View Model 11

This chapter discusses the XGL viewing pipeline. It includes information on the following topics:

- Coordinate systems and clipping
- 2D and 3D Context view model attributes
- 2D and 3D viewing and clipping pipelines

Introduction to the XGL Viewing Pipeline

The *viewing pipeline* is a series of transformations and clip operations applied to a graphics primitive before rendering. It transforms geometric coordinate values into a range suitable for display on the output device. The XGL viewing pipeline moves the geometric data from an application's space to rendering space via a set of transformation and clipping attributes. XGL has distinct and separate viewing pipelines for 2D and 3D. The XGL 2D viewing model is an affine[1] model. The XGL 3D viewing model is a homogeneous[2] model.

1. *Affine* means a geometric mapping that preserves the straightness of lines and the parallelism of parallel lines but possibly alters the distances between points or the angles between lines.

2. A *homogeneous* representation is a mapping of a point in n-dimensional ordinary coordinates to $(n+1)$-dimensional homogeneous coordinates. Given a point in three-dimensional ordinary coordinates $[x, y, z]$, the homogeneous representation is $[wx, wy, wz, w]$ for any $w \neq 0$. The inverse mapping from homogeneous coordinates to ordinary coordinates — via division by w — is called *projection*.

The XGL viewing model consists of four coordinate systems: Model Coordinates (MC), World Coordinates (WC), Virtual Device Coordinates (VDC), and Device Coordinates (DC). The Transforms presented in Chapter 10, "Transforms", map the geometry between any two sequential coordinate systems in the pipeline. Each Context has its own set of Transforms to implement these mappings. All Transforms default to the *Identity Transform* (which makes no change to the coordinate values); applications can ignore the Transforms, unless needed.

Clipping is performed in two places in the 2D pipeline and three places in the 3D pipeline. In both the 2D and 3D pipeline, view clipping can take place in VDC coordinates, and clipping to DC limits is always applied to ensure that the drawing fits into the display area (either a window or a non-raster device). In the 3D pipeline, model clipping can also be defined in Model Coordinates.

Coordinate Systems

Coordinate systems define the geometric environment in which the graphic primitive coordinate values are defined. Coordinate systems associated with the 2D pipeline are two-dimensional. Coordinate systems associated with the 3D pipeline are three-dimensional. An application can pass only 2D points to a 2D pipeline and only 3D points to a 3D pipeline. The viewing pipeline consists of the following coordinate systems:

Model Coordinates (Local and Global MCs)

Model Coordinates give the frame of reference in which a geometric model is defined. For example, a model of a bolt might be defined in inches or millimeters, whereas a ship might be defined in feet or meters. The application controls the placement of each Model Coordinate System into the next coordinate system of the pipeline, World Coordinates, with a transformation called the Model Transform.

World Coordinates (WC)

The World Coordinate System is common to all geometry for a particular view. XGL performs lighting in this space when 3D applications require illumination. The individual objects in the image to be rendered are represented here in proper perspective relative to each other. Viewing, or the application of a View Transform, maps the scene into a device-independent Virtual Device Coordinate System.

Virtual Device Coordinates (VDC)

XGL defines a Virtual Device Coordinate System to isolate applications from physical Device Coordinates, which might be different for different devices such as raster devices or file output devices. XGL performs view clipping in this space when applications require it.

Device Coordinates (DC)

Device Coordinates are the actual coordinates of the Device. If the Device is a Raster, the units of Device Coordinates are in pixels. In Device Coordinates, (0,0,0) is the upper left corner, where the axes are oriented right for positive x, down for positive y, and away for positive z.

The Context object includes information that specifies the mapping and clipping of geometric data between coordinate spaces. The Local Model Transform maps simple geometric primitives comprising a piece of the object to a global modeling space where the object is assembled. The Global Model Transform maps objects into the final scene in world coordinate space. The View Transform determines the application's viewing direction, orientation, and perspective.

Taken together, the Local Model Transform and the Global Model Transform map coordinate values from Model Coordinates to World Coordinates. The View Transform maps coordinate values from World Coordinates to Virtual Device Coordinates. The mapping from Virtual Device Coordinates to Device Coordinates is automatic and is computed by comparing the VDC and DC ranges (this mapping is limited to scaling and translation). It is also possible to map coordinates values directly from Model Coordinates to Device Coordinates using the MC-to-DC Transform. The MC-to-DC Transform results from the concatenation of the Model Transform, the View Transform, the VDC Transform, and the Device's DC orientation.

Figure 11-1 on page 300 and Figure 11-2 on page 301 illustrate the 2D and 3D view models, showing the transformations and the coordinate systems in the XGL viewing pipeline.

