SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

Copyright © 2018 Sung-eui Yoon, KAIST

FREELY AVAILABLE ON THE INTERNET

http://sglab.kaist.ac.kr/~sungeui/render

First printing, July 2018

3
Transformation

Many components of rasterization techniques rely upon different types of transformation. In this chapter, we discuss those transformation techniques.

3.1 Viewport Transformation

In this section, we explain the viewport transformation based on an example. Fig. 3.1 show different spaces that we are going to explain.

Suppose that you have an arbitrary function, $f(x, y)$, as a function of 2 D point (x, y); e.g., $f((x, y)) = x^2 + y^2$. Now suppose that you want to visualize the function in your computer screen with a particular color encoding method, e.g., heat map that assigns hot and cold colors depending on values of $f(x, y)$.

This function is defined in a continuous space, say x and y can be any real values. In computer graphics, we use a term of world to denote a model or scene that we would like to visualize or render. In this case, the function $f(x, y)$ is our world. Our goal is to visualize this function so that we can understand this function better. In many cases, the world is too large and thus we cannot visualize the whole world in a single image. As a result, we commonly introduce a camera to see a particular region of the world.

Unfortunately, our screen is not in the continuous space and has only a limited number of pixels, which is represented by a screen resolution. Our graphics application can use the whole screen space or some part of it. Let us call that area as a screen space. Fig. 3.2 show common conventions of the screen space. Finally, we visualize a part of the world seen through the camera into a part of our screen space, which is commonly known as a viewport; note that we can have multiple viewports in our screen.

Suppose a position, x_w, in the world that we are now seeing in the camera. In the end, we need to compute its corresponding position, x_s, in our screen space of the viewport. If we know x_s, we can draw

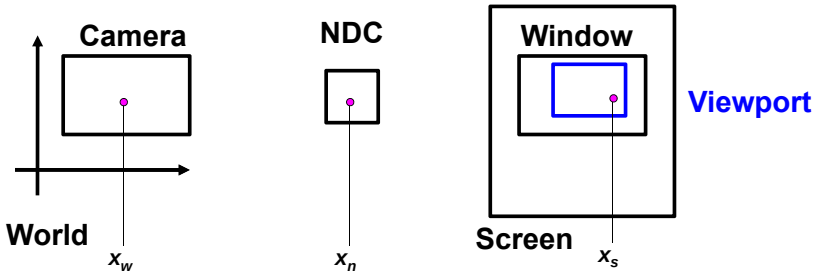

Figure 3.1: This shows a mapping from a viewable region in the world through a camera to the viewport in our screen space pass through the intermediate space, normalized device coordinate (NDC).

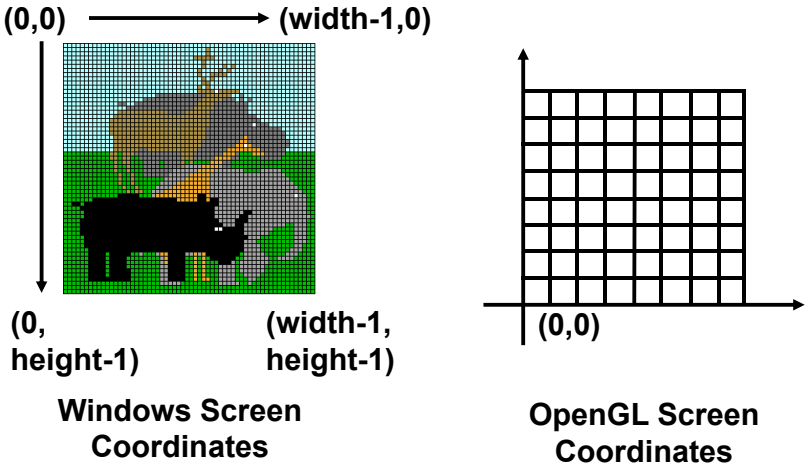

Figure 3.2: This shows two different conventions of screen coordinate spaces.

the color of the world position x_w at x_s. The question is how to compute x_s from x_w, i.e., the mapping from the world space to the viewport or screen space.

