Graphics 9

9.1 OVERVIEW

In graphics processing, geometric and topological information constitute the essence of the image. By contrast, in the more familiar image processing, an image consists of pixels. Graphics programs operate on data and data structures such as vector arrays, line lists, and polygons rather than pixels. Graphics processors use rotation, projection, ray tracing, and other techniques to *synthesize original* images, whereas image processors *enhance existing* image aspects such as contrast, definition, and edge delineation. Graphics applications include operations such as geometric modeling and drafting, solids and surface modeling, ray tracing, hidden line removal, shadow casting, texture mapping, perspective views, image synthesis, three-dimensional imagery, and animation.

Generally, high-end graphics applications are limited to 32-bit machines because of the greater resolution necessary in recursive transformations to avoid accumulating observable error. In fact, a floating-point format is often necessary to provide sufficient dynamic range to accommodate zooming and scaling operations. However, a low-end graphics engine that uses the 16-bit fixed-point format of the ADSP-2100 is more than adequate for applications such as video games and small computer graphics packages. To illustrate the graphics processing capabilities of the ADSP-2100, we present such an application in this chapter.

The complete graphics processor solution presented in this chapter consists solely of the ADSP-2100 single-chip microprocessor and some simple analog interface components. The ADSP-2100 performs all aspects of spatial rotation and display of a three-dimensional object in real time, demonstrating the principles of basic graphics operations discussed in this chapter. The example is expressly for demonstration and implements only a subset of the graphics processing capabilities available with the ADSP-2100.

9.2 GRAPHICS PROCESSING SYSTEM

Figure 9.1 shows a block diagram of the graphics processing system based on the ADSP-2100. An analog-to-digital converter (ADC) takes samples of

9 Graphics

a joystick's position as inputs. The graphics processor uses this data to control the amount of rotation of an object displayed on an oscilloscope. A digital-to-analog converter (DAC) generates beam deflection voltages for the oscilloscope from the output of the graphics processor to draw the object. The four-channel 8-bit ADC is memory-mapped into the ADSP-2100 data memory space and joystick input samples are obtained from this memory space. A quad 8-bit DAC is also mapped into the ADSP-2100 memory space; the ADSP-2100 writes data to this memory space to control the oscilloscope beam.

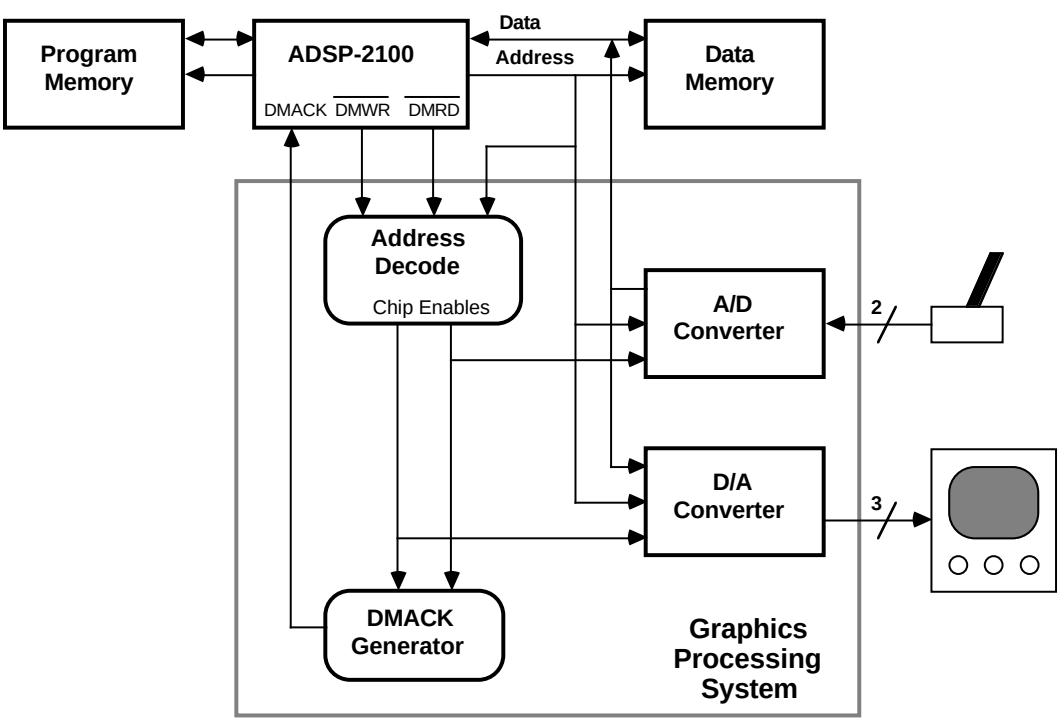


Figure 9.1 Graphics System Block Diagram

The reference object is stored in data memory as a series of (x, y, z) coordinate sets (vectors). Each vector represents a point or vertex of the object and all vertices are numbered. A line list (also stored in data memory) describes where (between which points) lines are to be drawn.

The software provides for rotating the object in four modes. The object can be rotated in each of three dimensions sequentially or in all three dimensions simultaneously. Rotation in these two modes is continuous

and requires no joystick control. The object can also be rotated in the direction indicated by the joystick. The joystick position can be sampled and processed in two different ways, each of which produces a different effect on the motion of the displayed object. A pushbutton in hardware switches from one rotation mode to the next.

In the two joystick-controlled modes, each new set of joystick input samples starts the process of rotating the displayed object. A rotational transform is generated from the joystick data. In the other modes, the rotational transform is generated automatically in software. Matrix multiplication (described in Chapter 8) of the current position of the reference object by the rotational transform calculates the new position of the reference object, point by point. The rotated object is then projected from the three-dimensional spatial coordinate system onto a two-dimensional screen coordinate system to enable it to be displayed; this process is similar to casting the shadow of the object.

The wire-frame drawing of the object is done using the Bressenham line segment drawing algorithm (Foley and Van Dam, 1983). The line list tells the Bressenham algorithm where to draw lines. The actual line drawing is done by moving the oscilloscope beam along the path between the two endpoints of the line. Each line is drawn, a pixel at a time, until the entire object has been completed. The drawing sequence is then repeated ad infinitum.

Other topics covered in this chapter necessarily include data normalization and scaling, finite precision arithmetic, numerical overflow, and saturating arithmetic. Performance measures, data structures, schematics, and program listings pertaining to the example are also presented.

9.3 SETTING THE STAGE

Three-dimensional scenes use a four-dimensional transform space, just as two-dimensional scenes use a three-dimensional transform space, because the (x, y) and (x, y, z) coordinates of two-dimensional and three-dimensional vectors need an additional scale factor, generally referred to as W. In two-dimensional notation, the point P(x, y) is represented as P(Wx, Wy, W), with the scale factor $W \neq 0$. The coordinates for the point P(X, Y, W) are then $x=X/W$ and $y=Y/W$. The scale factor W preserves vector scaling through any transformation. In three-dimensional notation, the point P(x, y, z) is represented as P(Wx, Wy, Wz, W), and the coordinates are recovered similarly.

The ability to scale vectors on a pointwise basis is important because it allows equal resolution of coarse-grained and fine-grained features. For example, one display of a data base may be a view of a space shuttle from a distance of 50 meters, while a second view of the same data base may detail the 1/4-20 bolt positions of the gibulator inside the pod bay door control assembly.

Large values of W allow fine-grained coordinates, which would otherwise underflow an integer format, to be represented with resolution comparable to that of coarse-grained coordinates (which necessarily have smaller values of W). In essence, W can be considered as an exponent associated with each fixed-point coordinate set that is much the same as the exponent a full floating-point hardware implementation would provide.

For the sake of simplicity, we set W=1 in this example so the three-dimensional point P(x, y, z) is represented as P(X, Y, Z, 1), in which x=X, etc. All points are represented as row vectors with normal scaling and (x, y, z) components:

$$P(X, Y, Z, 1) = [x\ y\ z\ 1]$$

The left-handed coordinate system shown in Figure 9.2 is used because this system provides for larger z values to be displayed as being further from the viewer: a more intuitive convention than the familiar right-

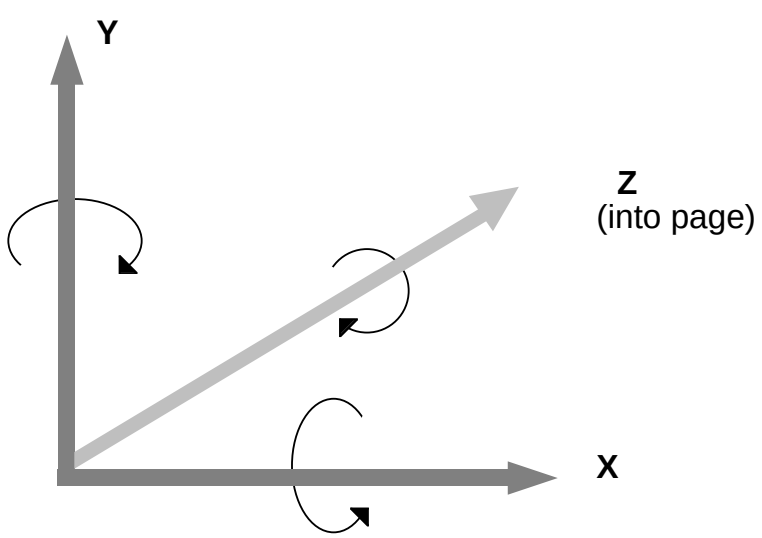


Figure 9.2 Left-Handed Coordinate System

handed system in which the z-axis comes out of the page. Positive rotations for the left-handed system are always clockwise when viewing the origin from a positive axis.

Individual transforms can be concatenated by matrix multiplication to form a single complex transform. The complex transform has the same total effect as each simple transform applied sequentially. Thus, multiple operations can be performed simultaneously, rather that sequentially, saving valuable processor time.

In general, a four-dimensional transform matrix is comprised of various submatrixes corresponding to different operations. Rotational operators comprise a 3x3 submatrix justified to the upper left corner of the 4x4 matrix, translation operators constitute a 1x3 submatrix in the lower left corner, perspective operators constitute a 3x1 in the upper right corner, and zooming (the simultaneous scaling of all three components) uses only a 1x1 element in the lower right corner, as shown in Figure 9.3.

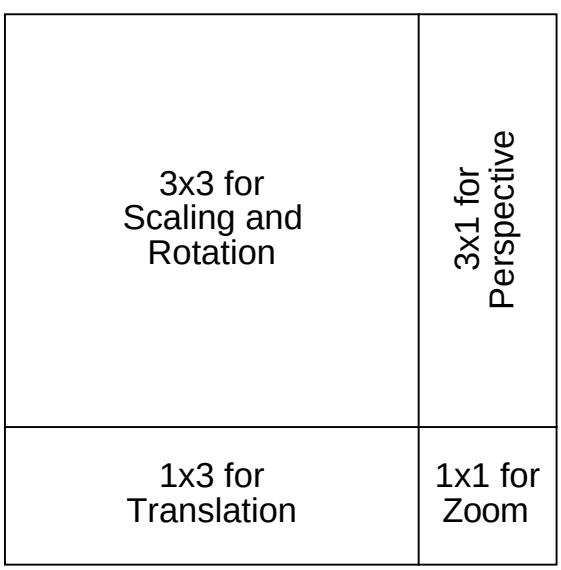


Figure 9.3 Components of the 4x4 Transformation Matrix

The conventional geometric operations that can be performed on three-dimensional coordinates (in a four-dimensional space) are rotation (see Figures 9.4 through 9.6), translation (Figure 9.7), and scaling (Figure 9.8). In these figures, Cx and Sy in the example matrixes denote the

trigonometric functions cos(x) and sin(y), in which x and y are angles of rotation. Perspective transformations and zooming are neglected for the moment.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cx & Sx & 0 \\ 0 & -Sx & Cx & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 9.4 Rotation About the X Axis

$$\begin{bmatrix} Cy & 0 & -Sy & 0 \\ 0 & 1 & 0 & 0 \\ Sy & 0 & Cy & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 9.5 Rotation About the Y Axis

$$\begin{bmatrix} Cz & Sz & 0 & 0 \\ -Sz & Cz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 9.6 Rotation About the Z Axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta x & \Delta y & \Delta z & 1 \end{bmatrix}$$

Figure 9.7 Translation by (Δx, Δy, Δz)

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 9.8 Scaling by (Sx, Sy, Sz)

9.4 COMPUTATIONAL REDUCTIONS IN TRANSFORMATIONS

The transformation matrix can be simplified to reduce computational requirements. There is a tradeoff in the complexity of the graphic display, but we will show that the tradeoff is not significant for this application.

Any number of rotation, scaling, and translation matrixes can be multiplied together before being applied to the object. The result is always a single matrix, **M**, of the form shown in Figure 9.9.

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Figure 9.9 Combined Rotation, Scaling and Translation Matrix

As shown in Figure 9.9, the upper-left 3x3 submatrix, R, gives the aggregate rotation and scaling of all the premultiplied matrixes, while the lower-left 1x3 submatrix **T** gives the aggregate translation. A reduction in the amount of numerical processing to evaluate the overall transform is obtained by the simplification:

$[x' \; y' \; z'] = [x \; y \; z] \bullet \mathbf{R} + \mathbf{T}$

rather than by implementing the full 1x4•4x4 multiplication directly:

$[x' \; y' \; z' \; 1] = [x \; y \; z \; 1] \bullet \mathbf{M}$

The 3x3 matrix provides a much simpler and faster implementation because only 9 multiplications and 6 additions are needed to transform each vector, as opposed to 16 multiplications and 12 additions for the 4x4 matrix: a 56% savings in multiplications alone!

The 3x3 matrix structure preserves both rotation and translation, although the zoom and perspective functions are lost. Applications needing zoom must either preserve the 4x4 transform structure and sustain increased computational load or use the 3x3 structure and apply any zoom operations as a postprocess to the rotation transform. The decision depends on how many vectors there are and what the throughput requirements are. Because we have no great dynamics in this example (W=1 for all points), the loss of the zoom function is of no consequence.

Perspective transformations introduce realism by use of one or more vanishing points. Without perspective, parallel lines converge at a point located at infinity. Vanishing points are imaginary points usually set at some finite distance from the object along a major axis. They move the convergence point of parallel lines in from infinity, introducing foreshortening in which foreground objects appear larger and background objects appear smaller, creating an illusion of realism (see Figure 9.10).