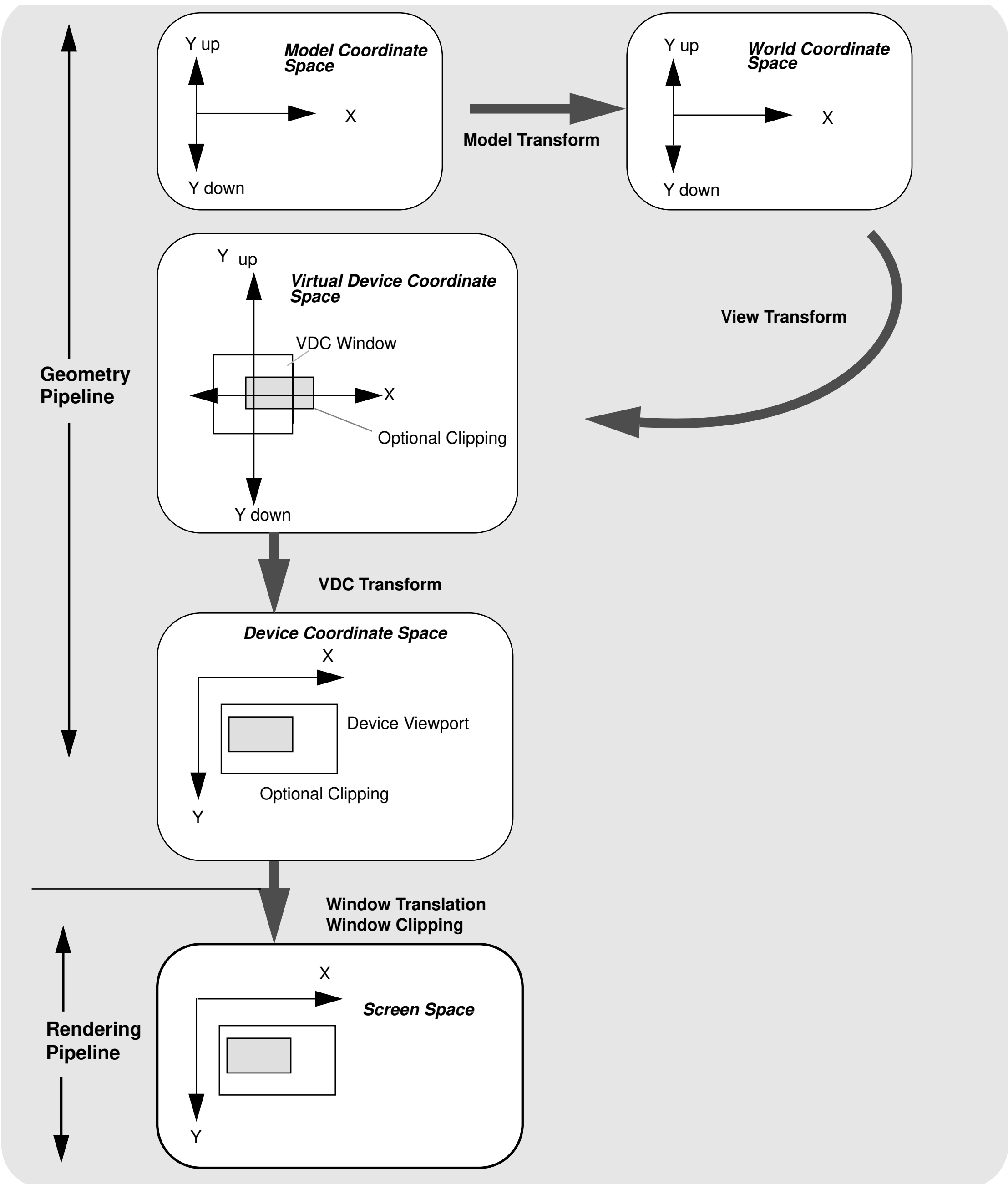

Figure 11-1 2D View Model

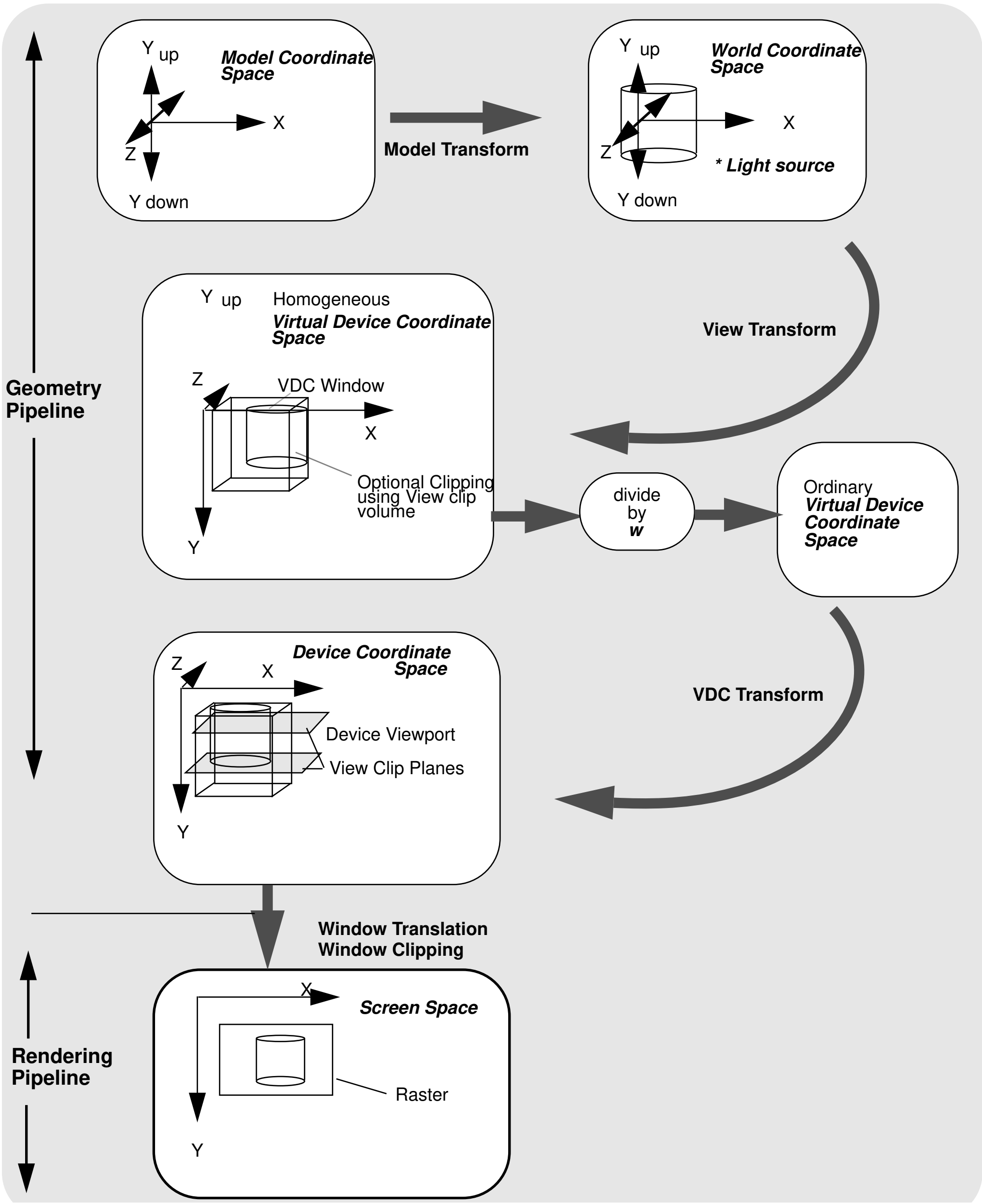

Figure 11-2 3D View Model

11

How to Use Transforms with the View Model

When an application creates a 2D or 3D Context object, the Context object creates a default transformation pipeline, which includes a Local Model Transform, a Global Model Transform, a Model Transform, an MC-to-DC Transform, and a View Transform. As part of the object initialization, the Context object sets these Transform objects to Identity. If a 3D Context is created, a Normal Transform is also created. An application can get the handles of these Transforms from the Context object. Although the Model Transform, the MC-to-DC Transform, and the Normal Transform cannot be modified, the application can change the values of the Local Model Transform, the Global Model Transform, and the View Transform using the Transform operators. Resetting the View Transform is shown in the code fragment below.

```
Xgl_trans      view_trans;

/* 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);
```

To get the handles to the Context's Transform objects, use these attributes:

Table 11-1 Getting Context Transform Handles

To get a handle to this Transform:	Use this attribute:
Local Model Transform	XGL_CTX_LOCAL_MODEL_TRANS
Global Model Transform	XGL_CTX_GLOBAL_MODEL_TRANS
Model Transform	XGL_CTX_MODEL_TRANS (RO)
View Transform	XGL_CTX_VIEW_TRANS
MC-to-DC Transform	XGL_CTX_MC_TO_DC_TRANS (RO)
Normal Transform (3D only)	XGL_3D_CTX_NORMAL_TRANS (RO)

When the application gets the handle to a Transform object, both the application and the Context, which owns the Transform objects, have a handle to the Transform in the Context's pipeline. If this handle is the destination parameter of a Transform operator, the Context automatically knows about the change to one of its Transforms. Note that if application-created Transforms are used as parameters by the Transform operators, the Transforms must be created before they can be used by the Transform operators. When a Transform is no longer needed in an application, it should be destroyed using the `xgl_object_destroy()` operator to free its resources.

Alternatively, an application can create a new Transform object for the Local Model Transform, the Global Model Transform, or the View Transform and set the Context attribute to the new Transform. However, the recommended way of modifying a transformation is to retrieve the relevant transformation matrix from the Context object and modify it rather than creating a new Transform object.