Normalized device coordinate (NDC). While world and screen spaces are two fundamental spaces, we also utilize NDC. NDC is a canonical space, whose both X and Y values are in a range of $[-1, 1]$. NDC serves as an intermediate space that is transformed to the screen space, which is hardware-dependent space. As a result, given the world potion x_w, we first need to compute a position in the NDC space, x_n, followed by mapping to x_s. We will also see various benefits of using NDC later, which include simplicity and thus efficiency of various rasterization operations.

Mapping from the world space to NDC. Suppose that the part of the world that we can see through a camera is represented by $[w.l, w.r] \times [w.b, w.t]$, where $w.l$ and $w.r$ are the visible range along X-axis and $w.b$ and $w.t$ define the visible range in Y-axis, while the

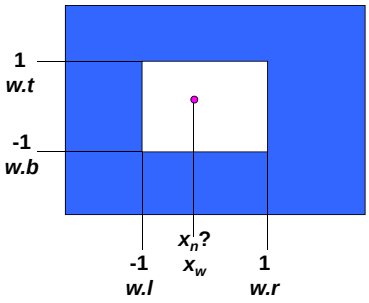

Figure 3.3: Mapping between the world space and NDC.

NDC space is represented by $[-1, 1] \times [1, 1]$.

Since the relative ratio of x_w and x_n is same in each space, we have the following relationship:

$$\frac{x_n - (-1)}{1 - (-1)} = \frac{x_w - (w.l)}{w.r - w.l}.$$

$$x_n = 2\frac{x_w - w.l}{w.r - w.l} - 1.$$

$$x_n = Ax_w + B,$$

where $A = \frac{2}{w.r - w.l}$, $B = -\frac{w.r + w.l}{w.r - w.l}$. This equation indicates that given the information, we can compute the NDC coordinate with one multiplication and one summation. Similarly, we can derive the mapping equation from x_n to x_s.

An issue of this approach is that there are too many pixels and thus evaluating such simple equations requires computational time. Since most graphics applications require interactive or real-time performance, we need to think about efficient way of handling these operations early in the history of computer graphics. Furthermore, it turns out that such mapping and similar transformations are very common operations in many graphics applications. The most common way of handling them in an efficient and elegant way is to adopt linear algebra and use matrix operations.

3.1.1 Common Questions

Can glBegin () with GL_POLYGON support concave polygons? According to its API description, GL_POLYGON works only with convex polygons. But, what may happen with concave polygons? Since it is not part of the specification of OpenGL, each vendor can have their own handling method for that kind of unspecified cases. If you are interested, you can try it out and let us know.

In the case of rendering circles, shown as an example in the lecture note, we render them by using lines. Is there a direct primitive that supports the circle? OpenGL has a limited functionality that supports continuous mathematical representations including circles, since a few model representations (e.g., triangles) have been widely used and it is hard to support all the possible representations. However, OpenGL keeps changing and it may support many continuous functions in a near future. At this point of time, we need to discretize continuous functions with triangles or other simple primitives and render them.

We use the NDC between the world space and the screen space. Isn't it inefficient? Also, don't we lose some precision during this

process? There is certainly some overhead by introducing the NDC. However, it is very minor compared to its benefits in terms of simplifying various algorithms employed throughout the rendering process. Yes. We can lose more precision during the conversion process due to float operations. However, it may be very small and may not cause significant problems for rendering purposes. Nonetheless, the transformation is based on analytic equations, not pixels, and thus can be easily recovered to the original information.

OpenGL is designed for cross-platform. But, I think that it means that we cannot use assembly programming for higher optimizations. Yes. You're right. We cannot use assembly languages for such optimizations. However, programmers for graphics drivers for each graphics vendor definitely use an assembly language and attempt to achieve the best performance. High-level programmers like us rely on such drivers and optimize programs with OpenGL API available to us.

