*/
      COMMAND = 0;                    /* withdraw request                  */
}
```

```
*************************************************************************
```

```
*************************************************************************
```

-->Listing 28: PC Routine to Redraw Screen

```
void draw_object()
{
      while(ACKNOWLEDGE  != 0);       /* wait for C30 to be free           */
      COMMAND = 7;                    /* command to compute screen co-ords */
      while(ACKNOWLEDGE  != 7);       /* wait for C30 to acknowlege request*/
      COMMAND = 0;                    /* withdraw request                  */
      while(ACKNOWLEDGE  != 0);       /* wait for C30 to be free           */
      COMMAND = 6;                    /* command to draw screen            */
      while(ACKNOWLEDGE  != 6);       /* wait for C30 to acknowlege request*/
      COMMAND = 0;                    /* withdraw request                  */
}
```

```
*************************************************************************
```

```
*************************************************************************
```

-->Listing 29: PC Routine to Load the Primitives of a Wireframe Cube

```
void cube(c)
long c;
{
   data->ptnum = 8;                   /* number of points (cube verteces)  */
   data->dtnum = 0;                   /* no dots                           */
   data->lnnum = 12;                  /* twelve lines     (cube edges)     */
   data->pgnum = 0;                   /* no filled polygons                */
/* ----X COORDINATE----   ----Y COORDINATE----    ----Z COORDINATE----     */
   DATAFLOAT(46)  =  1;  DATAFLOAT(50)  =  1;   DATAFLOAT(54)  =  1;
   DATAFLOAT(58)  =  1;  DATAFLOAT(62)  = -1;   DATAFLOAT(66)  =  1;
   DATAFLOAT(70)  =  1;  DATAFLOAT(74)  = -1;   DATAFLOAT(78)  = -1;
   DATAFLOAT(82)  =  1;  DATAFLOAT(86)  =  1;   DATAFLOAT(90)  = -1;
   DATAFLOAT(94)  = -1;  DATAFLOAT(98)  =  1;   DATAFLOAT(102) =  1;
   DATAFLOAT(106) = -1;  DATAFLOAT(110) = -1;   DATAFLOAT(114) =  1;
   DATAFLOAT(118) = -1;  DATAFLOAT(122) = -1;   DATAFLOAT(126) = -1;
   DATAFLOAT(130) = -1;  DATAFLOAT(134) =  1;   DATAFLOAT(138) = -1;
   COMMAND = 1;                      /* command to load object            */
   while (ACKNOWLEDGE != 1);         /* wait for C30 to acknowledge request*/
   COMMAND = 0;                      /* withdraw request                  */
   while (ACKNOWLEDGE != 127);       /* wait for C30 request lines        */
   COMMAND = 127;                    /* command to load lines             */
/* LINE COLOR-------   START POINT------   ENDPOINT---------              */
   DATASHORT(2)  = c;  DATASHORT(4)  = 0;  DATASHORT(6)  = 1;
   DATASHORT(8)  = c;  DATASHORT(10) = 1;  DATASHORT(12) = 2;
   DATASHORT(14) = c;  DATASHORT(16) = 2;  DATASHORT(18) = 3;
   DATASHORT(20) = c;  DATASHORT(22) = 3;  DATASHORT(24) = 0;
   DATASHORT(26) = c;  DATASHORT(28) = 4;  DATASHORT(30) = 5;
   DATASHORT(32) = c;  DATASHORT(34) = 5;  DATASHORT(36) = 6;
   DATASHORT(38) = c;  DATASHORT(40) = 6;  DATASHORT(42) = 7;
   DATASHORT(44) = c;  DATASHORT(46) = 7;  DATASHORT(48) = 4;
   DATASHORT(50) = c;  DATASHORT(52) = 0;  DATASHORT(54) = 4;
   DATASHORT(56) = c;  DATASHORT(58) = 1;  DATASHORT(60) = 5;
   DATASHORT(62) = c;  DATASHORT(64) = 2;  DATASHORT(66) = 6;
   DATASHORT(68) = c;  DATASHORT(70) = 3;  DATASHORT(72) = 7;
   while (ACKNOWLEDGE  != 1);        /* wait for C30 to resume loading    */
   COMMAND = 0;                      /* show no requests                  */
}
```

```
*************************************************************************
```

```
*********************************************************************

-->Listing 30: PC Main Routine to Draw a "Planetary System of" Cubes

main()
{
   register int x;
   dual_port = (char *)  0xE0008000; /* location of dual port sram          */
   data      = (trans *) 0xE0008002; /* location of object data             */
   COMMAND = 0;
   set_parameters(.0001,.0001,.0001,0.,0.,0.,0.,0.,0.);
   cube(3);
   set_parameters(.4,.4,.4,0.,0.,8.,0.,0.,0.);
   cube(2);
   set_parameters(.2,.2,.2,0.,5.,0.,0.,0.,0.);
   cube(6);
   target_parent();
   set_parameters(.2,.2,.2,0.,-5.,0.,0.,0.,0.);
   cube(4);
   target_parent();
   target_parent();
   set_parameters(.3,.3,.3,0.,0.,6.,0.,0.,0.);
   cube(5);
   target_parent();
   set_parameters(.3,.3,.3,0.,6.,0.,0.,0.,0.);
   cube(1);
   target_parent();
   set_parameters(.3,.3,.3,0.,0.,-6.,0.,0.,0.);
   cube(5);
   target_parent();
   set_parameters(.3,.3,.3,0.,0.,-6.,0.,0.,0.);
   cube(1);
   target_parent();
   for(x = 0; x < 1000; ++x)
   {
       adjust_object(1.00926,1.00926,1.00926,0.,0.,0.,0.,0.,.2);
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(2);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(3);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(4);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,0.,-.4);
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,.4,0.,0.);
       target_parent();
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,.4,0.,0.);
       target_parent();
       target_parent();
       screen_object();
       draw_screen();
   }
   for(x =0; x < 1100; ++x)
   {
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.005,.2);
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(2);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(3);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(4);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,-.4);
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,.3,0.,0.);
       target_parent();
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,.3,0.,0.);
       target_parent();
       target_parent();
       screen_object();
       draw_screen();
   }

*********************************************************************
```