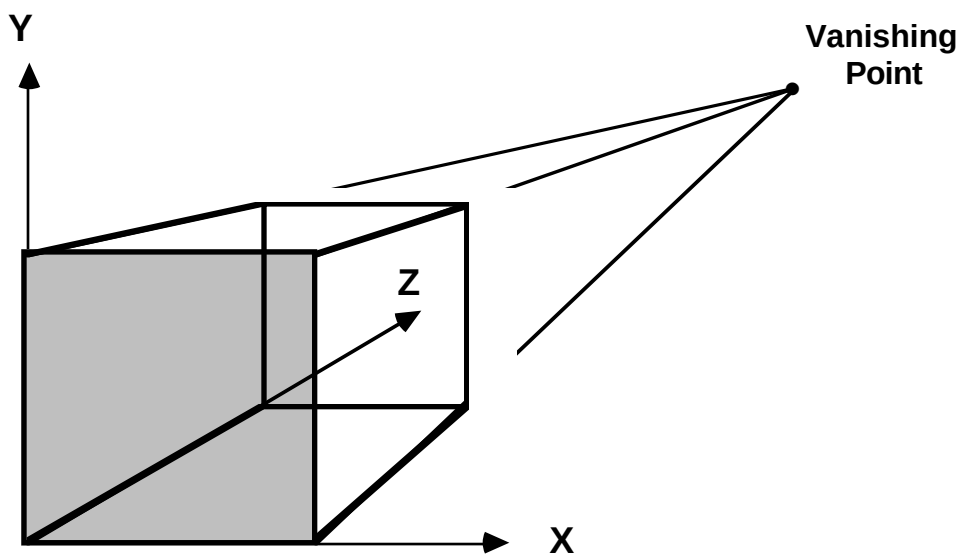


Figure 9.10 Perspective Projection Using One Vanishing Point

Nonzero elements in the 3x1 perspective submatrix migrate the vanishing point associated with the corresponding axis away from infinity. Perspective transformations are lost by the simplification to the more efficient 3x3 matrix structure because all perspective elements are assumed to be zero.

The example shown here uses a simple parallel projection technique and does not need perspective projection. Therefore, the loss of perspective transformations is of no consequence. If perspective transformations are needed, then the transform matrix size must be increased and the efficiency of the whole computational process suffers.

The forfeit of the perspective and zoom transformations for improved efficiency are somewhat subjective decisions. Only a viewing of the final

result will determine whether the correct cost/performance tradeoff has been made. The criteria for making these decisions consist of aesthetic and performance considerations; if more complex (and realistic) visual effects are needed, then use the perspective and zoom transformations.

The combined rotational transform matrix, **R**, used in the example is shown in Figure 9.11. Any translation (matrix **T**) would be applied after calculating **R** using simple addition as shown above. Translation, however, is not demonstrated in this example.

$$\mathbf{R} = \begin{bmatrix} CyCz & CySz & -Sy \\ SxSyCz - CxSz & SxSySz + CxCz & SxCy \\ CxSyCz + SxSz & CxSySz - SxCz & CxCy \end{bmatrix}$$

Figure 9.11 Concatenated Rotation Matrix

9.5 PROJECTION TECHNIQUES

Once the scene has been transformed in three dimensions, it must be projected onto a two-dimensional screen to be viewed. This operation is like casting a shadow onto the sidewalk; a three-dimensional object is projected onto the two-dimensional sidewalk by the sun. Two types of projection techniques exist: the perspective projection and the parallel projection.

Perspective projections are visually more pleasing and intuitive. These projections use vanishing points to which parallel lines (other than those parallel to the projection plane) converge. As a result of this convergence, distant objects seem to be further away because they are smaller than closer objects.

One or two vanishing points are generally used, depending upon the degree of realism desired. The two-vanishing-point scheme (see Figure 9.12, on the next page) is commonly used in engineering sketches, the graphic arts, and architectural and industrial design. Two-point renderings usually preserve vertical parallelism while "parallel" lines in the other two dimensions actually converge to their respective vanishing points.

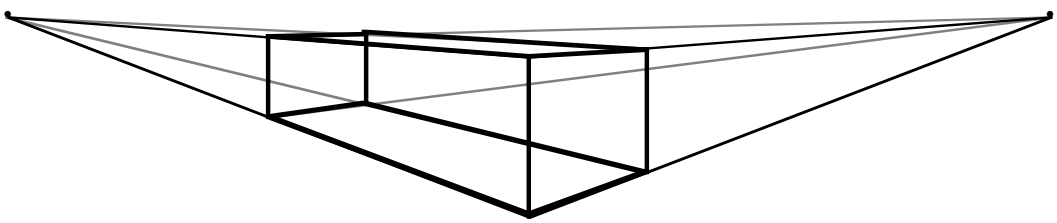


Figure 9.12 Prospective Projection Using Two Vanishing Points

Parallel projections do not use vanishing points at all; parallel lines in three-dimensional space remain parallel in two-dimensional space. The most common parallel projection is the orthographic projection, shown in Figure 9.13, in which top, end, and side views convey the essence of an object. Orthographic projections are so named because the normal of the projection plane is parallel to the projection direction.

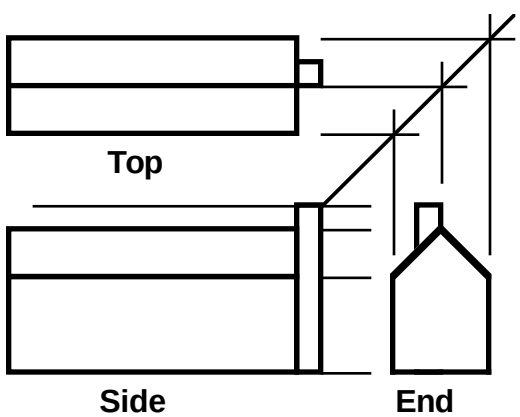


Figure 9.13 Three Orthographic Views of a House

The orthographic projection is used to depict machine parts and building structures because length and angular measures are preserved. Orthographic projections are difficult to visualize because they show only head-on projections of the different sides of objects, and the viewer must conceptualize the image.

A second class of parallel projections is the oblique projection. Oblique projections share the general orthographic property of the projection plane being normal to a principal axis, but differ slightly in that the projection (or viewing) direction is not. Oblique projections preserve linear and angular measure for faces which are parallel to the projection plane. Faces which are not parallel to the projection plane preserve only linear measures, whereas angles are distorted.

Oblique parallel projections, as shown in Figure 9.14, are used extensively because they are easy to draw. Everything remains parallel, yet objects look realistic. Two common oblique projections are the cavalier and the cabinet projections. Each makes a specific angle with the projection plane, cavalier being 45° and cabinet being the arccotangent of 1/2. Generally, cabinet projections look more realistic, but cavalier projections preserve uniform linear measure.

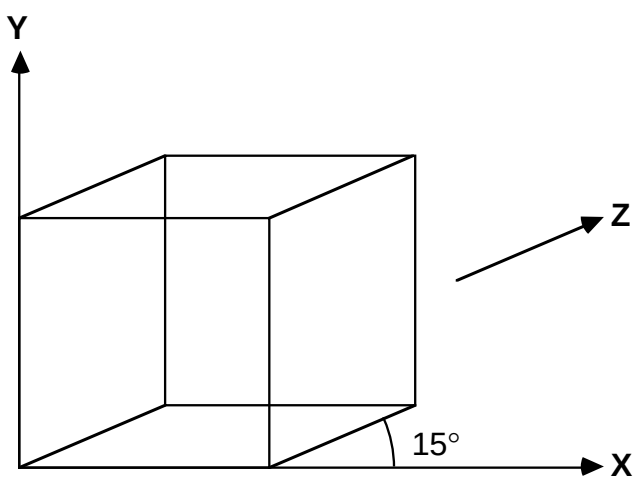


Figure 9.14 Oblique Parallel Projection of a Cube

We use the oblique parallel projection to generate the display data in this example because of its relative simplicity and effective realism. The two-dimensional screen coordinate (x_s, y_s) display data is derived from the transformed three-dimensional coordinates (x, y, z) using simple trigonometry:

$x_s = x + z \cos(15°)$
$y_s = y + z \sin(15°)$

Lines parallel to the z-axis appear to make a 15° angle with the x-axis (as projected on the screen) due to the angle which the projection screen normal and viewpoint (which share a common direction) make with the xy plane. Any angle may be used with varying manifestations in the appearance of the projection.

Note that if the full 4x4 matrix structure is utilized, both the transformation and projection operations may be combined into a single matrix. These operations are distinct here for simplicity and illustration. Foley and Van Dam (see *References* at the end of this chapter) discuss this issue more fully in section 8.2 of their book.

9.6 DATA FORMAT

Hardware multipliers don't know the difference between 101.01012 and 1010.101_2; the placement of the binary point is purely arbitrary as far as the hardware is concerned. However, ADSP-2100 users are strongly encouraged to use the 1.15 format (shown in Figure 9.15) because the ADSP-2100 multiplier is optimized for the 1.15 format.

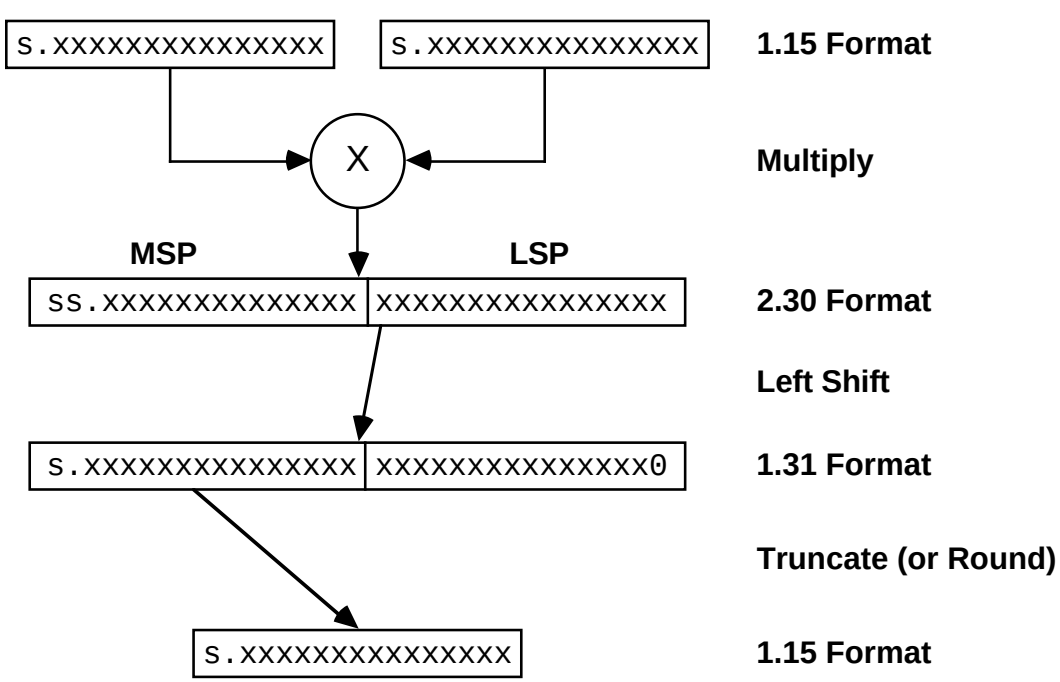


Figure 9.15 The 1.15 Data Format

The ADSP-2100 left-shifts products to renormalize them automatically to a destination format similar to that of the input operands, with the result that there is no binary point migration as long as both input operands are in 1.15 format. Two multiplicands in the 1.15 format produce a 32-bit product in the 2.30 format which is left-shifted to 1.31 format and then and rounded (or truncated) to the MSP (Most Significant 16-bit half of the 32-bit Product) format of 1.15. Automatic normalization works only with the 1.15 format; other formats will manifest binary point migration. Hence, in this example, all data is normalized to the 1.15 format prior to processing.

Note that it is the 16-bit MSP that contains the result in 1.15 format, although product summing is performed in the full 40-bit resolution of the accumulator (in 1.31 format) before rounding (or truncating) to the MSP. (See the complete block diagram and functional description of the MAC in the *ADSP-2100 User's Manual.*)

9.7 NORMALIZATION AND SCALING

Two operations must be performed on the input data before the program will run without data overflows. All input data must be normalized to the largest value, then all normalized data must be adjusted (by upshifting) to the 1.15 format.

Normalization is the division of a set of numbers by their largest member so that the largest number is normalized to unity and the rest of the numbers are all guaranteed to be less than or equal to one. Data normalization is necessary to guarantee that products get smaller after multiplication instead of larger and therefore do not overflow.

Normalized data is upshifted to the 1.15 format so that the automatic renormalization by left-shift works as described above. This upshift is accomplished by multiplying the normalized data with the 16-bit twos-complement positive full scale value ($7FFF_{16} = 2^{15}–1 = 32767$).

In the source data of this example, the largest component of any point vector is 21, so normalization entails the division of all vector components by 21. However, normalization to unity yields a few numbers which are still large enough to cause intermediate results to overflow during the transformation process due to addition operations. Therefore, we increase the normalization factor to guarantee that all overflows are eliminated. By trial and error, we determine that a normalization factor of 30 is sufficient.

Normalization of the source data therefore entails division by 30. For example, the normalized value of 21 is $21 \div 30 = 7/10$. The normalized data

is then multiplied by positive full scale to produce the source data used in the transformation process: for example, 7/10 (32767) = 5999_{16}.

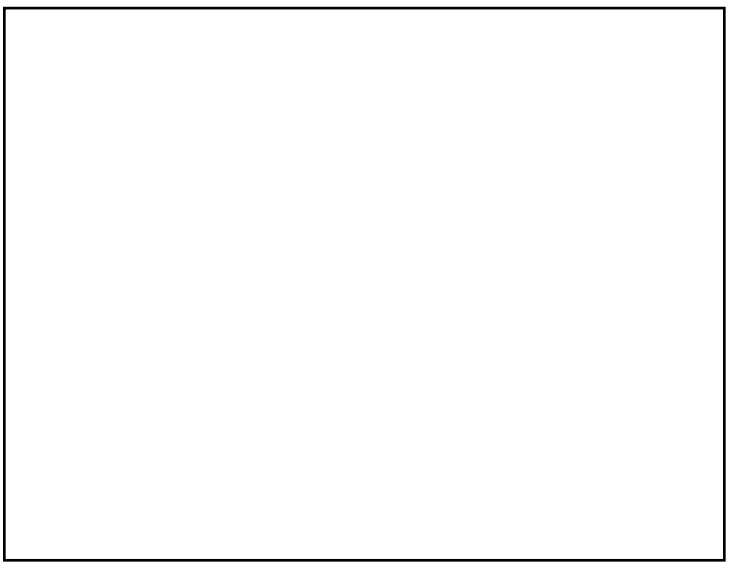
The finite precision of hardware processors' numerical formats and the selection of a data normalization factor (resolution) play crucial roles in the successful development of any numerical processing application. Too much resolution in the data (a small normalization constant) results in less headroom (allowance for overflow of intermediate results) within the fixed word size, while too little resolution (a large normalization constant) distorts the data. The key to success is to balance the normalization with the word size to maintain sufficient headroom throughout the process without compromising resolution to the point of introducing too much distortion.

An example of the problems that arise when too little headroom is provided is illustrated in the two photographs shown in Figure 9.16. Both examples show a slight overflow of the screen coordinate system resulting in points wrapping around the screen edges. Wraparound is due to insufficient normalization scaling (not enough headroom) which produces arithmetic overflows.