Multi-threading with OpenGL: Since OpenGL has been designed very long time ago and has many different threads, it requires some cares to use multiple threads for OpenGL. There are many articles in internet about how to use multiple threads with OpenGL. I recommend you to go over them, if you are interested in this topic.

Why do we use a viewport? The viewport space doesn't need to be the whole window space. Given a window space, we can decompose it into multiple sub-spaces and use sub-spaces for different purposes. An example of using multiple viewports is shown in Fig. 3.4.

3.2 *2D Transformation*

In this section, we discuss how to represent various two dimensional transformation in the matrix form. We first discuss translation and rotation.

2D translation has the following forms:

$$x' = x + t_x, \tag{3.1}$$
$$y' = y + t_y, \tag{3.2}$$

where (x, y) is translated as an amount of (t_x, t_y) into (x', y'). They are also represented by a matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}. \tag{3.3}$$

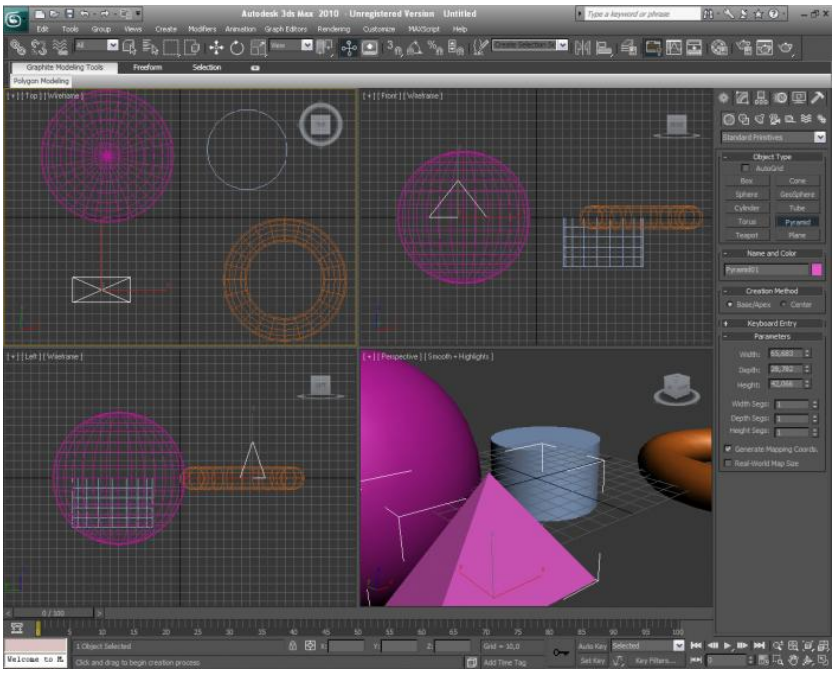

Figure 3.4: This figure shows multiple viewports, each of which shows an arbitrary 3D view in addition to top, front, and side views. The image is excerpt from screenshots.en.sftcdn.ne.

Given the 2D translation, its inverse function that undoes the translation is:

$$x = x' - t_x, \tag{3.4}$$
$$y = y' - t_y. \tag{3.5}$$

Also, its identity that does not change anything is:

$$x' = x + 0, \tag{3.6}$$
$$y' = y + 0. \tag{3.7}$$

Let us now consider 2D rotations. Rotating a point (x, y) as an amount of θ in the counter-clock wise is:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = R_\theta \begin{bmatrix} x \\ y \end{bmatrix}, \tag{3.8}$$

where R_θ is the rotation matrix. Its inverse and identity are defined as the following:

$$R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}, \tag{3.9}$$
$$R_{\theta=0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{3.10}$$

Suppose that you want to rotate an object by 30 degrees, followed by rotating it again with 60 degrees. We intuitively know that rotating 90 degrees in a single time gives the same effect of rotating 30 degrees and 60 degrees again.V Formally, one can prove the following equation:

$$R_{\theta_2} R_{\theta_1} = R_{\theta_1 + \theta_2}. \tag{3.11}$$

3.2.1 *Euclidean Transformation*

In this subsection, we would like to discuss a particular class of transformation, Euclidean transformation. The Euclidean transformation preserves all the distances between any pairs of points. Its example includes translation, rotation, and reflection. Since the shape of objects under this transformation is preserved, the Euclidean transformation is also known as rigid transformation.

