Transforms 10

This chapter discusses the XGL Transform object and its operators. It includes information on the following topics:

- Concepts of Transform objects
- Transform operators:
  - Interface-level representations
  - Basic utilities
  - Concatenation
- Examples using Transform operators

Introduction to the Transform Object

A transformation is a linear mapping of geometric data from one coordinate space to another. XGL stores the information for this mapping in a Transform object and applies the mapping to the *x*, *y*, and *z* coordinates of geometric primitives. Transforms are the basic building blocks for sequentially manipulating geometry. They *model* a complicated scene from simple geometric primitives, *view* the scene with a particular orientation and perspective, and place the image in a window. These operations are associated with the view model, which is discussed in Chapter 11, "View Model."

At the interface level, application programmers can think of Transforms as having matrix representations. XGL treats position vectors as homogeneous row vectors, which normally have a homogeneous component *w* set to 1. In the

2D case, the matrix is 3 × 3. This matrix can represent any sequence of translations, rotations, and scalings from one space to another in the following way:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

This representation extends to 3D, for which the matrix is 4 × 4.

Transforms can operate in 2D or 3D space. 2D Transforms can be represented in integer, floating point, or double precision format; 3D Transforms can be represented in floating point or double precision format. The dimension and data type of a Transform are attributes of the Transform object.

Creating a Transform Object

A new Transform object is created with the `xgl_object_create()` operator using a parameter `type` of `XGL_TRANS` and the parameter `desc` of `NULL`. The `xgl_object_create()` operator initializes the new Transform to identity and sets the initial values of the specified attributes.

Transform Operators

The XGL library provides operators for manipulating Transforms, including rotation, translation, and scaling operators. These operators can concatenate a rotation, translation, or scale transformation with an existing transformation. XGL also supplies other utility operators to compute the values of transformations.

Note – The Transforms used as parameters in the Transform operators must be created with `xgl_object_create()` before being used by the Transform operators. When a Transform is no longer needed by an application, it should be destroyed using `xgl_object_destroy()` to free the resources associated with it.

Rotation Operator

The matrices for rotation by angle θ in the right-handed sense have the following form. Note that 2D rotations are counterclockwise when the x-axis points right and the y-axis points up. They are clockwise when the x-axis points right and the y-axis points down.

For 2D: $\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

For 3D about the z-axis: $\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

For 3D about the y-axis: $\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

For 3D about the x-axis: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The Rotation operator replaces a Transform with a Rotation Transform constructed from the given parameters. It can also preconcatenate or postconcatenate a Rotation Transform with the given Transform. When a matrix is postconcatenated, the matrix is multiplied on the right of the original Transform matrix because XGL represents points as row vectors. The Rotation operator is:

```
void xgl_transform_rotate (
    Xgl_trans          trans,
    double             angle,
    Xgl_axis           axis,
    Xgl_trans_update   update);
```

Scaling Operator

The matrices for scaling have the following form:

For 2D: $\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

For 3D: $\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The Scale operator replaces a Transform with a Scale Transform constructed from the given parameters. It can also preconcatenate or postconcatenate a Scale Transform with the given Transform. Scale factors set to zero may result in errors. The Scale operator is defined as:

```
void xgl_transform_scale(
    Xgl_trans          trans,
    Xgl_pt             *scale_factors,
    Xgl_trans_update   update);
```

Translation Operator

The matrices for translation have the following form:

For 2D: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$

For 3D: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$

The Translation operator replaces an existing Transform with a Translation Transform constructed from the given parameters. It can also preconcatenate or postconcatenate a Translation Transform to the given Transform. The Translation operator is defined as:

```
void xgl_transform_translate(
    Xgl_trans          trans,
    Xgl_pt             *offset,
    Xgl_trans_update   update);
```

Reading and Writing Transforms

Applications may need to read or write the data in a Transform if the formulation of the transformation is more complicated than a sequence of translations, rotations, and scalings. XGL provides operators for reading a Transform into an array and writing an array into a Transform. For 3D, the arrays are 4×4. For 2D, the arrays are 3×2 because the last column is always $[0,0,1]^T$.