Note – If you set the `XGL_CTX_LOCAL_MODEL_TRANS`, `XGL_CTX_GLOBAL_MODEL_TRANS`, or `XGL_CTX_VIEW_TRANS` attributes to a Transform whose handle is retrieved from the same Context or a different Context, the original attribute and the current attribute use the same Transform. Changing the Transform of one Context attribute has the same effect on the other Context attribute. Be sure that you are aware of the consequences when you share a Transform in this manner. Do not attempt to set these three attributes to a handle of a Transform retrieved from `XGL_CTX_MODEL_TRANS`, `XGL_CTX_MC_TO_DC_TRANS`, or `XGL_3D_CTX_NORMAL_TRANS`, because these Transforms are read-only.

Using XGL 2D and 3D Viewing

This section provides information on using the XGL viewing pipeline.

Model Coordinate Space and Model Transforms

An application passes graphical primitives to XGL in Model Coordinates (MC). The application can define any number of Model Coordinate Systems. Their relation to each other is determined by their locations and orientations with respect to World Coordinates (WC). The application controls the placement of each Model Coordinate System in World Coordinates by a transformation called the Model Transform.

Applications using the 2D or 3D pipeline can specify the Model Transform in two levels with a Local Model Transform and a Global Model Transform. The Model Transform is the Local Model Transform concatenated with the succeeding Global Model Transform. The Model Transform is affected by changes to either the Local Model Transform or the Global Model Transform. When two Transforms are concatenated to form a single Transform, the single Transform behaves as both Transforms applied in succession. These Transforms are given by the `XGL_CTX_LOCAL_MODEL_TRANS`, `XGL_CTX_GLOBAL_MODEL_TRANS`, and `XGL_CTX_MODEL_TRANS` attributes.

Applications written for 3D Contexts also have a Normal Transform, which maps normal vectors from Model Coordinates to World Coordinates. The Normal Transform is the inverse of the Model Transform.

`XGL_CTX_MODEL_TRANS` and `XGL_3D_CTX_NORMAL_TRANS` are read-only attributes for the Transform objects that map position or normal vectors from Model Coordinates to World Coordinates. Both can be changed only by setting the Local or Global Model Transforms.

Note – Evaluation of the Model Transform and the Normal Transform is deferred until either XGL needs the Transform internally or the application explicitly gets it. Before attempting to use the Model Transform or the Normal Transform, call `xgl_object_get()` to get `XGL_CTX_MODEL_TRANS` or `XGL_3D_CTX_NORMAL_TRANS` to ensure that XGL evaluates the Transform if it requires updating.

The default value for the Model Transforms and the Normal Transform is the Identity Transform.

World Coordinate Space and the View Transform

Viewing projects a scene in World Coordinates onto an image plane. This step is similar to selecting a camera location, a direction and orientation for holding the camera, and the focal length of the lens, which determines the field of view and perspective. The transformation pipeline accomplishes this by passing the geometrical data assembled into World Coordinates through the View Transform into the Virtual Device Coordinate Space. The `XGL_CTX_VIEW_TRANS` attribute is the View Transform object. When the application defines the View Transform, it must take into account both the view clipping bounds defined with the attribute `XGL_CTX_VIEW_CLIP_BOUNDS` and the orientation of VDC space as defined by `XGL_CTX_VDC_ORIENTATION`. The View Transform, view clip bounds, and VDC orientation collectively determine the generated image in VDC space.

For 3D, the View Transform may include a *perspective transformation*, which would change an implicit w component of the geometric data from 1.0 to some other value. XGL allows in VDC both homogeneous coordinates (x, y, z, w) and ordinary coordinates (x, y, z). The geometry is projected into ordinary 3D space after the view clipping (specified by the `XGL_CTX_CLIP_PLANES` and `XGL_CTX_VIEW_CLIP_BOUNDS` attributes).

The default value for the View Transform is the Identity Transform.

Virtual Device Coordinate Space and the VDC Transform

Virtual Device Coordinate space isolates viewing from physical Device Coordinates. Each Device may have its own units, so the VDC-to-DC mapping allows much of the pipeline — from Model Coordinates to Virtual Device Coordinates — to remain device-independent.

The VDC Transform performs this mapping, but it is not directly accessible as an attribute to the application. The VDC Transform maps a rectangular region in VDC known as a *window* to a rectangular region in DC known as a *viewport*. The VDC Transform defaults to the Identity Transform, so VDC is identical to DC. It allows scaling and translation, but not rotation.

XGL provides implicit window resizing policies (although the application must first inform XGL that a window resize event occurred). The window resizing policies automatically recalculate the VDC Transform when a user resizes a Window Raster. The action taken depends on the values of three attributes: `XGL_CTX_VDC_MAP`, `XGL_CTX_VDC_WINDOW`, and `XGL_CTX_DC_VIEWPORT`.

- `XGL_CTX_VDC_MAP` determines the VDC mapping method (which is also known as the window resizing policy, although it applies to Devices other than Window Rasters). It also specifies whether the application provides the mapping from VDC-to-DC, or if XGL calculates the mapping. It has four values:

  `XGL_VDC_MAP_OFF`
  This value gives the application complete control over the VDC-to-DC mapping. The application must specify `XGL_CTX_VDC_WINDOW` and `XGL_CTX_DC_VIEWPORT`. This means that the application must know the dimensions of the Device (which a user could change if the Device is a Window Raster), and the value of the attribute

XGL_CTX_VDC_ORIENTATION, which provides the correct regions for mapping. (See the next section on VDC Orientation for more information).