This rigid transformation is one of most common transformation that we use for various game and movie applications. For example, camera rotation and panning are implemented by the rigid transformation.

Mathematically, the Euclidean transformation is represented by:

$$T(x) = Rx + t, \tag{3.12}$$

where R and t are rotation matrix and 2D translation vector.

While this is a commonly used mathematical representation, this representation has a few drawback for graphics applications. Typically, we have to perform a series of rotation and translation transformation for performing the viewport transformation, camera operations, and other transformation applied to objects. As a result, it can take high memory and time overheads to apply them at runtime. Furthermore, there is cases that we need to compute a invert operation from a coordinate from the screen space to the corresponding one in the world space. Given the series of rotation and translation operations, the inverting operation can require multiple steps.

As an elegant and efficient approach to these issues, the homogeneous coordinate has been introduced and explained in the next section.

3.2.2 *Homogeneous Coordinate*

Homogeneous coordinates are originally introduced for projective geometry, but are widely adopted for computer graphics, to represent the Euclidean transformation in a single matrix.

Homogeneous coordinates provides various benefits for transformation and are thus commonly used in graphics.

Suppose a 2D point, (x, y) in the 2D Euclidean space. For the homogeneous coordinates, we introduce an additional coordinate,

and (x, y) in the 2D Euclidean space corresponds to $(x, y, 1)$ in the 3D homogeneous coordinates. In fact, $(zx, zy, z), z \neq 0$ also corresponds to (x, y) by dividing the third coordinate z to the first and second coordinates, to compute the corresponding 2D Euclidean coordinate.

Intuitively speaking, (zx, zy, z) represents a line in the 3D homogeneous coordinate space. Nonetheless, any points in the line maps to a single point (x, y) in the 2D Euclidean space. As a result, it can describe a projection of a ray passing through a pin hole to a point.

Let us now describe its practical benefits for our problem. Before we describe the Euclidean transformation (Eq. 3.12) and its problems. By using the 3D homogeneous coordinate, the Euclidean transformation is represented by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.13}$$

Note that the translation amount t_x and t_y are multiplied with the homogeneous coordinate, which is one. As a result, the translation is incorporated within the transformation matrix that also encodes the rotation part simultaneously.

One of benefits of using the homogeneous coordinates is to support the translation and rotation in a single matrix. This property addresses problems of the Euclidean transformation (Sec. 3.2.1). Specifically, even though there are many transformations, we can represent each transformation in a single matrix and thus their multiplication is also represented in a single matrix. Furthermore, its inversion can be efficiently performed. Thanks to these properties resulting in a higher performance, the homogeneous coordinates have been widely adopted.

Revisit to mapping from the world space to NDC. We discussed viewport mapping, one of which operation transforms world space coordinates to those in NDC space (Sec. 3.1). Since this transformation uses multiplication, followed by the additions, it can be represented by homogeneous coordinates and thus in a single matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{w.r-w.l} & 0 & -\frac{w.r+w.l}{w.r-w.l} \\ 0 & \frac{2}{w.t-w.b} & -\frac{w.t+w.b}{w.t-w.b} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.14}$$

Nonetheless, the matrix is not exactly in the Euclidean transformation, since it involves scaling. This is covered in the affine transformation in the next section.

3.2.3 *Affine Transformation*

We discussed the Euclidean transformation that is a combination of rotation and translation in Sec. 3.2.1. We now study on an affine transformation, which covers wider transformation than the Euclidean transformation.

In the 2D case, the affine transformation has the following matrix representation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.15}$$

The affine transformation preserves parallel lines under the transformation, but does not necessarily preserve angles of lines. The affine transformation covers a combination of rotation, translation, shearing, reflection, scaling, etc. The transformation is also called projective transformation, since it also supports projection, which is discussed in Sec. 4.2.