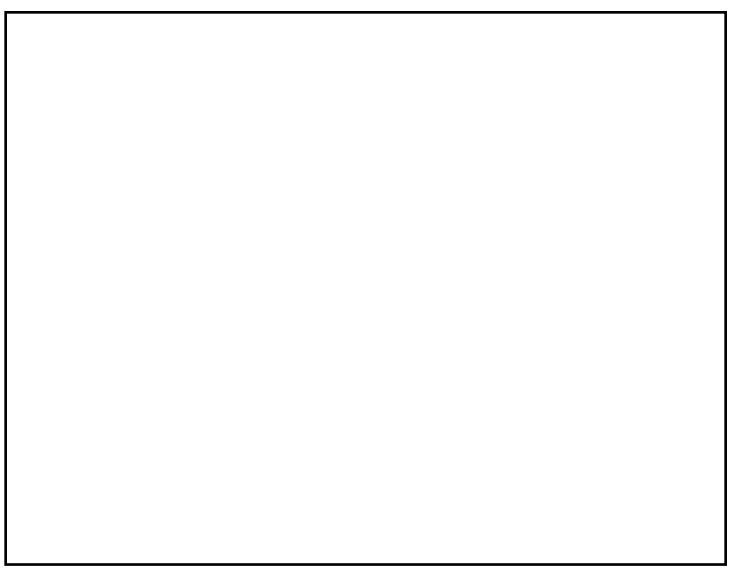
The first photograph illustrates that such wrapped points produce lines which must cross the screen to make their connections. The second photograph illustrates saturating arithmetic (an optional mode of operation on the ADSP-2100 ALU and MAC) in which any overflows are automatically saturated, or set to full scale. Points which would otherwise wrap around the screen are constrained to the edge (clipped). The effect of saturation arithmetic is an appreciable reduction in the severity of overflow distortion.

The upshifting and normalization of input data are necessary to ensure data integrity through transformation and projection. Before displaying the data, however, the output data must be further scaled to adapt to the display driver.

A simple example of a vector graphic display is the oscilloscope. The hardware used in this example employs a straight binary-coded quad 8-bit (not twos-complement) DAC to drive the x and y deflection inputs of an oscilloscope (see the Joystick and Scope Interface schematic at the end of this chapter). The 8-bit resolution of the DAC provides a screen resolution of 256x256 pixels upon which to display the rotating object.



Overflow without Saturation Logic



Overflow with Saturation Logic

Figure 9.16 Overflows With and Without Saturation Logic

9 Graphics

The three-dimensional coordinate system of the source data has its origin located in the center of the object with points (vertices of the object) assuming ± twos-complement values (corresponding to the format of the ADSP-2100) in three dimensions. All two-dimensional display data must therefore be converted to the unsigned 8-bit binary format used by the DAC prior to display. This is done by multiplying each screen coordinate (x_s, y_s) by the DAC's half-scale value (80_{16}) and then adding an offset of half-scale to shift the center of the object to the DAC's half-scale point.

In the example, the maximum value of all source coordinates is 21, which when normalized and converted to 1.15 format, becomes 5999_{16}. Assuming that the worst case gain through rotation and projection is unity, the maximum display value is 5999_{16}. Prior to being written to the DAC, this value is multiplied by the DAC's half-scale value, 80_{16}, which translates the normalized value to a corresponding voltage of the DAC's output range. The left-shifted resultant product is $(5999 \times 0080 = 0059\ 9900)_{16}$, which after rounding to the MSP (recall, we only use MR1), becomes $5A_{16}$, a worst case screen coordinate value.

Adding 80_{16} to $5A_{16}$ yields DA_{16}. This addition simply moves the object to center screen and has no scaling effect. The final value which is written to the DAC for display is DA_{16}. Note that the ratio of the worst case screen coordinate value to the positive full scale DAC value, $(5A:80)_{16}$, is the same as the original source coordinate to the normalization factor (21:30).

Figure 9.17 uses a number line analogy to summarize all the data format transitions and dynamics during operations, and available headroom for each of six stages described above. In summary, these stages are:

Stage 1: The actual source data consisting of manually quantized (x,y,z) coordinates of the object is edited into a data file.

Stage 2: The quantized data is normalized by a Pascal program.

Stage 3: The same Pascal program formats the normalized data producing a hexadecimal data file. This data is ultimately loaded into the allocated area of data RAM on the target system by the ADSP-2100 assembler INIT directive in the main program.

Stage 4: After the ADSP-2100 has performed rotation and projection transformations, the same limits and headroom are present as in the previous stage, but during the processing between these two stages, the computational dynamics of the operations require the headroom to avoid data overflow.

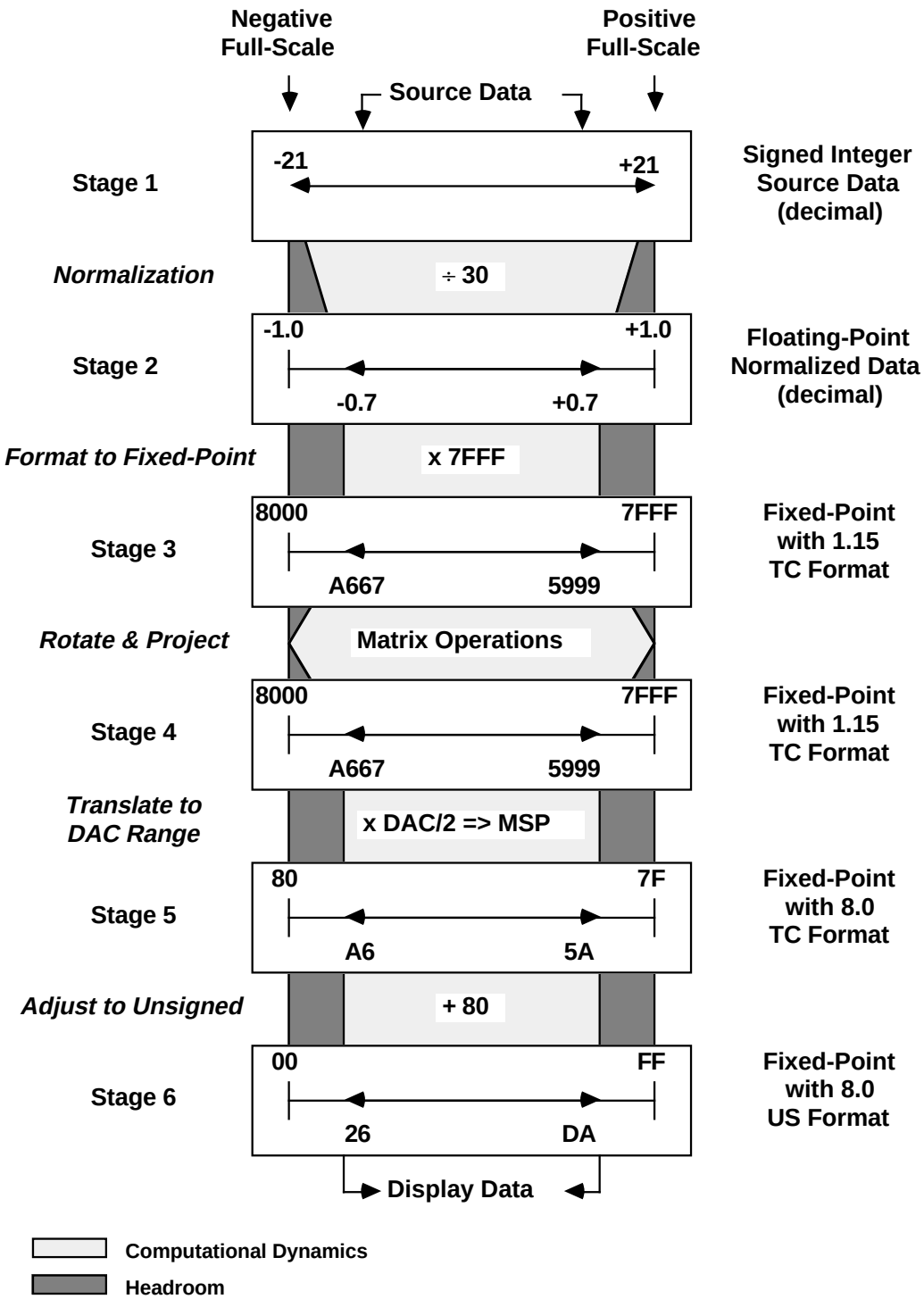


Figure 9.17 Data Format Transition Summary

Stage 5: The data has been multiplied by the half-scale DAC value (the MSP of the MAC contains the result) to translate the twos-complement data range to a corresponding full-scale range for the DAC (remember that the twos-complement format provides for only half the actual range as the unsigned format does).

Stage 6: The last step is to compensate the data for the unsigned format of the DAC by adding the half-scale DAC value to all data. This operation moves the twos-complement negative full-scale value to zero, zero to mid-scale, and positive full-scale to positive full-scale. The resulting data is what actually defines the vertices of the two-dimensional object between which the line segment drawing routines (see *Display Driver*, section 9.9) draw lines.

9.8 PROGRAM AND FILE DESCRIPTIONS

This section presents the files and programs used in the example graphics application. The flowchart in Figure 9.18 illustrates the various operations and how they interrelate; files are shown as ovals, and operations are shown as rectangles. The brief descriptions in this section give general explanations of each file.

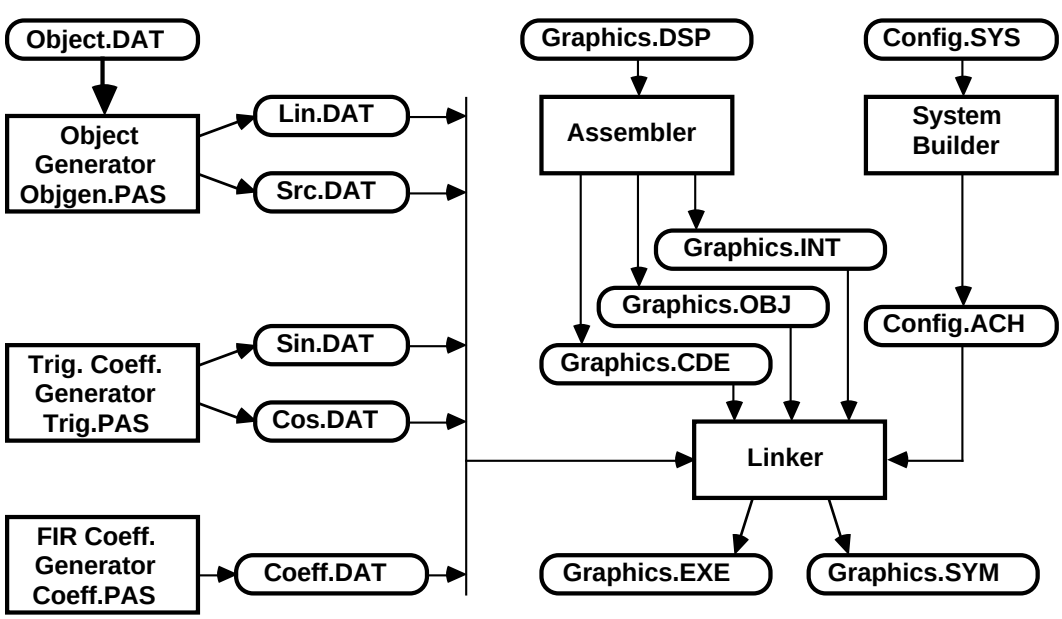


Figure 9.18 Program and File Flowchart

9.8.1 Object Generation

The OBJGEN.PAS program is a Pascal program that translates textual information representing vector coordinates, connectivity, scaling, and number of vectors (all from OBJECT.DAT) to produce hexadecimal versions of the source vector coordinates, fully normalized and formatted, and the line list (SRC.DAT and LIN.DAT, respectively). These files are used as resources in the main GRAPHICS.DSP file, through INIT directives that load the data into the arrays allocated in RAM during the linking process.

```
{Contents of the file OBJGEN.PAS}
program objgen(input,output,object,src,lin);
const
        fs=                                     32767;
var
        object, src, lin:                       text;
        scale,numpoints,points:                 integer;
        x,y,z,s:                                integer;

begin
reset(object);
rewrite(src);
read(object,scale);
read(object,numpoints);
for points:= 1 to numpoints do begin
        read(object,x,y,z,s);
        x:= trunc(x/scale*fs);
        y:= trunc(y/scale*fs);
        z:= trunc(z/scale*fs);
        writeln(src,hex(x,4,4));
        writeln(src,hex(y,4,4));
        writeln(src,hex(z,4,4));
        end;
writeln(src,'');
close(src);

rewrite(lin);
repeat
        read(object,x);
        writeln(lin,hex(x,4,4));
        until x= -1;
writeln(lin,'');
close(lin);
end.
```

Listing 9.1 Object Generation Program

```
{Contents of the file OBJECT.DAT}
30                      {Normalization Constant}
112                     {Number of Vectors comprosing object}

   -21    3   -2    1         {Vector list for foreground “2”}
   -21    5   -2    1
   -19    7   -2    1
   -13    7   -2    1
   -11    5   -2    1
   -11    1   -2    1
   -18   -5   -2    1
   -11   -5   -2    1
   -11   -7   -2    1
   -21   -7   -2    1
   -21   -5   -2    1
   -13    2   -2    1
   -13    4   -2    1
   -14    5   -2    1
   -18    5   -2    1
   -19    4   -2    1
   -19    3   -2    1

    -9    4   -2    1         {Vector list for foreground “1”}
    -9    5   -2    1
    -7    7   -2    1
    -5    7   -2    1
    -5   -7   -2    1
    -7   -7   -2    1
    -7    4   -2    1

    -3    5   -2    1         {Vector list for 1st foreground
“0”}
    -1    7   -2    1
     6    7   -2    1
     8    5   -2    1
     8   -5   -2    1
     6   -7   -2    1
    -1   -7   -2    1
    -3   -5   -2    1
    -1    4   -2    1
     0    5   -2    1
     5    5   -2    1
     6    4   -2    1
```

```
     6   -4   -2    1
     5   -5   -2    1
     0   -5   -2    1
    -1   -4   -2    1

    10    5   -2    1         {Vector list for 2nd foreground
“0”}
    12    7   -2    1
    19    7   -2    1
    21    5   -2    1
    21   -5   -2    1
    19   -7   -2    1
    12   -7   -2    1
    10   -5   -2    1
    12    4   -2    1
    13    5   -2    1
    18    5   -2    1
    19    4   -2    1
    19   -4   -2    1
    18   -5   -2    1
    13   -5   -2    1
    12   -4   -2    1

   -21    3    2    1         {Vector list for background “2”}
   -21    5    2    1
   -19    7    2    1
   -13    7    2    1
   -11    5    2    1
   -11    1    2    1
   -18   -5    2    1
   -11   -5    2    1
   -11   -7    2    1
   -21   -7    2    1
   -21   -5    2    1
   -13    2    2    1
   -13    4    2    1
   -14    5    2    1
   -18    5    2    1
   -19    4    2    1
   -19    3    2    1

    -9    4    2    1         {Vector list for background “1”}
    -9    5    2    1
```