The `xgl_transform_read()` operator reads a Transform's matrix into an array supplied by the application:

```
void_xgl_transform_read (
    Xgl_trans          src_trans,
    void               *matrix);
```

The `xgl_transform_write_specific()` operator writes an array into a Transform object and specifies the membership of the matrix to the matrix groups:

```
void xgl_transform_write_specific (
    Xgl_trans          dest_trans,
    void               *matrix,
    Xgl_trans_member   membership);
```

A matrix group is a particular type of matrix with a well-defined set of properties. An application can give XGL hints about the matrix it is writing into a Transform by specifying the matrix's membership in a matrix group.

Membership in a matrix group can significantly improve the performance of most operations involving transformations. The matrix groups are listed in Table 10-1. For the form of each matrix, see the `xgl_transform_write_specific()` manual page in the *XGL Reference Manual*.

Table 10-1 Transform Memberships in Matrix Groups

Value	Description
XGL_TRANS_MEMBER_IDENTITY	Identity matrix.
XGL_TRANS_MEMBER_TRANSLATION	Translation matrix.
XGL_TRANS_MEMBER_SCALE	Scale matrix.
XGL_TRANS_MEMBER_ROTATION	Rotation matrix.
XGL_TRANS_MEMBER_WINDOW	A window matrix is a combined scale and translation matrix.
XGL_TRANS_MEMBER_SHEAR_SCALE	Shear-scale matrix. This matrix type is 3D only.
XGL_TRANS_MEMBER_LENGTH_PRESERV	A length-preserving matrix can translate, rotate, and reflect about an axis. These operations preserve the distance between any two points and the angle between any two intersecting lines.
XGL_TRANS_MEMBER_ANGLE_PRESERV	An angle-preserving matrix can translate, rotate, and reflect about an axis, and scale by an amount which is uniform in all directions. These operations preserve the angle between any two intersecting lines.
XGL_TRANS_MEMBER_AFFINE	An affine matrix can translate, rotate, reflect, scale anisotropically, and shear. Lines remain straight, and parallel lines remain parallel, but the angle between intersecting lines can change.
XGL_TRANS_MEMBER_LIM_PERSPECTIVE	Limited perspective matrix. This matrix type is 3D only.

The `xgl_transform_write()` operator writes an array supplied by the application into the Transform's matrix.

```
void xgl_transform_write (
    Xgl_trans          dest_trans,
    void               *matrix);
```

Other Transform Operators

The following transformation operators provide utility functions. For operators that require a source Transform and a destination Transform, the two Transforms can be the same.

Copy

The Copy operator copies the contents of the matrix associated with the Source Transform to the matrix associated with the Destination Transform. 2D and 3D Transforms can be mixed in this operation.

```
void xgl_transform_copy (
    Xgl_trans          dest_trans,
    Xgl_trans          src_trans);
```

Identity

The Identity operator sets the Transform's matrix to the Identity. An Identity Transform has no effect on geometric data passing through it. The Identity operator is defined as:

```
void xgl_transform_identity (Xgl_trans trans);
```

Invert

The Invert operator inverts a Transform's matrix. The inverse of a matrix, when multiplied with the original matrix, produces the identity matrix. A Transform may be singular, having no unique inverse. The operator normally returns *dest_tran*s, but will return `NULL` if the Transform is singular. The numerical accuracy depends on the Transform's data type. The Invert operator is:

```
Xgl_trans xgl_transform_invert(
    Xgl_trans          dest_trans,
    Xgl_trans          src_trans);
```

Note – Do not mix 2D and 3D Transforms with the `xgl_transform_invert()` operator.