OpenGL functions. Various transformation functions (e.g., $glTranslate(\cdot)$) available at early versions of OpenGL (e.g., version 2) are deprecated in recent versions. Nonetheless, it is informative to see its usage with corresponding matrix transformations, which are adopted in the recent OpenGL.

The following code snippet shows a display function of rendering a rectangle with a rotation matrix.

```
void display(void)
{
  // we assume the current transformation matrix to be the identify matrix.
   glClear(GL_COLOR_BUFFER_BIT); // initialize the color buffer.

   glPushMatrix(); // store the current matrix, the identify matrix, in the matrix stack
   glRotatef(spin, 0.0, 0.0, 1.0); // create a rotation matrix, M_r.
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0); // create geometry, say, v.
   glPopMatrix(); // go back to the initial identify matrix.

   glFinish (); // send all OpenGL commands to GPU and finish once they are done.

   glutSwapBuffers();
}
```

The actual rasterization done in GPU occurs once $glFinish()$ is called. Before rasterizing the rectangle, we perform the specified transformation, which is to compute v', where $v' = M_r v$. We then rasterize the rectangles with transformed geometry, v'.

3.2.4 *Common Questions*

Is there any benefit of using column-major ordering for the matrix over row-major ordering? Not much. Some people prefers to use column-major, while others like to use row-major. Somehow, people who designed OpenGL may prefer column-major ordering.

3.3 *Affine Frame*

In this chapter, we started with viewport transformation, followed by 2D transformation. Overall, an underlying question along these discussions is this: suppose that we have two different frames and we know coordinates of a point in a frame. What is the coordinates of the point in the different frame? For example, the viewport transformation is an answer to this question with the world and viewport frames.

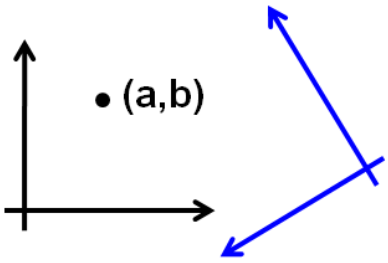

Figure 3.5: What is the coordinate of the point against the blue frame?

We use a set of linearly independent basis vectors to uniquely define a vector. Suppose that $\vec{V}_1, \vec{V}_2, \vec{V}_3$ are to be three such basis vectors represented in a column-wise vector. We can then define a vector, $\vec{X}$, with three different coordinates, c_1, c_2, c_3, as the following:

$$\vec{X} = \sum_{i=1}^{3} c_i \vec{V}_i = \begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \mathbf{V}c, \tag{3.16}$$

where $\mathbf{V}$ is a 3 by 3 matrix, whose columns corresponds to the basis vectors.

Now let's consider how we can represent a point, $\dot{p}$, in the 3D space. Unfortunately, the point cannot be represented in the same manner as we used for defining a vector in above. To define a point in the space, we need an anchor, i.e., origin, of the coordinate system. This makes the main difference between points and vectors. Specifically, points are absolute locations, while vectors are relative quantity.

A point, $\dot{p}$, is defined with respect to the absolute origin, $\dot{o}$, as the following:

$$\dot{p} = \dot{o} + \sum_{i=1}^{3} c_i \vec{V}_i = \begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 & \dot{o} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}. \tag{3.17}$$

Simply speaking, we can define a point by using 4 by 4 matrix $\begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 & \dot{o} \end{bmatrix}$, whose each column includes three basis vectors and the origin. As a result, this matrix is also called a affine frame; in this chapter, we will just use a frame for simplicity.

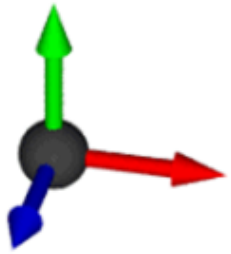

Figure 3.6: The affine frame consisting of three basis vectors and the origin.