(listing continues on next page)

```
    -7    7    2    1
    -5    7    2    1
    -5   -7    2    1
    -7   -7    2    1
    -7    4    2    1

    -3    5    2    1         {Vector list for 1st background
"0"}
    -1    7    2    1
     6    7    2    1
     8    5    2    1
     8   -5    2    1
     6   -7    2    1
    -1   -7    2    1
    -3   -5    2    1
    -1    4    2    1
     0    5    2    1
     5    5    2    1
     6    4    2    1
     6   -4    2    1
     5   -5    2    1
     0   -5    2    1
    -1   -4    2    1

    10    5    2    1         {Vector list for 2nd background
"0"}
    12    7    2    1
    19    7    2    1
    21    5    2    1
    21   -5    2    1
    19   -7    2    1
    12   -7    2    1
    10   -5    2    1
    12    4    2    1
    13    5    2    1
    18    5    2    1
    19    4    2    1
    19   -4    2    1
    18   -5    2    1
    13   -5    2    1
    12   -4    2    1

{Connection list using vector numbers, 0=penup mode}
```

Graphics 9

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 1 0 {foreground “2”}
18 19 20 21 22 23 24 18 0                       {foreground “1”}
25 26 27 28 29 30 31 32 25 0                    {foreground 1st “0”}
33 34 35 36 37 38 39 40 33 0                    {foreground 1st “0”}
41 42 43 44 45 46 47 48 41 0                  {foreground 2nd “0”}
49 50 51 52 53 54 55 56 49 0                  {foreground 2nd “0”}
57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 57 0
                                              {background “2”}
74 75 76 77 78 79 80 74 0                     {background “1”}
81 82 83 84 85 86 87 88 81 0                  {background 1st “0”}
89 90 91 92 93 94 95 96 89 0                  {background 1st “0”}
97 98 99 100 101 102 103 104 97 0             {background 2nd “0”}
105 106 107 108 109 110 111 112 105 0         {background 2nd “0”}

{connect foreground to background}
1 57 0
2 58 0
3 59 0
4 60 0
5 61 0
6 62 0
7 63 0
8 64 0
9 65 0
10 66 0
11 67 0
12 68 0
13 69 0
14 70 0
15 71 0
16 72 0
17 73 0
18 74 0
19 75 0
20 76 0
21 77 0
22 78 0
23 79 0
24 80 0
25 81 0
26 82 0
27 83 0
28 84 0
```

(listing continues on next page)

```
29 85 0
30 86 0
31 87 0
32 88 0
33 89 0
34 90 0
35 91 0
36 92 0
37 93 0
38 94 0
39 95 0
40 96 0
41 97 0
42 98 0
43 99 0
44 100 0
45 101 0
46 102 0
47 103 0
48 104 0
49 105 0
50 106 0
51 107 0
52 108 0
53 109 0
54 110 0
55 111 0
56 112 -1
```

Listing 9.2 Object Data File

9.8.2 Trigonometric Coefficient Generation

The TRIG.PAS program is a Pascal program that generates 256 uniformly spaced samples of the sine and cosine functions corresponding to the 256 possible positions (using 8-bit quantization) which the joystick may assume. These hexadecimal data files (SIN.DAT and COS.DAT) are fully normalized and formatted. The lookup tables used during the generation of the transformation matrix are initialized with data from these files.

Note that zero is positioned in the middle of the arrays to correspond with a zero rotation at the center joystick position. The data in these files is loaded during the linking process through INIT directives.

```
{Contents of the file TRIG.PAS}
program trig(input,output,sin_table,cos_table);

const
        pi=3.141592654;

var
        sin_table:                              text;
        cos_table:                              text;
        degree:                                 integer;
        arg:                                    real;
        result:                                 integer;

begin
rewrite(sin_table);
rewrite(cos_table);
for degree:= -128 to 127 do begin
        arg:= 360 * degree / 256 * pi / 180;
        result:= trunc(sin(arg) * 32767);
        writeln(sin_table, hex(result,4,4));
        result:= trunc(cos(arg) * 32767);
        writeln(cos_table, hex(result,4,4));
        end;
writeln(sin_table,'');
writeln(cos_table,'');
close(cos_table);
close(sin_table);
end.
```

Listing 9.3 Sine and Cosine Table Generation Program

9.8.3 FIR Filter Coefficient Generation

The FIR coefficient generator program COEFF.PAS generates underdamped FIR filter coefficients which are used in the filtered joystick display mode (described in *Display Driver*, section 9.9). The response of the filter was derived experimentally by plotting various exponentially damped sine waves as a function of ringing and settling time. The best response (a subjective determination) was taken as the impulse response of the desired filter; quantizing this response into 128 samples produced the FIR coefficients. The actual settling time of the filter is the number of taps divided by the frame rate, or 128÷90 ≈ 1.5 seconds. Coefficients were normalized (uniformly scaled so that their sum was approximately equal

to one) to produce a unity gain filter and then converted to 1.15 format. The hexadecimal values were stored in the COEFF.DAT file, allowing the INIT directive to load them during the linking process.

```
{Contents of the file COEFF.PAS}
PROGRAM the_function (input, output, coeff);

CONST
        pi=       3.141592654;
        cycles=   19.84;
        scale=    0.362951735;
        tc=       -15;

VAR
        x, y:     real;
        i, ypt:   integer;
        coeff:    text;

BEGIN
        rewrite(coeff);
        FOR i := 127 DOWNTO 0 DO BEGIN
          x := i * cycles * pi / 180;
          y := scale * exp(i/tc) * sin(x);
          ypt:= trunc(y * 32767 + 0.5);
          writeln(coeff, hex(ypt,4,4));
        END;
        close(coeff);
END.
```

Listing 9.4 FIR Filter Coefficient Generation Program

9.8.4 System Configuration

System configuration is mandatory for all ADSP-2100 applications. The GRAPHICS.SYS file is used by the System Builder to specify the target system configuration. The memory and peripheral mapping defined in GRAPHICS.SYS must correspond to the target system memory configuration and peripheral address decoding. This file defines RAM and ROM segments and their locations. Device interfaces are also declared using the PORT directive. Notice that data memory can be interleaved

with peripheral devices, so long as contiguous arrays are kept smaller than the allocation block size. The System Builder produces the GRAPHICS.ACH file required by the Linker.

```
{Contents of the file GRAPHICS.SYS}
.SYSTEM   graphics_config;

{allocate a 2K (0-07FF) block of PMC}
.SEG/RAM/ABS=h#0000/PM/CODE     pmc[h#0800];

{allocate a 2K (0-07FF) block of PMD}
.SEG/RAM/ABS=h#4000/PM/DATA     pmd[h#0800];

{allocate a 4K (0000-0FFF) block of DMD}
.SEG/RAM/ABS=h#0000/DM/DATA     dmd1[h#0800];

{allocate I/O map starting at DMD location h#1000}
.CONST  ioblk=h#1000;

{define x and y joystick inputs on the 4-channel adc}
.PORT/ABS=ioblk+h#0               adx;
.PORT/ABS=ioblk+h#1               ady;

{define deflection and intensity scope channels on the quad dac}
.PORT/ABS=ioblk+h#8               dax;
.PORT/ABS=ioblk+h#9               day;
.PORT/ABS=ioblk+h#A               daz;

.ENDSYS;
```

Listing 9.5 System Configuration File

9.8.5 Main Source Program

The GRAPHICS.DSP file contains the actual ADSP-2100 source code. The ADSP-2100 Assembler assembles the source code and allocates variable storage. The Assembler produces the three files (GRAPHICS.INT, GRAPHICS.OBJ, and GRAPHICS.CDE) used by the Linker. The Linker accepts the various data files mentioned above as INIT directive arguments to initialize the various RAM arrays. It produces the GRAPHICS.EXE file (executable image) and GRAPHICS.SYM file (symbol table) which can both be downloaded to a RAM-based system. If ROMs are to be burned, then an additional formatting step is required.

9 Graphics

Only 950 lines of source code are used for this example (approximately 2000 lines of executable code), although as the performance benchmarks indicate (see *Performance*, section 9.10), much of the code consists of loops. Note that the main loop takes about 94,000 instruction cycles to complete one iteration (this includes all iterations of inner loops), which corresponds to roughly a 100:1 cycles-to-line-of-code ratio. Various allocation and initialization steps are performed at startup before the program enters the main loop. The main loop consists of building a new transform, applying the transform to the object, projecting the object, and displaying the object; this loop is repeated over and over again. A manual interrupt button on the target board generates IRQ2, whose service routine sequences between the four display modes.

An interesting technique is used in the display routine: an indexed indirect jump. A jump table consists of different JUMP LABEL instructions. An index into the jump table is created and added to the base address of the jump table. Then, an indirect jump into the table is performed. The index determines which jump instruction in the jump table gets executed.

The code in Listing 9.6 below is part of the display routine. The AF register stores the index value. In this case, the index determines in which of eight possible octants the point is located. The signs (positive or negative) of the Δx and Δy values select a quadrant, and the difference in magnitude between the values ($|\Delta x| - |\Delta y|$) selects one of the two octants in that quadrant. The index picks out the jump instruction to the correct octant routine to draw the line to the point.

```
AF=PASS 0;      {init for indirect jump offset}
AX1=DM(newx);   {compute delta x}
AY1=DM(oldx);
AR=AX1-AY1;
DM(dx)=AR;
AX1=4;
IF GT AF=AX1 OR AF;  {set bit 2 if delta x is
                      positive}
AX1=DM(newy);   {compute delta y}
AY1=DM(oldy);
AR=AX1-AY1;
DM(dy)=AR;
AX1=2;          {set bit 1 if delta y is positive}
IF GT AF=AX1 OR AF;
AX1=DM(dy);     {compute |dx|-|dy|}
```