Multiply

The Multiply operator multiplies two Transforms. All Transforms passed to this operator must have the same dimension. The Multiply operator is defined as:

```
void xgl_transform_multiply (
    Xgl_trans          dest_tr,
    Xgl_trans          left_src_tr,
    Xgl_trans          right_src_tr);
```

Note – Do not mix 2D and 3D Transforms with the `xgl_transform_multiply()` operator.

Transform Point

The Transform Point operator transforms a single point by treating the point as a row vector and multiplying it by the Transform's matrix. The result is stored back in the memory allocated for the point, overwriting the original point.

```
void xgl_transform_point (
    Xgl_trans          trans,
    Xgl_pt             *point);
```

Transform Point List

The Transform Point List operator transforms a list of points, treating each point as a row vector and multiplying it by the Transform's matrix. This operator is defined as:

```
void xgl_transform_point_list(
    Xgl_trans          trans,
    Xgl_pt_list        *src_pt_list,
    void               *dest_pts);
```

Transpose

The Transpose operator transposes a Transform's matrix. The *i*th column of a transposed matrix is the *i*th row of the original matrix. The *i*th row of a transposed matrix is the *i*th column of the original matrix.

This operator may be useful if an application uses `xgl_transform_write()` and treats position vectors as column vectors instead of row vectors like XGL. Another application of this operator is with rotation Transforms: the inverse is its transpose. 2D and 3D Transforms can be mixed using this operator. The Transpose operator is:

```
void xgl_transform_transpose (
    Xgl_trans          dest_trans,
    Xgl_trans          src_trans);
```

10

Transform Attributes

The following Transform attributes configure Transforms.

`XGL_TRANS_DATA_TYPE`
This attribute defines the data type of the matrix associated with a Transform. For 2D Transforms, the data type can be integer, floating point, or double precision. For 3D Transforms, the data type can be floating point or double precision.

`XGL_TRANS_DIMENSION`
This attribute defines whether the space in which the Transform operates is 2D or 3D. The attribute can be set to `XGL_TRANS_2D`, in which case the Transform operates in two-dimensional space, or `XGL_TRANS_3D`, so that the Transform operates in three-dimensional space.

Examples Using the Transform Object

This section explains the example 2D and 3D XGL programs at the end of this chapter. The first example shows an untransformed triangle in 2D space. In the next three examples, the triangle is translated, rotated, and scaled. The final example shows a method for creating a View Transform to give an oblique parallel projection of a cube.

The transformation example programs are to be used within the framework of the general utility program `ex_utils.c`. Each example is a fragment of a larger program. The complete program includes `ex_utils.c` and `tran_main.c`, both of which are listed in Appendix B, as well as all the examples listed in this section. To compile the complete program, type `make tran` in the example program directory. The compiled program allows you to look at all the transformation example programs.

2D Transform Examples

The example program `ex_utils.c` creates a 2D Context. This Context has a pipeline consisting of Transforms and structures for describing clipping and mapping. The pipeline implements the XGL view model. Creation of a 2D Context automatically causes creation of all the Transforms in the Context's view model. Therefore, the example programs need not explicitly create the Context's View Transform.

The 2D example programs are alike, except in the particular Transform operator each invokes. The first example, which draws the triangle in its original position, obtains a handle to the 2D Context's View Transform. When the handle, `view_trans`, is the target of a Transform operator, the Context becomes aware of the change to its pipeline. In the first example, `view_trans` is a parameter of `xgl_transform_identity()`, and sets the View Transform to the Identity matrix.

The example program set the orientation of Virtual Device Coordinates (VDC) so the x-axis points right and the y-axis points up. (Chapter 11, "View Model," describes this in more detail.) This places the origin in the lower left corner of the window. Since the View Transform is the Identity, the vertices defining the triangle are measured in pixels.

The second, third, and fourth examples translate, rotate, and scale the triangle, respectively. The translation and scaling operators take a vector that specifies the appropriate information. The translation example translates the triangle by 100 in the x-direction and 10 in the y direction. The scale example scales the triangle by 2 in both the *x*- and y directions.