XGL_VDC_MAP_ALL
This value allows the application to provide the value of XGL_CTX_VDC_WINDOW. XGL sets the value of XGL_CTX_DC_VIEWPORT to the Device's bounding box. This method ensures use of the full range of Device Coordinate Space; however, it may result in the image being scaled by different increments along the *x*- and *y*-axes.

XGL_VDC_MAP_ASPECT
Like the previous value, this value lets the application provide the value XGL_CTX_VDC_WINDOW, but XGL determines the value of XGL_CTX_DC_VIEWPORT. This method maps the VDC window to DC so that the region is as large as possible while still maintaining the aspect ratio of width to height. The top left corner of the XGL_CTX_VDC_WINDOW is mapped to the top left corner (0,0) of the Device.

XGL_VDC_MAP_DEVICE
For this value, XGL ignores XGL_CTX_VDC_WINDOW and sets the value of XGL_CTX_DC_VIEWPORT to the Raster bounding box. Here XGL maps VDC directly onto DC. The *x*-axis points to the right, and one unit of distance in VDC is one unit in DC. The *y*- and *z*-axes may have opposite directions, as described in the next section, VDC Orientation.

The default value for XGL_CTX_VDC_MAP is XGL_VDC_MAP_DEVICE. If the VDC mapping method is XGL_VDC_MAP_DEVICE, the DC viewport is set to the device maximum coordinates when a Device is attached to a Context.

- XGL_CTX_VDC_WINDOW is the VDC rectangular region to be transformed into DC. The default value for XGL_CTX_VDC_WINDOW is [-1,1] × [-1,1] for 2D Contexts and [-1,1] × [-1,1] × [0,1] for 3D Contexts. This attribute is ignored if the VDC mapping method is XGL_VDC_MAP_DEVICE.
- XGL_CTX_DC_VIEWPORT is the DC rectangular region into which the VDC window maps. If the application attempts to set the value of XGL_CTX_DC_VIEWPORT so that the DC viewport extends partly or entirely outside the Device's boundaries, XGL will not accept this value but will set XGL_CTX_DC_VIEWPORT to the entire extent of the device. This is usually not the desired effect, so applications should detect this condition and take the following actions. If the DC viewport is partially outside the Device's

boundaries, the application should reduce the size of the DC viewport so it is entirely inside the Device, and reduce the VDC window by a proportional amount. If the DC viewport is entirely outside the Device's boundaries, the application can set XGL_CTX_RENDERING to FALSE so that the primitives will not draw into the Device.

The default value for XGL_CTX_DC_VIEWPORT is $[0,M_x] \times [0,M_y] \times [0,M_z]$ where (M_x, M_y, M_z) are the maximum dimensions of the device (see XGL_DEV_MAXIMUM_COORDINATES). However, the value of XGL_CTX_DC_VIEWPORT changes to match a Device when it is associated with the Context.

VDC Orientation

As discussed in the previous section, the orientation of VDC (specified by the attribute XGL_CTX_VDC_ORIENTATION) affects the mapping from VDC to DC. Since the VDC mapping (defined by XGL_CTX_VDC_MAP) only maps a rectangular region of VDC into a rectangular region in DC, the orientation of VDC is the same as DC. However, some applications may prefer to have a VDC space with the y-axis going from bottom to top.

On a workstation screen, DC space is right-handed: the x-axis is positive to the right, the y-axis positive top to bottom, and the z-axis positive away (into the screen). But the interpretation of *up, down, near* and *far* (specified by the orientation of the y- and z-axes) can be altered. This capability is provided by the attribute XGL_CTX_VDC_ORIENTATION. This attribute can take two values:

XGL_Y_DOWN_Z_AWAY

XGL_Y_UP_Z_TOWARD

These values are illustrated in Figure 11-3 on page 308. The default value is XGL_Y_DOWN_Z_AWAY.

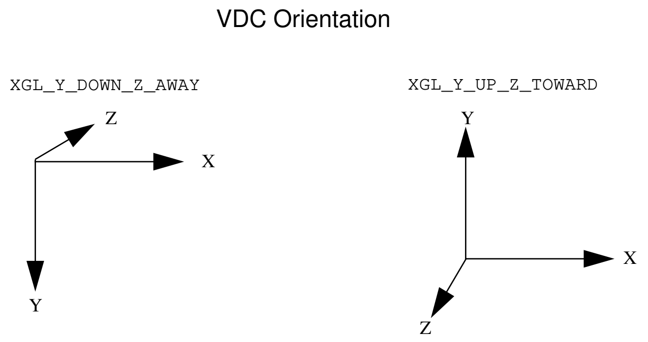

Figure 11-3 VDC Orientation

The Context attribute XGL_CTX_VDC_ORIENTATION controls the interpretation of the coordinates in the VDC system. Application programmers should keep the value of this attribute in mind when they decide how to compute the View Transform. Changing the value of XGL_CTX_VDC_ORIENTATION without changing XGL_CTX_VIEW_TRANS does not make sense because of resulting inconsistency.