```
                AR=ABS AX1;
                AY1=AR;
                AX1=DM(dx);
                AR=ABS AX1;
                AX1=AR;
                AR=AX1-AY1;
                AX1=1;          {set bit 0 if delta x is greater}
                IF GT AF=AX1 OR AF;
                AX1=^jump_table; {add jump table base address}
                AR=AX1+AF;
                I4=AR;
                JUMP (I4);      {do the indirect jump}
jump_table:     JUMP octant6;
                JUMP octant5;
                JUMP octant3;
                JUMP octant4;
                JUMP octant7;
                JUMP octant8;
                JUMP octant2;
                JUMP octant1;
```

Listing 9.6 Jump Table

9.8.6 Data Structures

Some of the arrays and variables in the GRAPHICS.DSP source code (see the program listing at the end of the chapter) are explained in this section.

xfm_array
The nine coefficients of the 3x3 transformation are stored in this circular buffer. The circular buffer organization eliminates the need to reinitialize the transform coefficient address pointer after each vector has been transformed.

coeff
The 128 FIR coefficients associated with the filtered display mode (see next section) are stored in this array. These coefficients are applied to the joystick input samples to introduce a little ringing to the joystick response. The effect is as though the display object were resistant to changes in position.

xbuff and *ybuff*
These arrays are also used in the filtered display mode (see next section)

to store the previous joystick input samples (delay line). The FIR routine can then perform the convolution of the delayed samples with the coefficients to produce the filtered version of the joystick control signal.

sin_array and *cos_array*
These arrays hold the sine wave and cosine wave values generated by the TRIG.PAS program. During the generation of the transformation array, the various sine and cosine values dictated by the joystick inputs are fetched from these arrays.

src_array
This array stores the actual reference data describing the source object. Each new transformation always uses this source data as a starting point to avoid introducing the recursive errors that are found in systems that transform previous transforms.

line_list
The connection information describing which vectors have lines connected to them and where those lines go is stored in this array. As described below, a 0 in the line list means that no line should be drawn to the next point, nonzero values denote point numbers to which lines should be drawn (all points are numbered), and a –1 means that all lines have been drawn and thus another transform can start.

wcs_array
The "World Coordinate System" (WCS) is used in this context to refer to the transformed source data which is still in three-dimensional coordinates.

ecs_array
The "Eye Coordinate System" (ECS) is used in this context to refer to the WCS data that has been projected to a two-dimensional space and is ready for display.

xpntr and *ypntr*
These two data RAM pointers are used to keep track of the current starting position of the *xbuff* and *ybuff* arrays during the FIR filtering of the joystick input samples. These arrays are circular buffers, and therefore the starting position circulates through the buffer as new samples are brought in after each filter pass; the pointers keep track of the changing starting position within each buffer.

oldx and *oldy*
These two pointers into the *ecs_array* locate the last point which the line

drawing routine processed. This data is necessary because the line list structure only indicates the next point to which a line should be drawn, not the starting point.

newx and *newy*
These temporary variables hold the location of the current x and y coordinates in the *ecs_array* during the line drawing routine for repeated access, so as to avoid having to calculate these locations more than once.

dx and *dy*
These variables hold the differential change in x and y between the last point and the current point. They are used to determine the octant in which the new point resides.

mode
This variable keeps track of the display mode that is currently activated (see next section).

rotation
This variable, used in the *autorotate* mode (see next section), tracks the amount of rotation to apply to the object. The value of *rotation* is incremented by one after each iteration of the main loop.

xyorzflag
This variable holds a flag which is used in the *autoxyz* display mode (see next section) to indicate which axis is currently being rotated: x, y or z.

9.9 DISPLAY DRIVER

The display driver has the job of drawing the wire-frame object on the screen. The line list describes points from which lines are drawn and to which points the lines go to form the polygons that comprise the object. Zeros in the line list indicate to the line-drawing routine to jump to the next point without drawing a line (equivalent to a plotter "penup"), as in the start of a new polygon. Nonzero numbers in the line list mean to draw a line from the last point to the next point ("pendown"), which is identified by the number. A –1 value (8000_{16}) in the line list indicates that no more points remain and the drawing is complete.

Four display modes are demonstrated in this example: 1) automatic rotation about the x, y, and z axes sequentially; 2) automatic rotation about all three axes simultaneously; 3) averaged joystick control in x and y axes; and 4) filtered joystick control in x and y axes. The display is

advanced from one mode to the next by a debounced pushbutton connected to the IRQ2 interrupt input of the ADSP-2100 (see the *ir2_serve* routine in the listing).

The autorotate modes (1 and 2) rotate the object about 1.5 degrees per frame, which corresponds to the resolution of the trigonometric function tables (360°/revolution ÷ 256 entries/revolution). The averaged joystick mode sums 128 samples and then downshifts the result by seven bits to produce an average reading for each direction (x axis and y axis). Averaging reduces the potentiometer jitter associated with the joystick wiper action. The filtered mode applies an FIR filter to both x axis and y axis readings. Two 128-tap delay lines are used to track historic samples of x and y. These samples are convolved with the FIR coefficients of an exponentially underdamped sine wave. The filter parameters were selected to introduce a sense of inertial mass, complete with overshoot and ringing (see FIR filters in Chapter 5).

Before drawing each line, the program determines in which of eight octants (see Figure 9.19) the end point is relative to the first. Eight octants are used because certain aspects of the line segment routines vary uniquely for each octant. The determination of the octant in which the new point resides is made by calculating the three parameters: Δx, Δy, and $|\Delta x - \Delta y|$. The first two determine in which of four quadrants the second point resides, while the last test essentially checks whether the slope of the line is greater than or less than one, and thus determines which half of the

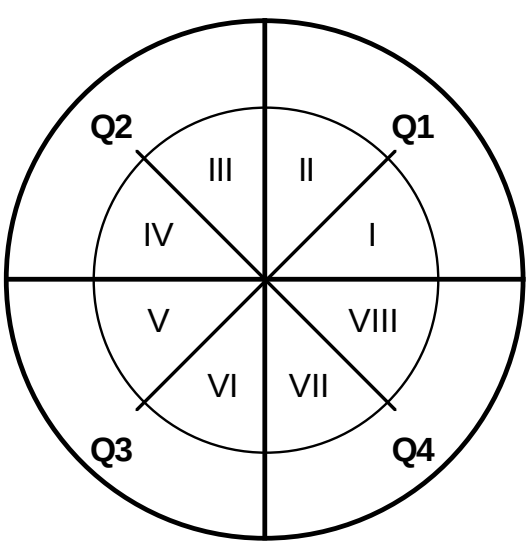


Figure 9.19 Quadrants and Octants

quadrant (which octant) the second point is in.

The actual line is drawn pixel-by-pixel using an optimized Bressenham's algorithm (see references) for generating line segments between endpoints quickly. Bressenham's algorithm is particularly attractive for this hardware implementation because it requires no division or multiplication, only simple integer arithmetic (see program listings). Depending on which octant the new point resides in, either the x axis or y axis (whichever has the faster rate of change) is either incremented or decremented (depending on the direction) a pixel at a time while the other axis is conditionally incremented or decremented. An error term that is tracked with each iteration determines whether the conditional increment or decrement is made (see program listings). Typical object images are shown in Figure 9.20, found on the following page.

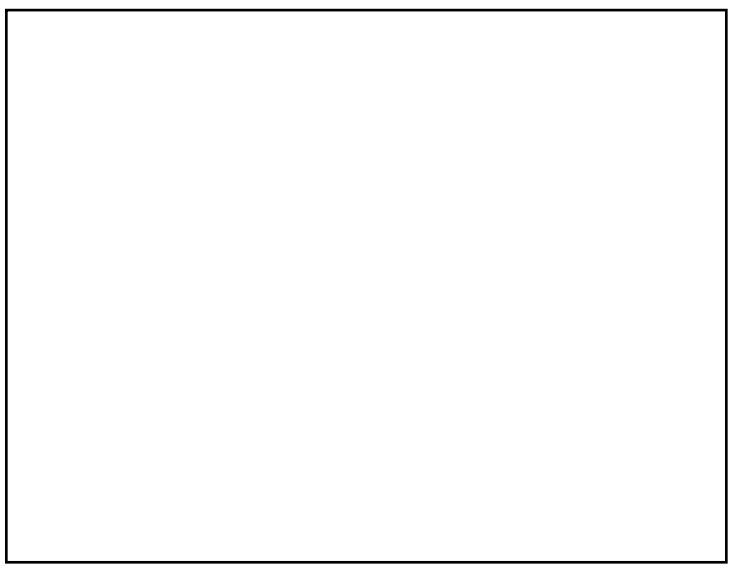
Moves to new points ("penup" moves) on the display screen are made with the beam turned off and are accomplished by writing FF_{16} to the z-axis DAC. The output of the z-axis DAC is connected to the z-axis input (or beam intensity) control, found on the back of most scopes. After the DACs are updated with the (x, y) coordinates of each new pixel, a beam-on macro (see program listings at the end of this chapter) turns on the beam for about ten cycles to make the pixel visible. The beam-on macro ends by turning the beam off again. The z-axis modulation eliminates extraneous display artifacts such as retrace and DAC transitions.

The background register set of the ADSP-2100 is used during the Bressenham algorithm because the other operations (matrix generation, transformation and projection) have many constants already stored. Having a complete set of background registers which can be instantaneously activated makes the time-consuming process of context switching (push data - process new context - pop data) obsolete.

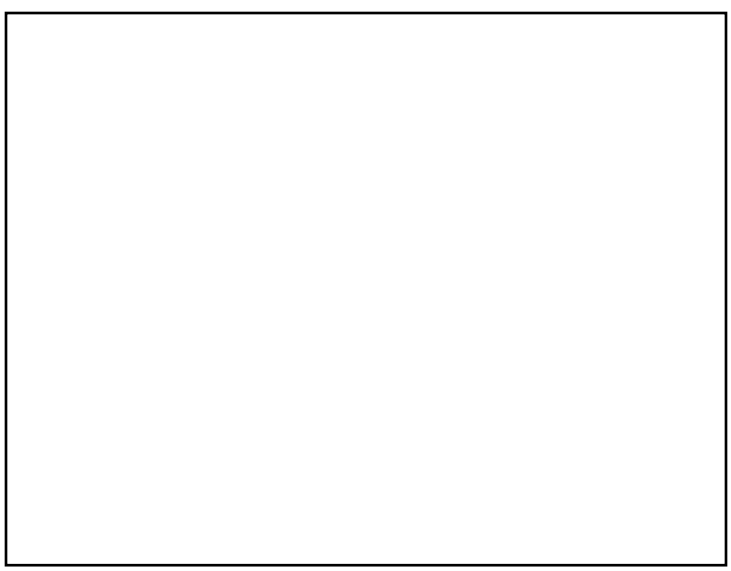
9.10 PERFORMANCE

The object in this example consists of 112 three-dimensional vectors and 170 line segments. The major program loop consists of the following functions, with execution times shown in cycles for each:

Program Phase	*Duration*
Generate or gather new position data*	
autoxyz mode:	26 cycles
autorotate mode:	20 cycles
averaged mode:	2,848 cycles
filtered mode:	330 cycles



The Object



Closeup

Figure 9.20 Typical Displays

Generate a new transform	133 cycles
Apply the transform	1,927 cycles
Project and scale the scene to two dimensions	1,595 cycles
Draw the scene	89,695 cycles

* Joystick input samples are either averaged or filtered over 128 samples, hence some modes require less time than others. See program listings for details.

The process is repeated almost 90 times a second, three times faster than necessary for a perception of continuous rotation. In other words, the ADSP-2100 could handle three times as complex an application and still convey the illusion of smooth rotation! In fact, the 90 frames/second display rate already includes performance enhancements (such as the joystick filtering and averaging modes) which are nice but unnecessary. A 33% processor utilization leaves ample processing power for extras such as hidden line removal, shading and texture mapping, shadow casting, etc. (see *Overview* at the beginning of this chapter for other ideas).

The key number in the benchmarks is the 1,927 cycles required for the entire transformation subroutine (see *doxfm* in the program listings). This measure is made from the subroutine call to the return and includes all the subroutine setup overhead instructions. A way to put this benchmark in perspective is to normalize it by the number of transforms which are actually performed: 112 1x3 vectors each multiplied by a 3x3 transform matrix produces a transform rate of 1927÷112=17.21 cycles/transform. (Although the loop is only 9 instructions long, the iterations require some overhead.) Within each 17-odd cycle transform, the following steps are performed:

- Fetch the instructions (the cache RAM is used after the first iteration),
- Fetch the nine coefficients and three vector components,
- Perform nine 16-bit multiply/accumulates,
- Store three results, and
- Maintain RAM pointers to both the transform and data arrays on each cycle.

The number of cycles in the transformation subroutine is

> [(4 inner loop instructions x 3 columns) + 5 outer loop instructions] x 112 vectors + 23 overhead instructions = 1927 cycles

as shown above. If the transform matrix size were increased from 3x3 to 4x4, the number of cycles would be

[(5 inner loop instructions x 4 columns) + 5 outer loop instructions] x 112 vectors + 23 overhead instructions = 2823 cycles

or a 46% increase.
However, the impact of this increase in the overall context is relatively insignificant. Tallying the above benchmarks for the 3x3 structure, we have about 93,373 cycles per frame (using an auto display mode), which corresponds to an 85.7 frame rate if an 8MHz processor is used. A similar tally for the 4x4 structure gives about 94,313 cycles (allowing for the increased transform time and an estimated increase for the transform build function), corresponding to an 84.8 frame rate. We may conclude that going to the full 4x4 structure (which includes the translation, zoom and perspective operations), would cost a mere 1% decrease in the frame rate.

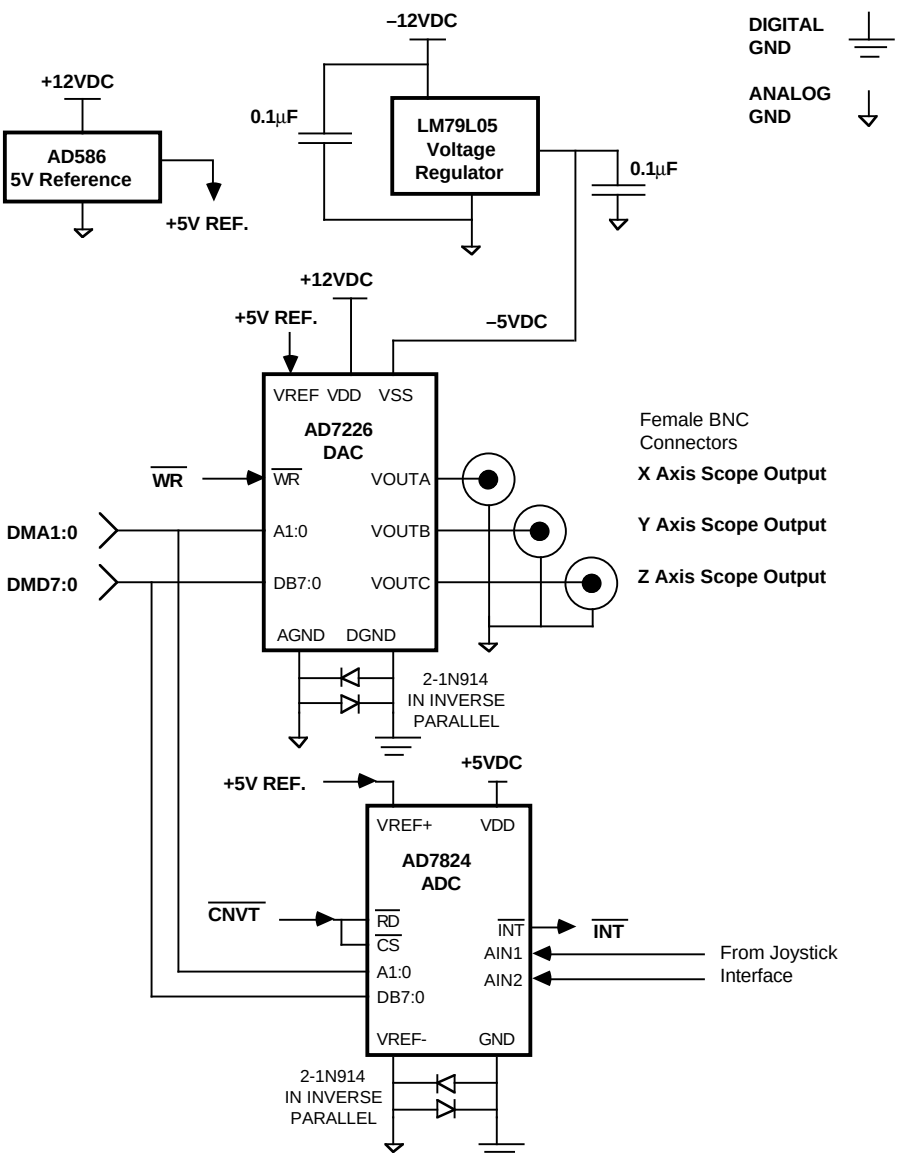
A more significant factor affecting the overall performance is the beam dwell time (see the beam-on macro in the program listings). The beam dwell is used to saturate the screen phosphor of the oscilloscope at each pixel long enough to leave a nice bright trace, but not longer. The value for the beam dwell used in the benchmark measures is 10 cycles per pixel. Because the vast majority of time is spent drawing lines, variations in the beam dwell time produce large changes in the overall frame rate. In fact, cutting the dwell time in half increases the frame rate from 85 to 106, decreasing the processor utilization from 35% to 28% while still producing an acceptable display.

The reason for the discrepancy in computation time between the averaged and filtered display modes is that filtering takes one sample from the joystick and 127 samples from the delay line maintained in data RAM, while averaging takes 128 new joystick samples of both x and y fore each frame. The filtered mode is faster because most of the data is already available in the data buffer. Resampling each value takes about 20 cycles per sample due to the relatively long ADC conversion time.

9.11 SCHEMATICS

The schematics for the graphics processor are shown in Figures 9.21 through 9.23. Figure 9.21 shows the ADC and DAC connections. The AD7824 is a four-channel, 8-bit ADC with a 2.4μs (20 cycles of the ADSP-2100) conversion time. The scope inputs are driven by an AD7226 quad 8-bit DAC. The IOSEL signal is a predecoded bank select into which both the ADC and the DAC are mapped. Reads from the IOSEL memory region come from the ADC, whereas writes to the same region go to the DAC (see the GRAPHICS.SYS system configuration file listing).

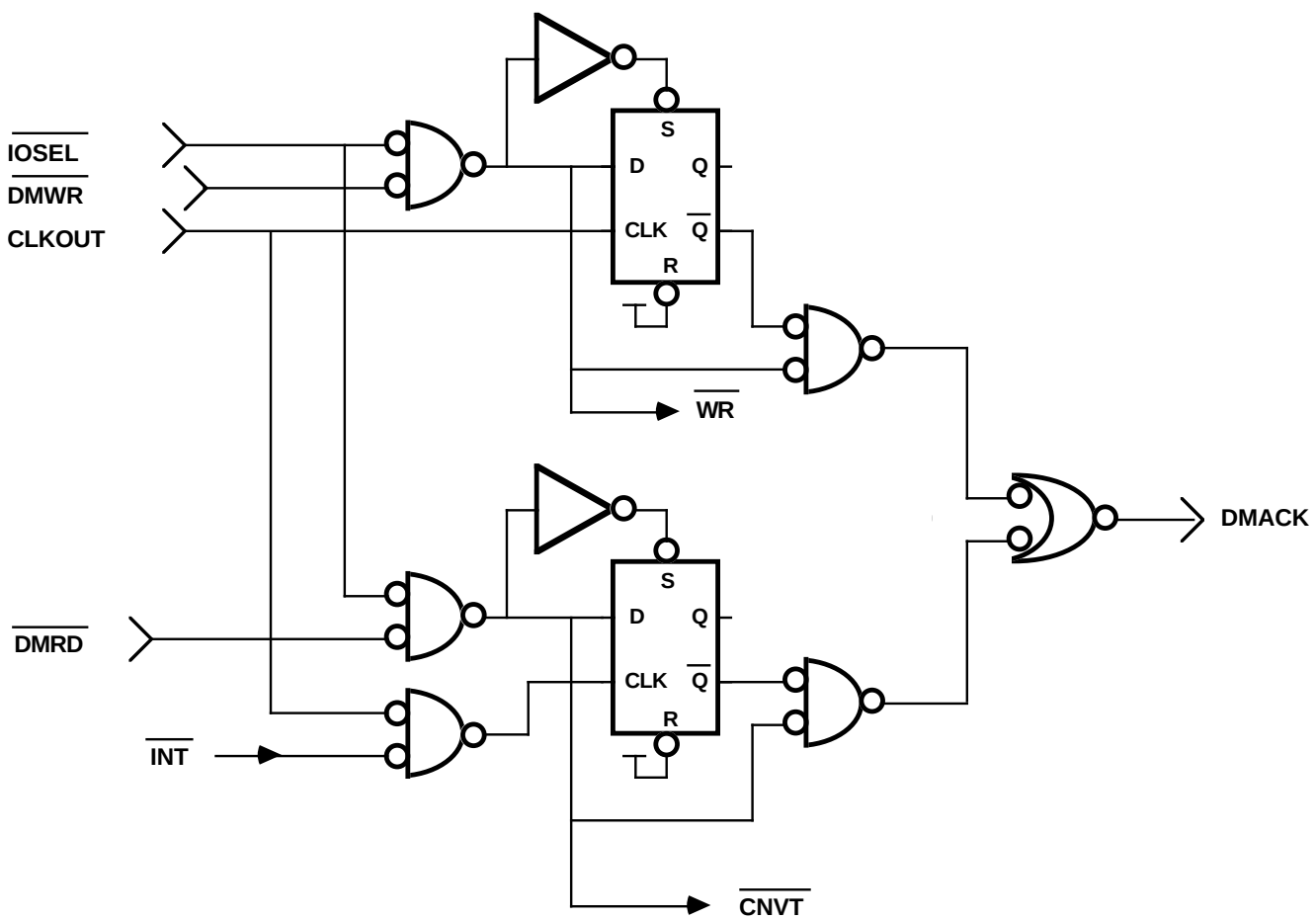
Read/write decoding and DMACK generation are provided by the circuit in Figure 9.22. Control signals from the ADSP-2100 are decoded to provide the WR and CNVT control signals for the DAC and ADC, respectively. The INT output of the ADC, which goes active LO when the



WR	write strobe (from the Read/Write Decoder)
CNVT	convert command (from the Read/Write Decoder)
INT	conversion complete handshake (to the DMACK Generator)

Figure 9.21 ADC and DAC Connections

conversion is complete, is used to produce the data memory acknowledge signal, DMACK, for the ADSP-2100. DMACK generates wait states during ADC and DAC conversion times by delaying the ADSP-2100 the appropriate amount of time.
The A/D conversion is started by the assertion of IOSEL and DMRD, which issues the CNVT signal to the ADC. The ADC converts within 2.4µs, during which time the ADSP-2100 is held in a "slow peripheral



WR write strobe (to the quad DAC)
CNVT convert command (to the quad ADC)
INT conversion complete handshake (from the ADC)

(writes delay one cycle, reads delay until INT)

Figure 9.22 Read/Write Decoder and DMACK Logic

read" mode by DMACK; wait states (nops) are executed until the conversion is complete. DAC writes also hold off DMACK to expand the write pulse width of the ADSP-2100 to meet the longer requirement of the AD7226.
The joystick interface circuit is shown in Figure 9.23. The RC4558 dual op

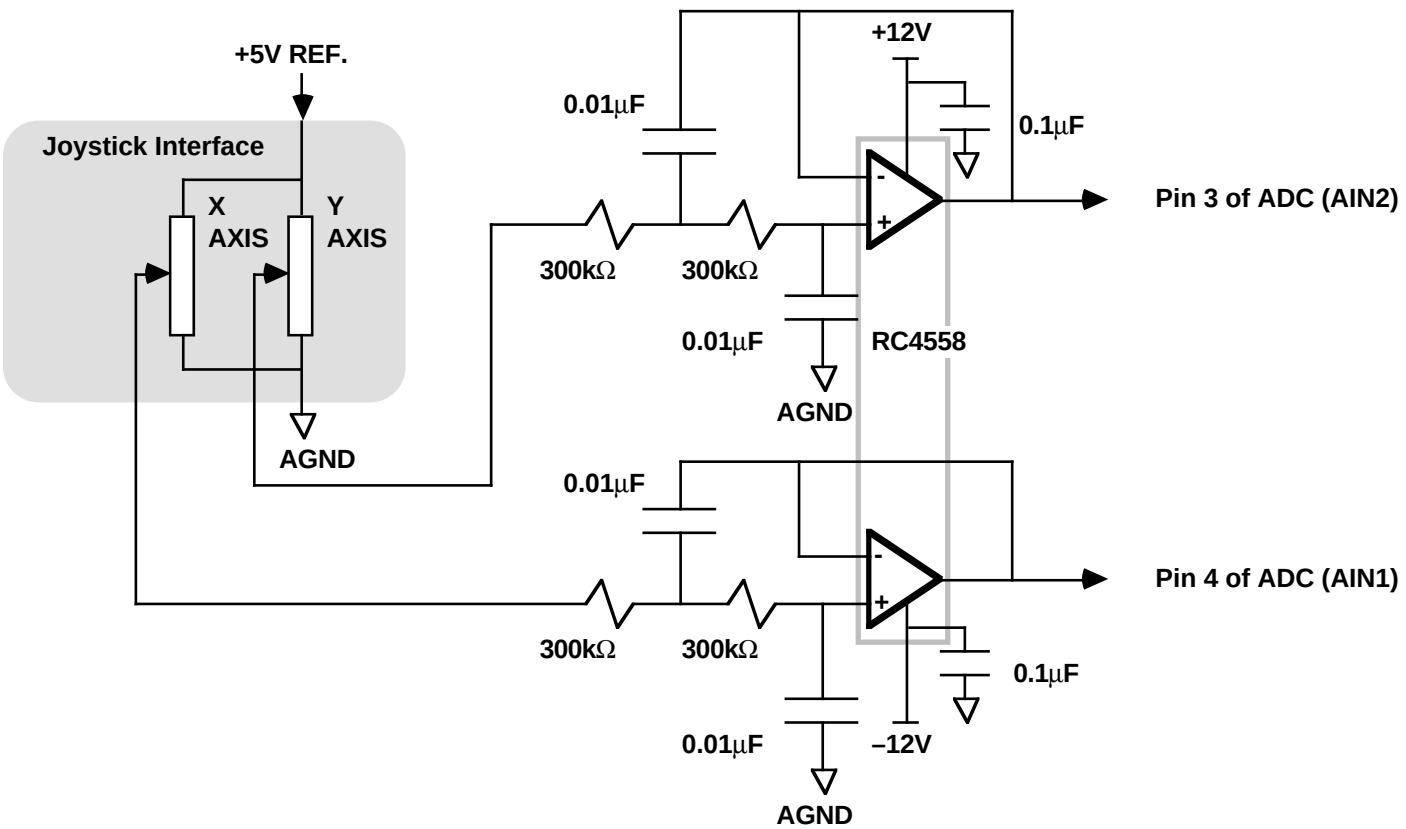


Figure 9.23 Joystick Interface

amp buffers the joystick x and y inputs to the ADC. The op amp also low-pass filters some of the joystick potentiometer noise to stabilize the display.

9.12 SUMMARY

The ADSP-2100 can be the basis of a complete, hardware-oriented application for performing graphics operations on a three-dimensional database. The example application presented in this chapter performs normalization and formatting to avoid overflow and preserve data formats through the transformation operation. It uses data structures that facilitate the object rendering by the Bressenham line segment drawing algorithm. A 3x3 rotation matrix has been derived for this application; the means for implementing translation, scaling, perspective, and zoom are also described in this chapter. Both perspective and parallel projection techniques have been discussed as well.

Software and the accompanying benchmarks show that a three-dimensional object can be rotated smoothly in a real-time display on an oscilloscope. Miscellaneous support software illustrates the basic techniques of generating source data and coefficients and getting them into the program.

The ADSP-2100 proves to be more than adequate in graphics-oriented applications. In fact, the complete application presented in this chapter uses less than a third of the available processing power of the ADSP-2100; three times as complex an application as is shown could be implemented on the ADSP-2100 while maintaining the 30Hz frame rate needed for smooth display.

9.13 REFERENCES

Foley, J. D. and Van Dam, A. 1983. *Fundamentals of Interactive Computer Graphics*. Reading, MA: Addison-Wesley Publishing Company.

Newman, William M. and Sproull, Robert F. 1983. *Principals of Interactive Computer Graphics*. New York, NY: McGraw-Hill Book Company.

Stone, Maureen C., ed. 1987. *SIGGRAPH '87 Conference Proceedings: Computer Graphics, Vol. 21, No. 4.* Anaheim, CA: Association for Computing Machinery's Special Interest Group on Computer Graphics.

9.14 PROGRAM LISTING

This section contains the complete program listing for the graphics application described in this chapter.