In the rotation example, the rotation operator takes an angle in radians. A positive angle results in a counterclockwise rotation here because the y-axis points up in this example. XGL ignores the third parameter for 2D rotations. In 3D, a positive angle gives a right-handed rotation around the axis specified by the third parameter.

The translation, rotation, and scaling operators all take a parameter that specifies how to combine the requested transformation with the target Transform. The second, third, and fourth examples all replace the contents of the target Transform to perform a pure translation, rotation, and scaling, respectively. An application can also concatenate a requested transformation with the target Transform. Preconcatenation and postconcatenation are both supported. Concatenation of two Transforms produces a Transform that gives the same result as when applying the two original Transforms in sequence.

Displaying the Original Geometry

This code fragment displays the triangle before the application of any Transforms. Figure 10-1 shows the result.

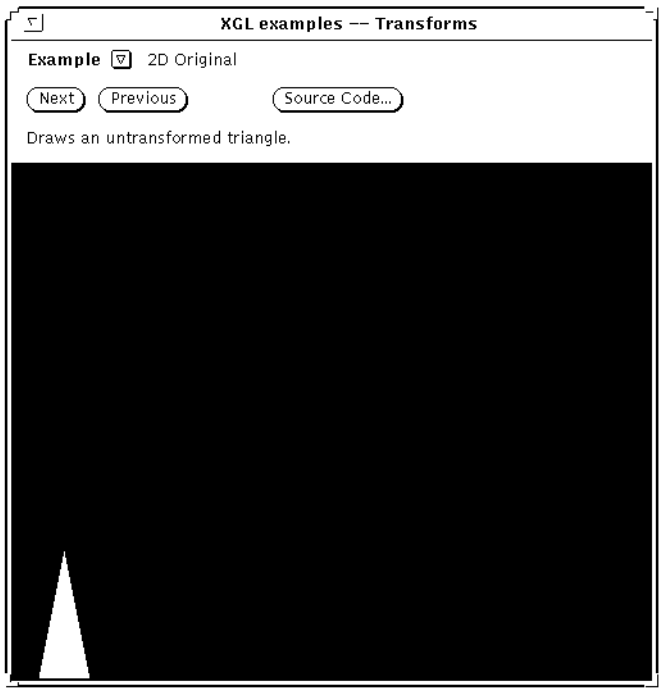

Figure 10-1 Output of `tran_2d_orig.c`

Code Example 10-1 Geometry Before Transformations

```
/*
 * tran_2d_orig.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern Xgl_pt_list   pl_2d;
```

```
void
tran_2d_orig (Xgl_object   ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;

    color = cyan_color;

    xgl_object_set (ctx,
                XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                XGL_CTX_SURF_FRONT_COLOR, &color,
                0);

    /* Get the view transform and set it to the identity */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_identity (view_trans);

    /* Draw a 2D polygon */
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

Translating the Geometry

This function illustrates how to set up a Transform object for translation. Figure 10-2 illustrates the output.

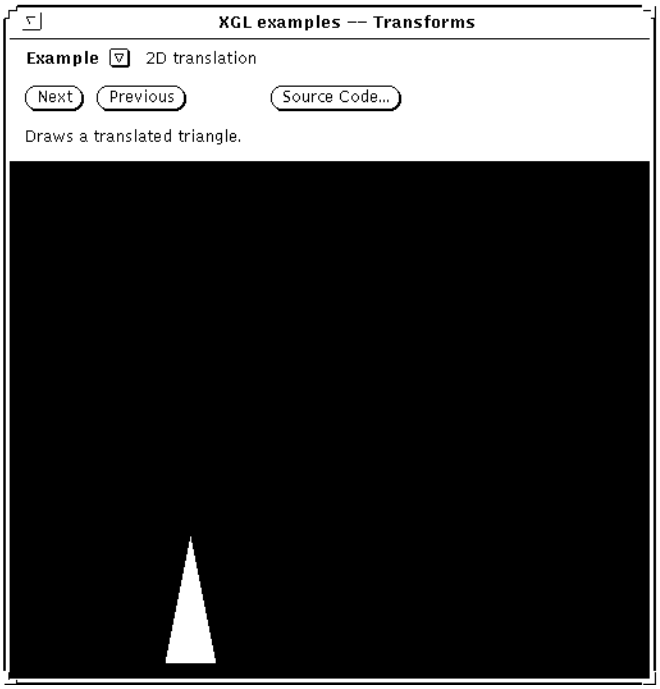

Figure 10-2 Output of `tran_2d_transl.c`

Code Example 10-2 Translation Example