MC-to-DC Transform

With the MC-to-DC Transform, an application can find the device coordinate values of a point in model space. The MC-to-DC Transform results from the concatenation of the Model Transform, the View Transform, the VDC Transform, and the Device's DC orientation.

The XGL_CTX_MC_TO_DC_TRANS attribute is read-only. The MC-to-DC Transform can only be changed by modifying the Local Model Transform, the Global Model Transform, the View Transform, or by setting one of the attributes that XGL uses to calculate the VDC-to-DC transformation: XGL_CTX_VDC_MAP, XGL_CTX_VDC_ORIENTATION, XGL_CTX_VDC_WINDOW, and/or XGL_CTX_DC_VIEWPORT. Changing XGL_CTX_MC_TO_DC_TRANS directly with Transform operators is not allowed. The default value of the MC-to-DC Transform is the Identity Transform.

Note – Evaluation of the MC-to-DC Transform is deferred until either XGL needs the Transform internally or the application explicitly gets it. Before attempting to use the MC-to-DC Transform, call `xgl_object_get()` to get `XGL_CTX_MC_TO_DC_TRANS` to ensure that XGL evaluates the MC-to-DC Transform if it requires updating.

Model Clipping in 3D

Model clipping in the 3D pipeline is provided to enable clipping of graphics primitives before viewing. Each model clipping plane is defined by a pair of values given in World Coordinates: a point in the plane and a unit vector normal to the plane. The direction of the vector points in the direction of a half-space whose boundary is the model clipping plane. All the geometry on the side of the normal vector is kept, and the rest is removed by clipping. A clipping volume is defined by intersecting several half-spaces defined by their planes.

When performing model clipping, XGL considers all the enabled model clipping planes in the order provided by the application. By default, XGL does not perform any model clipping.

Model clipping of some primitives, such as annotated primitives, is performed only on the reference point. If the reference point is clipped, the entire primitive is considered clipped.

Model clipping of primitives defined with a point type that lacks an edge flag field will generate a clipped primitive with all edges visible, including those on the model clipping plane boundaries.

The two attributes that define model clipping in the view pipeline are `XGL_3D_CTX_MODEL_CLIP_PLANES`, which specifies the list of model clip planes to use, and `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM`, which specifies the number of model clipping planes in use. When new model clipping planes are set, the application must first specify the number of clip planes with the `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM` attribute and then the list of model clipping planes with `XGL_3D_CTX_MODEL_CLIP_PLANES`. When the attribute `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM` is set to 0, model clipping is disabled.

View Clipping

Applications using a 2D transformation pipeline can perform view clipping in VDC against a rectangular boundary: $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. The default clip area is $[-1,1] \times [-1,1]$.

An application can disable view clipping against selected edges of the clip boundary.

In a 3D transformation pipeline, primitives can be clipped in homogeneous Virtual Device Coordinates against a view clip volume. The application specifies the volume as a rectangular parallelpiped in ordinary Virtual Device Coordinates: $[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$. Thus, view clipping is performed in homogeneous VDC against the view clip volume given by

$$\{ wx_{min} \le x \le wx_{max},\ wy_{min} \le y \le wy_{max},\ wz_{min} \le z \le wz_{max},\ w > 0 \}$$

where w is the homogeneous coordinate. The default clip volume in ordinary VDC is $[-1,1] \times [-1,1] \times [0, 1]$. An application can disable view clipping against selected planes of the clip volume.

XGL has three attributes for defining the view clipping (clipping of the image between WC Space and VDC Space):

`XGL_CTX_VIEW_CLIP_BOUNDS`
: This attribute defines the clipping *area* (for 2D Contexts) or *volume* (for 3D Contexts) in ordinary VDC. XGL converts the specified clipping region to homogeneous VDC, where clipping is performed. The values given by the application for `XGL_CTX_VIEW_CLIP_BOUNDS` need not be the same as those for the `XGL_CTX_VDC_WINDOW` attribute. The clipping planes defined by this attribute can be individually controlled by the attribute `XGL_CTX_CLIP_PLANES`. The default value is $[-1,1] \times [-1,1]$ for 2D, or $[-1,1] \times [-1,1] \times [0, 1]$ for 3D.

XGL_CTX_CLIP_PLANES
This attribute allows the application to specify individually which planes are used in view clipping within the current Context. The application can enable any combination of clipping planes by combining flags that represent each plane with bitwise logical-OR. The flags are:

XGL_CLIP_XMIN

XGL_CLIP_XMAX

XGL_CLIP_YMIN

XGL_CLIP_YMAX

XGL_CLIP_ZMIN

XGL_CLIP_ZMAX

The default value is NULL: no view clipping planes enabled.

XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY
An application can specify a 3D Transform in homogeneous space where the geometry is in front of and behind the viewer (the homogeneous coordinates can have both positive and negative w values). This attribute tells XGL how to handle the positive and negative w values. If the attribute is FALSE, the 3D clipping process checks for both positive and negative w values. Primitives are clipped in each region. For example, a line segment that has one endpoint in positive w and the other in negative w will be split into two line segments. If the attribute is TRUE (default value), clipping is restricted to positive w.

View Example Program

The following code fragment, view_perspect.c, is a section of an example program on perspective viewing. It contains the geometric data and the OpenWindows code for setting up a control panel to generate different views of a wireframe cube. To compile the complete program, type make view. The complete program includes view_perspect.c, view_main.c, and ex_utils.c. The latter two programs are listed in Appendix B.

Usage Note

In general, an application should not set the Context attributes XGL_CTX_LOCAL_MODEL_TRANS, XGL_CTX_GLOBAL_MODEL_TRANS, and XGL_CTX_VIEW_TRANS. When an application creates a Context, XGL creates these Transforms to implement the Context's transformation pipeline. An application can get the values of these attributes. Then both the application and the Context, which owns the Transform objects, have a handle to a Transform in the Context's pipeline. If this handle is the destination parameter of a Transform operator, the Context automatically knows about the change to one of its Transforms. For example, the function `view_calc` in the section of code in `view_perspect.c` gets a handle to the View Transform, which it stores in `view_trans`. It then copies the result of concatenations in `trans` to `view_trans`. Figure 11-4 shows the output of `view_perspect.c`.

