# THE VIEW TRANSFORM

The transform that converts world-space positions to eye-space positions is the *view transform*. Once again, you express the view transform with a 4x4 matrix.

The typical view transform combines a translation that moves the eye position in world space to the origin of eye space and then rotates the eye appropriately. By doing this, the view transform defines the position and orientation of the viewpoint.

Figure 4-3 illustrates the view transform. The left side of the figure shows the robot from Figure 4-2 along with the eye, which is positioned at <0, 0, 5> in the world-space coordinate system. The right side shows them in eye space. Observe that eye space positions the origin at the eye. In this example, the view transform translates the robot in order to move it to the correct position in eye space. After the translation, the robot ends up at <0, 0, -5> in eye space, while the eye is at the origin. In this example, eye space and world space share the positive *y*-axis as their "up" direction and the translation is purely in the *z* direction. Otherwise, a rotation might be required as well as a translation.

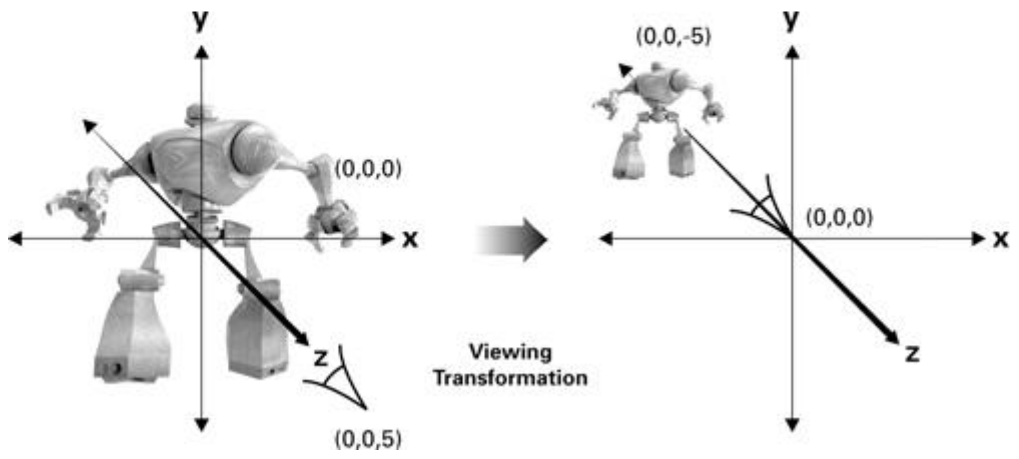

Figure 4-3 The Effect of the Viewing Transformation

**The Modelview Matrix**

Most lighting and other shading computations involve quantities such as positions and surface normals. In general, these computations tend to be more efficient when performed in either eye space or object space. World space is useful in your application for establishing the overall spatial relationships between objects in a scene, but it is not particularly efficient for lighting and other shading computations.

For this reason, we typically combine the two matrices that represent the modeling and view transforms into a single matrix known as the *modelview matrix*. You can combine the two matrices by simply multiplying the view matrix by the modeling matrix.

**Clip Space**

Once positions are in eye space, the next step is to determine what positions are actually viewable in the image you eventually intend to render. The coordinate system subsequent to eye space is known as *clip space,* and coordinates in this space are called*clip coordinates*.

The vertex position that a Cg vertex program outputs is in clip space. Every vertex program optionally outputs parameters such as texture coordinates and colors, but a vertex program *always* outputs a clip-space position. As you have seen in earlier examples, the **POSITION** semantic is used to indicate that a particular vertex program output is the clip-space position.

**The Projection Transform**

The transform that converts eye-space coordinates into clip-space coordinates is known as the *projection transform*.

The projection transform defines a *view frustum* that represents the region of eye space where objects are viewable. Only polygons, lines, and points that are within the view frustum are potentially viewable when rasterized into an image. OpenGL and Direct3D have slightly different rules for clip space. In OpenGL, everything that is viewable must be within an axis-aligned cube such that the *x*, *y*, and *z* components of its clip-space position are less than or equal to its corresponding *w* component. This implies that $-w \leq x \leq w$, $-w \leq y \leq w$, and $-w \leq z \leq w$. Direct3D has the same clipping requirement for *x* and *y*, but the zrequirement is $0 \leq z \leq w$. These clipping rules assume that the clip-space position is in homogeneous form, because they rely on *w*.

The projection transform provides the mapping to this clip-space axis-aligned cube containing the viewable region of clip space from the viewable region of eye space—otherwise known as the view frustum. You can express this mapping as a 4x4 matrix.

## The Projection Matrix

The 4x4 matrix that corresponds to the projection transform is known as the *projection matrix*.

Figure 4-4 illustrates how the projection matrix transforms the robot in eye space from Figure 4-3 into clip space. The entire robot fits into clip space, so the resulting image should picture the robot without any portion of the robot being clipped.

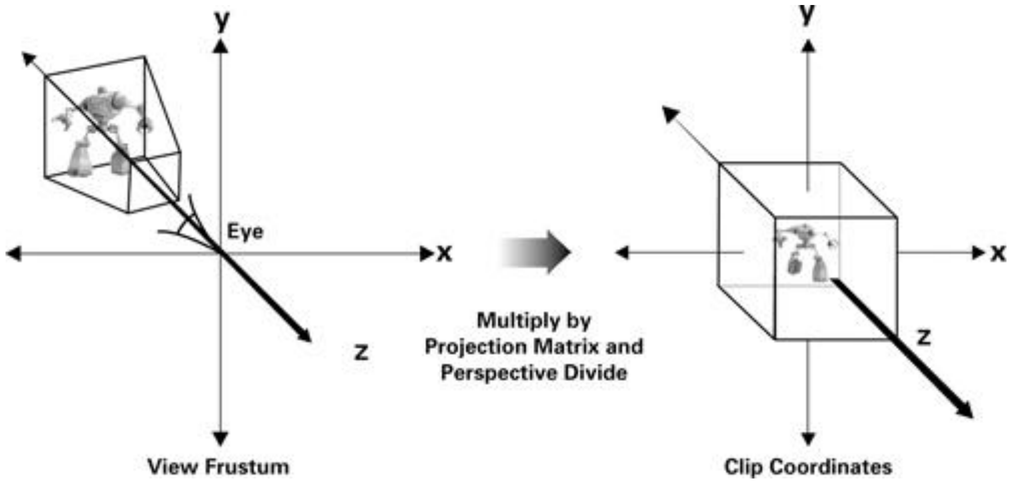

Figure 4-4 The Effect of the Projection Matrix

The clip-space rules are different for OpenGL and Direct3D and are built into the projection matrix for each respective API. As a result, if Cg programmers rely on the appropriate projection matrix for their choice of 3D programming interface, the distinction between the two clip-space definitions is not apparent. Typically, the application is responsible for providing the appropriate projection matrix to Cg programs.

Source : http://elearningatria.files.wordpress.com/2013/10/cse-vi-computer-graphics-and-visualization-10cs65-notes.pdf