```
/*
 * tran_2d_transl.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern Xgl_pt_list   pl_2d;
```

```
void
tran_2d_translate (Xgl_object    ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;
    Xgl_pt               pt;
    Xgl_pt_f2d           pt_f2d;

    color = cyan_color;

    xgl_object_set (ctx,
                XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                XGL_CTX_SURF_FRONT_COLOR, &color,
                0);

    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

    pt.pt_type = XGL_PT_F2D;
    pt.pt.f2d = &pt_f2d;
    pt_f2d.x = 100.0;
    pt_f2d.y = 10.0;

    xgl_transform_translate (view_trans, &pt, XGL_TRANS_REPLACE);

    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

Rotating the Geometry

This code fragment illustrates how to set up a Transform object for rotation. Figure 10-3 shows the rotated object.

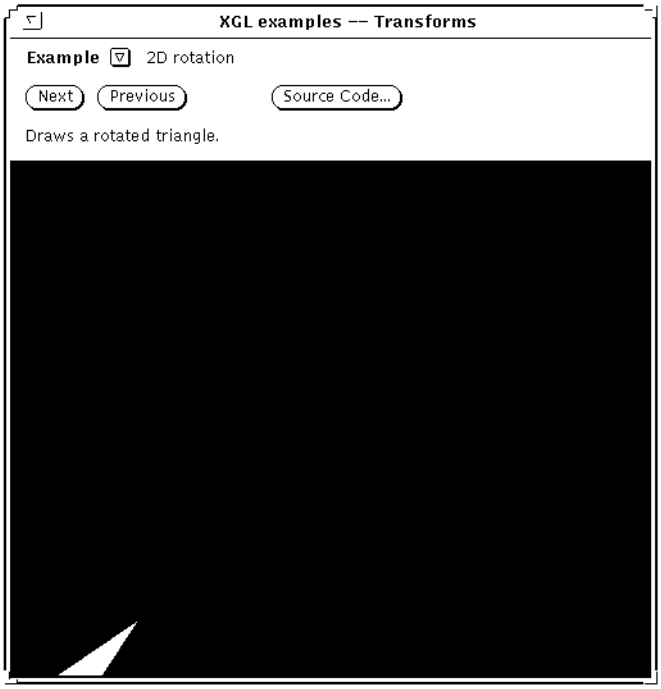

Figure 10-3 Output of `tran_2d_rot.c`

Code Example 10-3 Rotation Example

```
/*
 * tran_2d_rot.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>
#include "ex.h"

#define    PI      3.141592654
```

```
extern Xgl_pt_list   pl_2d;

void
tran_2d_rotate (Xgl_object   ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;

    color = cyan_color;

    xgl_object_set (ctx,
                XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                XGL_CTX_SURF_FRONT_COLOR, &color,
                0);

    /* Get the view transform and set it to a rotation transform */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_rotate (view_trans, -PI / 4.0, XGL_AXIS_Z,
                 XGL_TRANS_REPLACE);

    /* Draw a 2D polygon */
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

Scaling the Geometry

This example illustrates how to set up the Transform object for scaling. Figure 10-4 shows the scaled object.

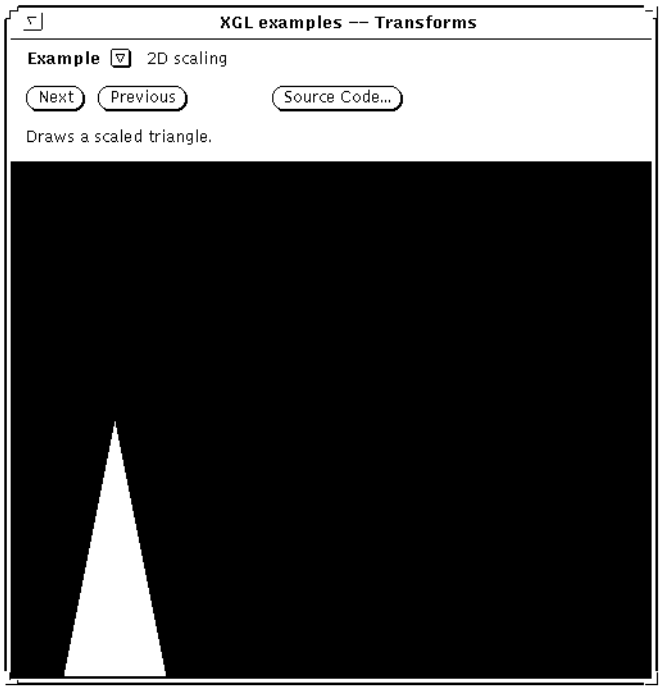

Figure 10-4 Output of `tran_2d_scale.c`

Code Example 10-4 Scale Example

```
/*
 * tran_2d_scale.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>
#include "ex.h"

extern Xgl_pt_list   pl_2d;
```

```
void
tran_2d_scale (Xgl_object     ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;
    Xgl_pt               pt;
    Xgl_pt_f2d           pt_f2d;

    color = cyan_color;

    xgl_object_set (ctx,
                XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                XGL_CTX_SURF_FRONT_COLOR, &color,
                0);

    /* Get the view transform and set it to a scale transform */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

    pt.pt_type = XGL_PT_F2D;
    pt.pt.f2d = &pt_f2d;
    pt_f2d.x = 2.0;
    pt_f2d.y = 2.0;

    xgl_transform_scale (view_trans, &pt, XGL_TRANS_REPLACE);

    /* Draw a 2D polygon */
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

10

3D Transform Example

The final example, `tran_3d.c`, illustrates a method for constructing a 3D View Transform. It begins by creating a temporary Transform to hold intermediate results. By default, a new Transform holds floating-point values and handles 3D points. An application can change the dimension or data type, or both, of a Transform at creation or at any time afterward. However, the Transforms associated with a 3D Context must be 3D; those associated with a 2D Context must be 2D.

The example program applies a sequence of one translation and three rotations, with the operators appearing in the 2D example programs. The first operator replaces the contents of the Transform with a translation, and the succeeding rotations follow the effect of the translation. Postconcatenation of each rotation to the preceding sequence of transformations successively builds a single Transform that gives the cumulative effect.

Having completed the basic operations for building the desired view transformation, the example gets a handle to the 3D Context's View Transform and copies the result of the preceding calculations to this Transform in the Context's view model. The example then sets some additional view model attributes (described in Chapter 11, "View Model") and draws a projection of a wireframe cube using the view transformation just calculated. The final step is destruction of the Transform created at the beginning for holding intermediate results. Figure 10-5 on page 293 shows the wireframe cube.

An explanation of the mathematical principles used in these examples can be found in *Principles of Interactive Computer Graphics*, pages 54-55 and 348-351[1]. The XGL convention for positive angles corresponding to rotations in the right-handed sense, however, is opposite to the convention proposed by Newman and Sproull.