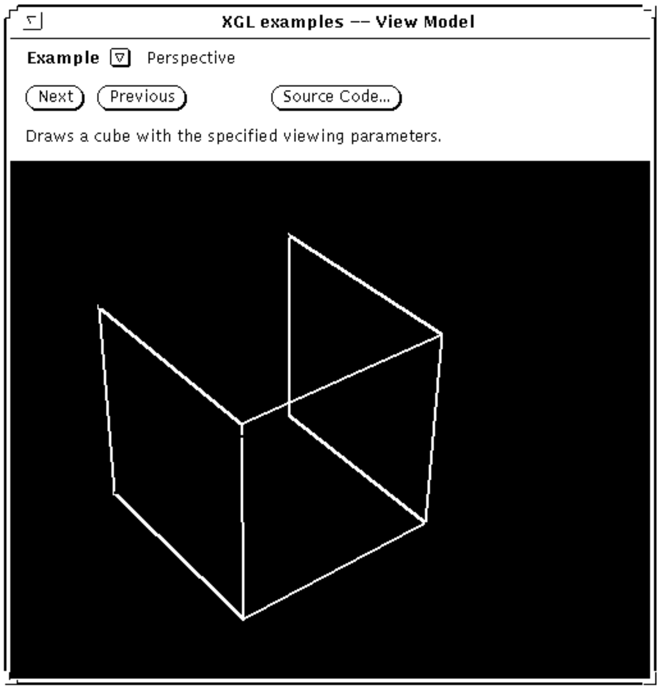

Figure 11-4 Output of `view_perspect.c`

Code Example 11-1 View Example

```
/*
 * view_perspect.c
 */
#include <stdio.h>
#include <math.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xgl/xgl.h>

#include "ex.h"

#define PI              3.141592654

extern Xgl_pt_list      pls[];
extern Panel_item       eye_x_item,
                        eye_y_item,
                        eye_z_item,
                        field_of_view_item;

static Xgl_object       ctx = NULL;
static Xgl_bounds_d3d   vdc_window = {-1.0,1.0,-1.0,1.0, -1.0,0.0};
static Xgl_vdc_map      vdc_map = XGL_VDC_MAP_ASPECT;

static void             draw_polylines ();
static void             view_calc (Xgl_pt_f3d*, float);


/****
 *
 * view_perspective
 *
 * This is the entry point from the controls on the example panel.  It
 * initializes some VDC parameters and calculates the next view.
 *
 ***/
void
view_perspective (Xgl_object  ctx3d)
{
    Xgl_pt_f3d             eye_pos;
    float                  fov;

    ctx = ctx3d;
    xgl_object_set (ctx,
```

```
                        XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                        XGL_CTX_VDC_MAP, vdc_map,
                        XGL_CTX_VDC_WINDOW, &vdc_window,
                        XGL_CTX_VIEW_CLIP_BOUNDS, &vdc_window,
                        XGL_CTX_CLIP_PLANES,
                        XGL_CLIP_XMIN | XGL_CLIP_XMAX | XGL_CLIP_YMIN |
                        XGL_CLIP_YMAX | XGL_CLIP_ZMIN | XGL_CLIP_ZMAX,
                        NULL);

    /* use wider lines to make them more visible */
    xgl_object_set (ctx,
                        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, 2.0,
                        NULL);

    /* get eye position and field of view from panel */
    sscanf((char *)panel_get_value (eye_x_item), "%f", &eye_pos.x);
    sscanf((char *)panel_get_value (eye_y_item), "%f", &eye_pos.y);
    sscanf((char *)panel_get_value (eye_z_item), "%f", &eye_pos.z);
    sscanf((char *)panel_get_value(field_of_view_item),"%f",&fov);

    /* calculate viewing and draw wireframe cube */
    view_calc (&eye_pos, fov);
}


/****
 *
 * view_set
 *
 * This is the entry point from the button "Set view" on the panel
 * for view model parameters. It reads the eye position and field
 * of view, then calculates the next view.
 *
 ***/
void
view_set (
    Panel_item          item,
    int                 value,
    Event              *event)
{
    Xgl_pt_f3d          eye_pos;
    float               fov;

    sscanf ((char *)panel_get_value (eye_x_item), "%f", &eye_pos.x);
    sscanf ((char *)panel_get_value (eye_y_item), "%f", &eye_pos.y);
    sscanf ((char *)panel_get_value (eye_z_item), "%f", &eye_pos.z);
```

```
    sscanf((char *)panel_get_value(field_of_view_item),"%f",&fov);

    view_calc (&eye_pos, fov);
}


/****
 *
 * view_calc
 *
 * Calculate the View Transform consisting of the
 * orientation given by the eye position and
 * the perspective given by the field of view.
 * Then redisplay the wireframe object.
 *
 ***/
static
void
view_calc (
    Xgl_pt_f3d          *eye,       /* Eye position in WC */
    float                fov)       /* Field of view (degrees) */
{
    Xgl_pt               pt;
    Xgl_pt_f3d           pt_f3d;
    Xgl_object           trans;
    Xgl_object           view_trans;
    Xgl_object           perspective_trans;
    Xgl_matrix_f3d       matrix;
    double               temp;
    float                distance;
    float                near;
    float                far;
    float                x_view_ratio;

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

    /* Set view orientation */

    /* Translate the origin of VDC to the eye position in WC */
    pt.pt_type = XGL_PT_F3D;
    pt.pt.f3d = &pt_f3d;
    pt_f3d.x = -eye->x;
    pt_f3d.y = -eye->y;
    pt_f3d.z = -eye->z;
    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 */
temp = atan (eye->x / eye->y);
if (eye->y > 0.0)
    temp += PI;
xgl_transform_rotate (trans, temp, XGL_AXIS_Y,
                      XGL_TRANS_POSTCONCAT);

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

/* Set view perspective and concatenate with view orientation */
distance=sqrt(eye->x *eye->x +eye->y *eye->y +eye->z *eye->z);
near = 2.0 - distance;
far = -2.0 - distance;
x_view_ratio = tan ((fabs (fov) / 2.0) * (PI / 180.0));
matrix[0][0] = 1.0 / x_view_ratio;
matrix[0][1] = matrix[0][2] = matrix[0][3] = 0.0;
matrix[1][1] = 1.0 / x_view_ratio;
matrix[1][0] = matrix[1][2] = matrix[1][3] = 0.0;
matrix[2][2] = 1.0 / (1.0 - near / far);
matrix[2][3] = -1.0;
matrix[2][0] = matrix[2][1] = 0.0;
matrix[3][2] = -near / (1.0 - near / far);
matrix[3][0] = matrix[3][1] = matrix[3][3] = 0.0;
perspective_trans = xgl_object_create (sys_st, XGL_TRANS, NULL,
                                       NULL);
xgl_transform_write (perspective_trans, matrix);
xgl_transform_multiply (trans, trans, perspective_trans);
xgl_object_destroy (perspective_trans);

/* Clear display before changing transformation pipeline */
xgl_context_new_frame (ctx);

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

/* Redisplay with new view transform */
```

```
    draw_polylines ();

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


/****
 *
 * vdc_map_set
 *
 * This is the notification procedure for the
 * VDC map choice stack.
 * Set the VDC map and redisplay the polylines.
 *
 ***/
void
vdc_map_set (
    Panel_item           item,
    int                  val,
    Event               *event)
{
    switch (val) {

        /* Map VDC window to entire window raster */
    case 0:
        xgl_context_new_frame (ctx);
        vdc_map = XGL_VDC_MAP_ALL;
        xgl_object_set (ctx, XGL_CTX_VDC_MAP, vdc_map, NULL);
        draw_polylines ();
        break;

       /* Map VDC window to window raster, but retain aspect ratio */
    case 1:
        xgl_context_new_frame (ctx);
        vdc_map = XGL_VDC_MAP_ASPECT;
        xgl_object_set (ctx, XGL_CTX_VDC_MAP, vdc_map, NULL);
        draw_polylines ();
        break;
    default:
        break;
    }
}
```

```
/****
 *
 * draw_polylines
 *
 * Displays the geometric data as polylines.
 *
 ***/
static
void
draw_polylines ()
{
    xgl_multipolyline (ctx, NULL, 4, pls);
}
```