```
{Contents of the file GRAPHICS.DSP}
.MODULE/RAM/ABS=h#0000   graphics;

.VAR/PM/RAM/CIRC     xfm_array[h#0009];
.VAR/PM/RAM          coeff[128];

.VAR/DM/RAM/CIRC     xbuff[128];
.VAR/DM/RAM/CIRC     ybuff[128];
.VAR/DM/RAM          sin_table[h#0100];
.VAR/DM/RAM          cos_table[h#0100];
.VAR/DM/RAM          src_array[h#0150];
.VAR/DM/RAM          line_list[h#0132];
.VAR/DM/RAM          wcs_array[h#0150];
.VAR/DM/RAM          ecs_array[h#00E0];
.VAR/DM/RAM          xpntr;
.VAR/DM/RAM          ypntr;
.VAR/DM/RAM          oldx;
.VAR/DM/RAM          oldy;
.VAR/DM/RAM          newx;
.VAR/DM/RAM          newy;
.VAR/DM/RAM          dx;
.VAR/DM/RAM          dy;
.VAR/DM/RAM          mode;
.VAR/DM/RAM          rotation;
.VAR/DM/RAM          xyorzflag;

.CONST          numpoints=112;
.CONST          numpoints_2=224;
.CONST          half_scale=128;
.CONST          sin_angle=h#2121;         {sin(15 deg)*32767}
.CONST          cos_angle=h#7BA2;         {cos(15 deg)*32767}

.PORT           adx, ady, adz, dax, day, daz;

.INIT           sin_table: <sin.dat>;
.INIT           cos_table: <cos.dat>;
.INIT           src_array: <src.dat>;
.INIT           line_list: <lin.dat>;
.INIT           coeff:     <coeff.dat>;

.MACRO          nops;
                NOP; NOP; NOP; NOP; NOP; NOP;
.ENDMACRO;

.MACRO          beam_on;
.LOCAL          dwell;
```

(listing continues on next page)

```
                CNTR=10;
                DM(daz)=AX1;  {turn beam on}
                DO dwell UNTIL CE;
dwell:             NOP; {wait}
                DM(daz)=AX0;  {turn beam off}
.ENDMACRO;

{initializations...}
                RTI; RTI; JUMP ir2_serve; RTI;
{default to linear addressing}
                L0=0; L1=0; L2=0; L3=0; L4=0; L5=0; L6=0; L7=0;
                PX=0;   {clear bus exchange register}
                ENA SEC_REG;  {init secondary registers}
                MY0=1;  {used in line segment drawing}
                MY1=-1;
                AX0=h#00FF;   {to turn off the beam}
                AX1=h#0000;   {to turn it on}
                DIS SEC_REG;

                AY0=0;              {initialization value}
                DM(rotation)=AY0;   {init autorotation counter}
                DM(mode)=AY0;       {init display mode}
                DM(xyorzflag)=AY0;  {init x y or z flag}

                I0=^xbuff;    {init xbuff and ybuff delay lines}
                I1=^ybuff;
                M0=1;
                CNTR=128;     {clear samples of 128 tap filter}
                DO initloop UNTIL CE;
                   DM(I0,M0)=AY0;
initloop:          DM(I1,M0)=AY0;

                AY0=^xbuff;
                DM(xpntr)=AY0;      {init new x sample pointer}
                AY0=^ybuff;
                DM(ypntr)=AY0;      {init new y sample pointer}

                ICNTL=h#0004;       {make IRQ2 edge-sensitive}
                IMASK=h#0004;       {enable IRQ2}

{begin actual code}
mainloop:       CALL bldxfm;  {read adcs, build new transform matrix}
                CALL doxfm;   {transform source by new matrix}
                CALL doproj;  {calculate 2D projection of 3D object}
                CALL display; {drive xyz axes on scope using quad dac}
                JUMP mainloop;{display until next rotation}
```

```
{interrupt routine to sequence the display mode through
autorotate, unfiltered joystick and filtered joystick control}
ir2_serve:      AX0=DM(mode);
                AY0=0;
                AR=AX0 XOR AY0;
                IF EQ JUMP make1;
                AY0=1;
                AR=AX0 XOR AY0;
                IF EQ JUMP make2;
                AY0=2;
                AR=AX0 XOR AY0;
                IF EQ JUMP make3;
                AY0=3;
                AR=AX0 XOR AY0;
                IF EQ JUMP makez;
make1:          AX0=1;
                DM(mode)=AX0;
                RTI;
make2:          AX0=2;
                DM(mode)=AX0;
                RTI;
make3:          AX0=3;
                DM(mode)=AX0;
                RTI;
makez:          AX0=0;
                DM(mode)=AX0;
                RTI;



{BLDXFM
Module to build the master transformation matrix from the three
rotational axes components as sampled by the quad ADC.