1. Newman, William M. and Robert F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, N.Y., 2nd edition, 1979.

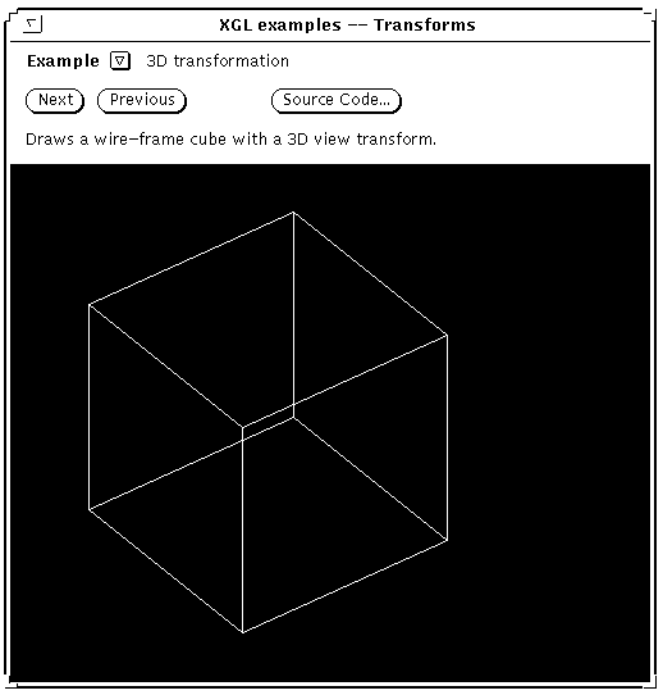

Figure 10-5 Output of `tran_3d.c`

Code Example 10-5 3D Transform Example

```
/*
 * tran_3d.c
 */
#include <math.h>
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

#define    PI  3.141592654

extern Xgl_object         sys_st;
extern Xgl_pt_list        pl_3d[];
```

```
void
tran_3d (Xgl_object      ctx)
{
    Xgl_color            color;
    Xgl_object           trans;
    Xgl_object           view_trans;
    Xgl_pt               pt;
    Xgl_pt_f3d           pt_f3d;
    Xgl_bounds_d3d       vdc_window;

    /* Create a 3D floating-point transform */
    trans = xgl_object_create(sys_st, XGL_TRANS, NULL, NULL);

    /* Set up view transform */

    /* Translate the origin of VDC to (6.0, 8.0, 7.5) in WC */
    pt.pt_type = XGL_PT_F3D;
    pt.pt.f3d = &pt_f3d;
    pt_f3d.x = -6.0;
    pt_f3d.y = -8.0;
    pt_f3d.z = -7.5;
    xgl_transform_translate (trans, &pt, XGL_TRANS_REPLACE);

    /* Swing the y-axis of VDC so that it is vertical in WC */
    xgl_transform_rotate (trans, -PI / 2, XGL_AXIS_X,
                      XGL_TRANS_POSTCONCAT);

    /* Swivel the z-axis of VDC away from WC's z-axis */
    xgl_transform_rotate (trans, atan (pt_f3d.x / pt_f3d.y) + PI,
                      XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);

    /* Tip VDC so that its z-axis points away from WC's origin */
    xgl_transform_rotate (trans, atan (-pt_f3d.z /
                         sqrt (pt_f3d.x * pt_f3d.x +
                               pt_f3d.y * pt_f3d.y)),
                    XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

    /* Set context's view transform */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_copy (view_trans, trans);

    /* Set line color and VDC parameters */
    color = yellow_color;
    vdc_window.xmin = -2.0;
    vdc_window.xmax = 2.0;
    vdc_window.ymin = -2.0;
```

```
    vdc_window.ymax = 2.0;
    vdc_window.zmin = -14.5;
    vdc_window.zmax = -10.5;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &color,
                    XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                    XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
                    XGL_CTX_VDC_WINDOW, &vdc_window,
                    0);

    /* Draw wireframe representation of data */
    xgl_multipolyline (ctx, NULL, 4, pl_3d);

    /* Clean up */
    xgl_object_destroy (trans);
}
```

10