The following registers may be overwritten by this routine,
depending upon the display mode:
    I0, I4
    L0
    M0-M3, M4
    AX0, AY0, AR
    MX0, MY0, MF, MR
}
```

(listing continues on next page)

```
bldxfm:     AX0=DM(mode);          {check out display mode}
            AY0=0;
            AR=AX0 XOR AY0;
            IF EQ JUMP autoxyz;
            AY0=1;
            AR=AX0 XOR AY0;
            IF EQ JUMP autorotate;
            AY0=2;
            AR=AX0 XOR AY0;
            IF EQ JUMP averaged;
            AY0=3;
            AR=AX0 XOR AY0;
            IF EQ JUMP filtered;
            TRAP;                  {should never get here}

autoxyz:    AY0=DM(rotation);
            AR=AY0+1;
            AY0=255;
            AR=AR AND AY0;
            DM(rotation)=AR;
            M1=AR;
            M2=AR;
            M3=AR;

            AX0=DM(xyorzflag);     {get xy or z flag}
            AY0=0;
            AR=AX0 XOR AY0;        {check for zero ==> rotate only x}
            IF EQ JUMP xonly;
            AY0=1;
            AR=AX0 XOR AY0;        {check for zero ==> rotate only y}
            IF EQ JUMP yonly;
            AY0=2;
            AR=AX0 XOR AY0;        {check for zero ==> rotate only z}
            IF EQ JUMP zonly;

xonly:      AX0=M1;                {get current x rotation}
            M2=128;                {zero out y}
            M3=128;                {zero out z}
            AY0=128;               {check current rotation}
            AR=AX0 XOR AY0;        {against zero}
            IF NE JUMP calculate;  {if not zero, keep going with x}
            AR=1;                  {otherwise, change axis of rotation}
            DM(xyorzflag)=AR;      {to y before going on}
            JUMP calculate;
```

```
yonly:       AX0=M2;                {get current y rotation}
             M1=128;                {zero out x}
             M3=128;                {zero out z}
             AY0=128;               {check current rotation}
             AR=AX0 XOR AY0;        {against zero}
             IF NE JUMP calculate;  {if not zero, keep going with y}
             AR=2;                  {otherwise, change axis of rotation}
             DM(xyorzflag)=AR;      {to z before going on}
             JUMP calculate;

zonly:       AX0=M3;                {get current z rotation}
             M1=128;                {zero out x}
             M2=128;                {zero out y}
             AY0=128;               {check current rotation}
             AR=AX0 XOR AY0;        {against zero}
             IF NE JUMP calculate;  {if not zero, keep going with z}
             AR=0;                  {otherwise, change axis of rotation}
             DM(xyorzflag)=AR;      {to x before going on}
             JUMP calculate;

autorotate:  AY0=DM(rotation);
             AR=AY0+1;
             AY0=255;
             AR=AR AND AY0;
             DM(rotation)=AR;
             M1=AR;
             M2=AR;
             M3=AR;
             JUMP calculate;

{average both x and y axis joysticks over 256 samples}
averaged:    AX1=h#00FF;            {mask bits for a/d samples}
             AY1=h#00FF;
             nops;
             AX0=DM(adx);           {get 1st sample}
             NOP;
             AR=AX0 AND AY1;
             AX0=AR;
             nops;
             AY0=DM(adx);           {get 2nd sample}
             NOP;
             AR=AX1 AND AY0;
             AY0=AR;
             AF=AX0+AY0;            {add 1st and 2nd samples}
             CNTR=126;
```

(listing continues on next page)

```
                DO xaverage UNTIL CE;
                   nops;
                   AX0=DM(adx);
                   NOP;
                   AR=AX0 AND AY1;
                   AX0=AR;
xaverage:          AF=AX0+AF;          {add in 126 more samples}
                AR=PASS AF;
                SI=AR;
                SR=LSHIFT SI BY -7 (HI);          {divide by 128 to get average}
                AX0=SR1;
                AR=AX0 AND AY1;
                M1=AR;

                nops;
                AX0=DM(ady);
                NOP;
                AR=AX0 AND AY1;
                AX0=AR;
                nops;
                AY0=DM(ady);
                NOP;
                AR=AX1 AND AY0;
                AY0=AR;
                AF=AX0+AY0;
                CNTR=126;
                DO yaverage UNTIL CE;
                   nops;
                   AX0=DM(ady);
                   NOP;
                   AR=AX0 AND AY1;
                   AX0=AR;
yaverage:          AF=AX0+AF;
                AR=PASS AF;
                SI=AR;
                SR=LSHIFT SI BY -7 (HI);
                AX0=SR1;
                AR=AX0 AND AY1;
                M2=AR;

                M3=128;                {no z on manual rotation}
                JUMP calculate;

{filter x axis}
filtered:       L0=%xbuff;             {setup circular buffer for sample buffer}
                M0=1;                  {init buffer increment}
```

```
             M4=1;                  {init coeff increment}
             I0=DM(xpntr);          {get current buffer pointer}
             I4=^coeff;             {init coeff pointer}
             nops;
             AX0=DM(adx);           {get a sample}
             NOP;
             AY0=h#00FF;
             AR=AX0 AND AY0;        {mask out upper bits}
             MX0=AR;                {load it into multiplier}
             DM(I0,M0)=MX0;         {add it to the delay line}
             MY0=PM(I4,M4);         {load 1st coeff}
             MR=0;
             CNTR=127;
             DO xfilter UNTIL CE;
xfilter:        MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
             MR=MR+MX0*MY0(SS);
             AX0=MR1;
             AR=AX0 AND AY0;        {mask out any wraparound}
             MODIFY(I0,M0);         {increment sample buffer once more}
             DM(xpntr)=I0;          {save buffer pointer}
             DM(newx)=AR;           {save filtered x}

{filter y axis}
             I0=DM(ypntr);          {get current buffer pointer}
             I4=^coeff;             {init coeff pointer}
             nops;
             AX0=DM(ady);           {get a sample}
             NOP;
             AY0=h#00FF;
             AR=AX0 AND AY0;        {mask out upper bits}
             MX0=AR;                {load it into multiplier}
             DM(I0,M0)=MX0;         {add it to the delay line}
             MY0=PM(I4,M4);         {load 1st coeff}
             MR=0;
             CNTR=127;
             DO yfilter UNTIL CE;
yfilter:        MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
             MR=MR+MX0*MY0(SS);
             AX0=MR1;
             AR=AX0 AND AY0;        {mask out any wraparound}
             MODIFY(I0,M0);         {increment sample buffer once more}
             DM(ypntr)=I0;          {save buffer pointer}
             DM(newy)=AR;           {save filtered x}

             L0=0;                  {restore linear addressing}
             M1=DM(newx);           {load filtered x value}
```

(listing continues on next page)

```
            M2=DM(newy);            {load filtered y value}
            M3=128;                 {no z on manual rotation}

calculate:  I4=^xfm_array;          {reset xfm pointer}
            M4=1;                   {to walk through xfm array}
            M0=0;                   {no modify after trig table lookup}

{calculate element xfm(11)...}
            I0=^cos_table;
            MODIFY(I0,M2);
            MX0=DM(I0,M0);          {cy}
            I0=^cos_table;
            MODIFY(I0,M3);
            MY0=DM(I0,M0);          {cz}
            MR=MX0*MY0(RND);        {cy*cz}
            PM(I4,M4)=MR1;

{calculate element xfm(21)...}
            I0=^sin_table;
            MODIFY(I0,M1);
            MX0=DM(I0,M0);          {sx}
            I0=^sin_table;
            MODIFY(I0,M2);
            MY0=DM(I0,M0);          {sy}
            MF=MX0*MY0(RND);        {sx*sy}
            I0=^cos_table;
            MODIFY(I0,M3);
            MX0=DM(I0,M0);          {cz}
            MR=MX0*MF(SS);          {sx*sy*cz}
            I0=^cos_table;
            MODIFY(I0,M1);
            MX0=DM(I0,M0);          {cx}
            I0=^sin_table;
            MODIFY(I0,M3);
            MY0=DM(I0,M0);          {sz}
            MR=MR-MX0*MY0(SS);      {sx*sy*cz-cx*sz}
            MR=MR(RND);             {loose round bug by not rounding above}
            PM(I4,M4)=MR1;

{calculate element xfm(31)...}
            I0=^cos_table;
            MODIFY(I0,M1);
            MX0=DM(I0,M0);          {cx}
            I0=^sin_table;
            MODIFY(I0,M2);
            MY0=DM(I0,M0);          {sy}
```

```
              MF=MX0*MY0(RND);       {cx*sy}
              I0=^cos_table;
              MODIFY(I0,M3);
              MX0=DM(I0,M0);         {cz}
              MR=MX0*MF(SS);         {sx*sy*cz}
              I0=^sin_table;
              MODIFY(I0,M1);
              MX0=DM(I0,M0);         {sx}
              I0=^sin_table;
              MODIFY(I0,M3);
              MY0=DM(I0,M0);         {sz}
              MR=MR+MX0*MY0(RND);    {cx*sy*cz+sx*sz}
              PM(I4,M4)=MR1;

{calculate element xfm(12)...}
              I0=^cos_table;
              MODIFY(I0,M2);
              MX0=DM(I0,M0);         {cy}
              I0=^sin_table;
              MODIFY(I0,M3);
              MY0=DM(I0,M0);         {sz}
              MR=MX0*MY0(RND);       {cy*sz}
              PM(I4,M4)=MR1;

{calculate element xfm(22)...}
              I0=^sin_table;
              MODIFY(I0,M1);
              MX0=DM(I0,M0);         {sx}
              I0=^sin_table;
              MODIFY(I0,M2);
              MY0=DM(I0,M0);         {sy}
              MF=MX0*MY0(RND);       {sx*sy}
              I0=^sin_table;
              MODIFY(I0,M3);
              MX0=DM(I0,M0);         {sz}
              MR=MX0*MF(SS);         {sx*sy*sz}
              I0=^cos_table;
              MODIFY(I0,M1);
              MX0=DM(I0,M0);         {cx}
              I0=^cos_table;
              MODIFY(I0,M3);
              MY0=DM(I0,M0);         {cz}
              MR=MR+MX0*MY0(RND);    {sx*sy*sz+cx*cz}
              PM(I4,M4)=MR1;
```

(listing continues on next page)

```
{calculate element xfm(32)...}
             I0=^cos_table;
             MODIFY(I0,M1);
             MX0=DM(I0,M0);          {cx}
             I0=^sin_table;
             MODIFY(I0,M2);
             MY0=DM(I0,M0);          {sy}
             MF=MX0*MY0(RND);        {cx*sy}
             I0=^sin_table;
             MODIFY(I0,M3);
             MX0=DM(I0,M0);          {sz}
             MR=MX0*MF(SS);          {cx*sy*sz}
             I0=^sin_table;
             MODIFY(I0,M1);
             MX0=DM(I0,M0);          {sx}
             I0=^cos_table;
             MODIFY(I0,M3);
             MY0=DM(I0,M0);          {cz}
             MR=MR-MX0*MY0(SS);      {cx*sy*sz-sx*cz}
             MR=MR(RND);             {loose round bug by not rounding above}
             PM(I4,M4)=MR1;

{calculate element xfm(13)...}
             I0=^sin_table;
             MODIFY(I0,M2);
             MR1=DM(I0,M0);          {sy}
             AR=-MR1;                {-sy}
             PM(I4,M4)=AR;

{calculate element xfm(23)...}
             I0=^sin_table;
             MODIFY(I0,M1);
             MX0=DM(I0,M0);          {sx}
             I0=^cos_table;
             MODIFY(I0,M2);
             MY0=DM(I0,M0);          {cy}
             MR=MX0*MY0(RND);        {sx*cy}
             PM(I4,M4)=MR1;

{calculate element xfm(33)...}
             I0=^cos_table;
             MODIFY(I0,M1);
             MX0=DM(I0,M0);          {cx}
             I0=^cos_table;
             MODIFY(I0,M2);
```

```
            MY0=DM(I0,M0);          {cy}
            MR=MX0*MY0(RND);        {cx*cy}
            PM(I4,M4)=MR1;

            RTS;

{DOXFM
Module to perform the actual transformation of the raw source data
by the master transformation matrix.

The transformation, xfm_array, is organized sequentially, first by columns,
then by rows:

                        | 11 12 13 |
        xfm_array =     | 21 22 23 |
                        | 31 32 33 |

i.e., xfm_array is stored in a nine location buffer in PMD as follows:

        xfm_array[1..9] = (11, 21, 31, 12, 22, 32, 13, 23, 33)

The following registers are blown away by this routine:
        I0, I1, I4
        L4
        M0, M1, M5
        MX0, MY0, MR
}

doxfm:
        I0=^src_array;              {get source dm array pointer}
        I1=^wcs_array;              {get wcs dm array pointer}
        I4=^xfm_array;              {get transform pm array pointer}

        L4=%xfm_array;   {to run modulo9 through the transform array}

        M0=2;            {to get the next xyz for each new transform}
        M2=-2;           {retard src_array pointer by 2 for each column}
        M1=1;            {general purpose for simple incrementing}
        M5=-1;           {special decrement for xfm at end of point loop}

        CNTR=numpoints;                          {transform all point vectors}
        DO points UNTIL CE;
           MX0=DM(I0,M1), MY0=PM(I4,M4);         {load 1st set of operands}
           CNTR=3;                               {do three columns}
              DO columns UNTIL CE;
                 MR=MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
                  {1st multiply clears}
```

(listing continues on next page)

```
                MR=MR+MX0*MY0(SS), MX0=DM(I0,M2), MY0=PM(I4,M4);
                MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
columns:        DM(I1,M1)=MR1;     {store the transformed component}
          MODIFY(I4,M5);           {retard xfm pointer by one}
points:   MODIFY(I0,M0);           {pick up next set of vectors}

        L4=0;

        RTS;                       {jump back}

{DOPROJ
Module to do the 3D to 2D object projection

The following registers are overwritten by this routine:
     I0-I3, M0, M1
     AX0, AY0, MX0, MX1, MY0
     MR, AR
}

doproj:
               I0=^wcs_array;      {x-component pointer}
               I1=^wcs_array+1;    {y-component pointer}
               I2=^wcs_array+2;    {z-component pointer}
               I3=^ecs_array;      {2D result pointer}

               M0=0;               {for no-modify access}
               M1=1;               {simple increment for ecs}
               M3=3;               {skip thru wcs by 3s}

               MX0=sin_angle;      {load constants}
               MX1=cos_angle;

               MY0=DM(I2,M3);      {preload z to start pipeline}

               CNTR=numpoints;
               ENA AR_SAT;         {saturate over/underflows}
          DO project UNTIL CE;
               MR=MX1*MY0(RND), AY0=DM(I0,M3);  {z*cos(angle), get x}
               AR=MR1+AY0;                      {x+z*cos(angle)}
               DM(I3,M1)=AR;                    {store x projection}

               MR=MX0*MY0(RND), AY0=DM(I1,M3);  {z*sin(angle), get y}
               AR=MR1+AY0, MY0=DM(I2,M3);       {y+z*sin(angle), get next z}
project:       DM(I3,M1)=AR;                    {store y projection}
               DIS AR_SAT;                    {restore normal ALU operation}
```

```
{in-place adjust ecs data for fullscale dac range}
                MX0=half_scale;     {load scale factor}
                AY0=half_scale;     {load axis offset}
                I3=^ecs_array;      {init 2D result pointer}
                CNTR=numpoints_2;   {twice numpoints for x&y}
                DO scale UNTIL CE;
                   MY0=DM(I3,M0);
                   MR=MX0*MY0(RND); {applly scaling}
                   AR=MR1+AY0;      {shift axis}
scale:             DM(I3,M1)=AR;    {save scaled x&y}

                RTS;

{DISPLAY
Module to display the 2D image in ecs_array on scope by writing to the dacs the
point vectors and z data (for beam on and beam off).

The line segment drawing routines for each octant are derived from the
Bressenham Algorithm which may be found in any computer graphics text.

The backround registers are used here during the actual line segment drawing
routines.

The following primary registers are blown away by this routine:
     AX0, AX1, AY0, AR, AF
     I0, I1, I4
}

display:        IMASK=0;            {disable interrupts}
                AX0=^ecs_array-2;   {-2 to adjust for 0/1 starting}
                I1=^line_list;

repeat:
                AX1=DM(I1,M1);      {load next linelist value}
                AR=PASS AX1;
                IF LT JUMP done;    {watch for -1 flag to get out}
                IF EQ JUMP moveto;  {move to new line with beam off}
                JUMP lineto;        {draw line from old to new vector}

moveto:         SI=DM(I1,M1);       {do a left shift by one to account}
                SR=LSHIFT SI BY 1 (HI);   {for xy interleave of ecs array}
                AY0=SR1;
                AR=AX0+AY0;         {add in base address of ecs-2}
                I0=AR;              {transfer to ag1}
                AR=DM(I0,M1);       {get new x-coordinate}
                DM(dax)=AR;         {write to dac with beam off}
```

(listing continues on next page)

```
                DM(oldx)=AR;        {update old x-coordinate}
                AR=DM(I0,M1);       {get new y-coordinate}
                DM(day)=AR;         {write to dac with beam off}
                DM(oldy)=AR;        {update old y-coordinate}
                JUMP repeat;

lineto:                             {get 1st point and}
                SI=AX1;             {do a left shift by one to account}
                SR=LSHIFT SI BY 1 (HI);   {for xy interleave of ecs array}
                AY0=SR1;            {with respect to line list}
                AR=AX0+AY0;         {add in base address of ecs-2}
                I0=AR;              {transfer x-addr of ecs to ag1}
                AR=DM(I0,M1);
                DM(newx)=AR;        {update new x-coordinate}
                AR=DM(I0,M1);
                DM(newy)=AR;        {update new y-coordinate}
                AF=PASS 0;          {init for indirect jump offset}
                AX1=DM(newx);       {do delta x}
                AY1=DM(oldx);
                AR=AX1-AY1;
                DM(dx)=AR;
                AX1=4;
                IF GT AF=AX1 OR AF;
                AX1=DM(newy);       {do delta y}
                AY1=DM(oldy);
                AR=AX1-AY1;
                DM(dy)=AR;
                AX1=2;
                IF GT AF=AX1 OR AF;
                AX1=DM(dy);         {do |dx|-|dy|}
                AR=ABS AX1;
                AY1=AR;
                AX1=DM(dx);
                AR=ABS AX1;
                AX1=AR;
                AR=AX1-AY1;
                AX1=1;
                IF GT AF=AX1 OR AF;
                AX1=^jump_table;    {add in jump table base address}
                AR=AX1+AF;
                I4=AR;
                JUMP (I4);          {do the indirect jump}

jump_table:     JUMP octant6;
                JUMP octant5;
                JUMP octant3;
```

```
                JUMP octant4;
                JUMP octant7;
                JUMP octant8;
                JUMP octant2;
                JUMP octant1;
{
In the following code segments:
     MY0' holds +2 (really +1, but shift makes it +2)
     MY1' holds -2 (really -1, but shift makes it -2)
     SR0' holds incr1
     SR1' holds incr2
     AY0' holds current x pixel
     AY1' holds current y pixel
     AX0' holds h#00FF to turn off the beam with
     AX1' holds h#0000 to turn on the beam
     AF   tracks the error term, d
     MX0' holds intermediate stuff
     MR gets hosed
     AR' holds intermediate stuff
}
octant1:        ENA SEC_REG;
                MX0=DM(dy);
                MR=MX0*MY0(SS);
                SR0=MR0;                {incr1 = 2dy}
                SR1=DM(dx);
                AY1=DM(dy);
                AR=SR1-AY1;
                MR=AR*MY1(SS);
                SR1=MR0;                {incr2 = -2(dx-dy)}
                AR=DM(dx);
                AF=ABS AR;
                AR=AF+1;
                CNTR=AR;                {draw line with |dx|+1 pixels}
                AY0=DM(dx);             {init AF with d = incr1-dx}
                AF=PASS AY0;
                AF=SR0-AF;
                AY0=DM(oldx);           {start at last point}
                AY1=DM(oldy);
                DO octant1loop UNTIL CE;
                   DM(dax)=AY0;        {move beam to new x along line segment}
                   DM(day)=AY1;        {move beam to new y along line segment}
                   beam_on;
                   AR=AY0+1;            {increment x}
                   AY0=AR;
                   AR=PASS AF;          {check sign of error term, d}
                   PUSH STS;            {save status for 'else' test}
```

(listing continues on next page)

```
                  IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                  POP STS;
                  IF LT JUMP octant1loop;
                  AR=AY1+1;            {'else' clause...}
                  AY1=AR;              {increment y}
                  AF=SR1+AF;           {d = d+incr2}
octant1loop:      NOP;
               JUMP update;

octant2:       ENA SEC_REG;
               MX0=DM(dx);
               MR=MX0*MY0(SS);
               SR0=MR0;                {incr1 = 2dx}
               SR1=DM(dx);
               AY1=DM(dy);
               AR=SR1-AY1;
               MR=AR*MY0(SS);
               SR1=MR0;                {incr2 = 2(dx-dy)}
               AR=DM(dy);
               AF=ABS AR;
               AR=AF+1;
               CNTR=AR;                {draw line with |dy|+1 pixels}
               AY0=DM(dy);             {init AF with d = incr1-dy}
               AF=PASS AY0;
               AF=SR0-AF;
               AY0=DM(oldx);           {start at last point}
               AY1=DM(oldy);
               DO octant2loop UNTIL CE;
                  DM(dax)=AY0;        {move beam to new x along line segment}
                  DM(day)=AY1;        {move beam to new y along line segment}
                  beam_on;
                  AR=AY1+1;            {increment y}
                  AY1=AR;
                  AR=PASS AF;          {check sign of error term, d}
                  PUSH STS;            {save status for 'else' test}
                  IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                  POP STS;
                  IF LT JUMP octant2loop;
                  AR=AY0+1;            {'else' clause...}
                  AY0=AR;              {increment x}
                  AF=SR1+AF;           {d = d+incr2}
octant2loop:      NOP;
               JUMP update;

octant3:       ENA SEC_REG;
               MX0=DM(dx);
```

```
                MR=MX0*MY1(SS);
                SR0=MR0;                {incr1 = -2dx}
                SR1=DM(dx);
                AY1=DM(dy);
                AR=SR1+AY1;
                MR=AR*MY1(SS);
                SR1=MR0;                {incr2 = -2(dx+dy)}
                AR=DM(dy);
                AF=ABS AR;
                AR=AF+1;
                CNTR=AR;                {draw line with |dy|+1 pixels}
                AY0=DM(dy);             {init AF with d = incr1-dy}
                AF=PASS AY0;
                AF=SR0-AF;
                AY0=DM(oldx);           {start at last point}
                AY1=DM(oldy);
                DO octant3loop UNTIL CE;
                   DM(dax)=AY0;        {move beam to new x along line segment}
                   DM(day)=AY1;        {move beam to new y along line segment}
                   beam_on;
                   AR=AY1+1;            {increment y}
                   AY1=AR;
                   AR=PASS AF;          {check sign of error term, d}
                   PUSH STS;            {save status for 'else' test}
                   IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                   POP STS;
                   IF LT JUMP octant3loop;
                   AR=AY0-1;            {'else' clause...}
                   AY0=AR;              {decrement x}
                   AF=SR1+AF;           {d = d+incr2}
octant3loop:       NOP;
                JUMP update;

octant4:        ENA SEC_REG;
                MX0=DM(dy);
                MR=MX0*MY0(SS);
                SR0=MR0;                {incr1 = 2dy}
                SR1=DM(dx);
                AY1=DM(dy);
                AR=SR1+AY1;
                MR=AR*MY0(SS);
                SR1=MR0;                {incr2 = 2(dx+dy)}
                AR=DM(dx);
                AF=ABS AR;
                AR=AF+1;
                CNTR=AR;                {draw line with |dx|+1 pixels}
```

(listing continues on next page)

```
                AY0=DM(dx);             {init AF with d = incr1+dx}
                AF=PASS AY0;
                AF=SR0+AF;
                AY0=DM(oldx);           {start at last point}
                AY1=DM(oldy);
                DO octant4loop UNTIL CE;
                   DM(dax)=AY0;        {move beam to new x along line segment}
                   DM(day)=AY1;        {move beam to new y along line segment}
                   beam_on;
                   AR=AY0-1;            {decrement x}
                   AY0=AR;
                   AR=PASS AF;          {check sign of error term, d}
                   PUSH STS;            {save status for 'else' test}
                   IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                   POP STS;
                   IF LT JUMP octant4loop;
                   AR=AY1+1;            {'else' clause...}
                   AY1=AR;              {increment y}
                   AF=SR1+AF;           {d = d+incr2}
octant4loop:       NOP;
                JUMP update;

octant5:          ENA SEC_REG;
                MX0=DM(dy);
                MR=MX0*MY1(SS);
                SR0=MR0;                {incr1 = -2dy}
                SR1=DM(dx);
                AY1=DM(dy);
                AR=SR1-AY1;
                MR=AR*MY0(SS);
                SR1=MR0;                {incr2 = 2(dx-dy)}
                AR=DM(dx);
                AF=ABS AR;
                AR=AF+1;
                CNTR=AR;                {draw line with |dx|+1 pixels}
                AY0=DM(dx);             {init AF with d = incr1+dx}
                AF=PASS AY0;
                AF=SR0+AF;
                AY0=DM(oldx);           {start at last point}
                AY1=DM(oldy);
                DO octant5loop UNTIL CE;
                   DM(dax)=AY0;        {move beam to new x along line segment}
                   DM(day)=AY1;        {move beam to new y along line segment}
                   beam_on;
                   AR=AY0-1;            {decrement x}
                   AY0=AR;
```

```
                  AR=PASS AF;          {check sign of error term, d}
                  PUSH STS;            {save status for 'else' test}
                  IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                  POP STS;
                  IF LT JUMP octant5loop;
                  AR=AY1-1;            {'else' clause...}
                  AY1=AR;              {decrement y}
                  AF=SR1+AF;           {d = d+incr2}
octant5loop:      NOP;
               JUMP update;

octant6:       ENA SEC_REG;
               MX0=DM(dx);
               MR=MX0*MY1(SS);
               SR0=MR0;               {incr1 = -2dx}
               SR1=DM(dx);
               AY1=DM(dy);
               AR=SR1-AY1;
               MR=AR*MY1(SS);
               SR1=MR0;               {incr2 = -2(dx-dy)}
               AR=DM(dy);
               AF=ABS AR;
               AR=AF+1;
               CNTR=AR;               {draw line with |dy|+1 pixels}
               AY0=DM(dy);            {init AF with d = incr1+dy}
               AF=PASS AY0;
               AF=SR0+AF;
               AY0=DM(oldx);          {start at last point}
               AY1=DM(oldy);
               DO octant6loop UNTIL CE;
                  DM(dax)=AY0;       {move beam to new x along line segment}
                  DM(day)=AY1;       {move beam to new y along line segment}
                  beam_on;
                  AR=AY1-1;           {decrement y}
                  AY1=AR;
                  AR=PASS AF;         {check sign of error term, d}
                  PUSH STS;           {save status for 'else' test}
                  IF LT AF=SR0+AF;    {'if then' clause: d = d+incr1}
                  POP STS;
                  IF LT JUMP octant6loop;
                  AR=AY0-1;           {'else' clause...}
                  AY0=AR;             {decrement x}
                  AF=SR1+AF;          {d = d+incr2}
octant6loop:      NOP;
               JUMP update;
```

(listing continues on next page)

```
octant7:        ENA SEC_REG;
                MX0=DM(dx);
                MR=MX0*MY0(SS);
                SR0=MR0;                {incr1 = 2dx}
                SR1=DM(dx);
                AY1=DM(dy);
                AR=SR1+AY1;
                MR=AR*MY0(SS);
                SR1=MR0;                {incr2 = 2(dx+dy)}
                AR=DM(dy);
                AF=ABS AR;
                AR=AF+1;
                CNTR=AR;                {draw line with |dy|+1 pixels}
                AY0=DM(dy);             {init AF with d = incr1+dy}
                AF=PASS AY0;
                AF=SR0+AF;
                AY0=DM(oldx);           {start at last point}
                AY1=DM(oldy);
                DO octant7loop UNTIL CE;
                   DM(dax)=AY0;        {move beam to new x along line segment}
                   DM(day)=AY1;        {move beam to new y along line segment}
                   beam_on;
                   AR=AY1-1;            {decrement y}
                   AY1=AR;
                   AR=PASS AF;          {check sign of error term, d}
                   PUSH STS;            {save status for 'else' test}
                   IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                   POP STS;
                   IF LT JUMP octant7loop;
                   AR=AY0+1;            {'else' clause...}
                   AY0=AR;              {increment x}
                   AF=SR1+AF;           {d = d+incr2}
octant7loop:       NOP;
                JUMP update;

octant8:        ENA SEC_REG;
                MX0=DM(dy);
                MR=MX0*MY1(SS);
                SR0=MR0;                {incr1 = -2dy}
                SR1=DM(dx);
                AY1=DM(dy);
                AR=SR1+AY1;
                MR=AR*MY1(SS);
                SR1=MR0;                {incr2 = -2(dx+dy)}
                AR=DM(dx);
                AF=ABS AR;
```

```
                AR=AF+1;
                CNTR=AR;                {draw line with |dx|+1 pixels}
                AY0=DM(dx);             {init AF with d = incr1-dx}
                AF=PASS AY0;
                AF=SR0-AF;
                AY0=DM(oldx);           {start at last point}
                AY1=DM(oldy);
                DO octant8loop UNTIL CE;
                   DM(dax)=AY0;        {move beam to new x along line segment}
                   DM(day)=AY1;        {move beam to new y along line segment}
                   beam_on;
                   AR=AY0+1;            {increment x}
                   AY0=AR;
                   AR=PASS AF;          {check sign of error term, d}
                   PUSH STS;            {save status for 'else' test}
                   IF LT AF=SR0+AF;     {'if then' clause: d = d+incr1}
                   POP STS;
                   IF LT JUMP octant8loop;
                   AR=AY1-1;            {'else' clause...}
                   AY1=AR;              {decrement y}
                   AF=SR1+AF;           {d = d+incr2}
octant8loop:         NOP;
                JUMP update;

update:         AY0=DM(newx);           {update old with last new point}
                DM(oldx)=AY0;
                AY1=DM(newy);
                DM(oldy)=AY1;
                DIS SEC_REG;
                JUMP repeat;

done:           IMASK=h#0004;           {re-enable irq2}
                RTS;

.ENDMOD;
```

9 Graphics