We can also define a vector with the frame as the following:

$$\vec{x} = \sum_{i=1}^{3} c_i \vec{V}_i = \begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 & o \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 0 \end{bmatrix}. \tag{3.18}$$

Interestingly, the fourth coordinate for any vector with the frame has 0, since the vector is not based on the origin.

Defining points and vectors with the frame has various benefits. Here are some of them:

1. **Consistent model.** Various operations between points and vectors reflects our intuition. For example, subtracting two points yields a vector and adding a vector to a point produces a point. These operations are consistent with respect to our representations with the frame:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 0 \end{bmatrix}. \tag{3.19}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}. \tag{3.20}$$

2. **Homogeneous coordinate.** We introduced the homogeneous coordinate to represent the rotation and translation in a single matrix (Sec. 3.2.2). Such homogeneous coordinates are actually defined in the affine frame, and the fourth coordinate indicates whether it represents points or vectors depending on its values.

3. **Affine combinations.** Adding one point to another point does not make sense. Nonetheless, there is a special case that makes sense. Suppose that we add two points with weights of α_1 and α_2, where the sum of those weights to be one, i.e., $\alpha_1 + \alpha_2$. We then have the following equation:

$$\alpha_1 \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}. \tag{3.21}$$

Intuitively speaking, this affine combination results in a linear interpolation between those two points. This idea can be also extended to any number of points. One example with three points includes the barycentric coordinate (Sec. 10.2).

3.4 *Local and Global Frames*

We would like to conclude this section by discussing local and global frames, followed by revisiting the viewport transformation in these frames.

Suppose that you have a point, $\dot{p}$, defined in the affine frame, $\mathbf{W}$, with a coordinate of c; i.e., $\dot{p} = \mathbf{W}c$. We now want to translate the point with $\mathbf{T}$ and then rotate it with $\mathbf{R}$. The transformed point, $\dot{p}'$, is defined as the following and can be interpreted in two different directions:

$$\begin{aligned}\dot{p}' =& \mathbf{WRT}c \\ =& \mathbf{W}(RTc) = \mathbf{W}c' \ // \text{ use the global frame} \quad (3.22)\\ =& (\mathbf{WRC})c = \mathbf{W}'c \ // \text{ use a local frame.} \quad (3.23)\end{aligned}$$

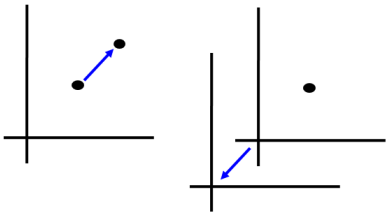

Figure 3.7: Global (left) and local (right) frames of the same transformation.

The second equation is derived by changing the coordinate given the global frame $\mathbf{W}$. The third equation is derived by modifying the frame itself into a new local frame, say $\mathbf{W}'$, while maintaining the coordinate. These two different interpretation can be useful for understanding different transformations.

Let us remind you that we started with this chapter by discussing the viewport transformation. Let's apply local and global frames to the viewport transformation. During the viewport transformation, the point does not move. Instead, we want to compute a coordinate in the viewport space, $\mathbf{V}$, from that in the world space, $\mathbf{W}$. In other words, we can represent them as the following:

$$\dot{p} = \mathbf{W}c = \mathbf{V}c', \quad (3.24)$$

where the relationship between the world and viewport spaces is represented by $\mathbf{V} = \mathbf{WS}$.

In this case, the coordinate c' in the viewport space is computed as the following:

$$\dot{p} = \mathbf{W}c = \mathbf{VS}^{-1}c = \mathbf{V}(\mathbf{S}^{-1}c) = \mathbf{V}c'. \quad (3.25)$$

This approach, considering coordinates with different frames, can be very useful for considering complex transformation. We will use this approach for explaining 3D rotation transformations in the next section.

3.5 *3D Modeling Transformation*

To create a scene consisting of multiple objects, i.e., models, we need to place those models in a particular place in the world. This

operation is modeling transformation that commonly consists of translation and rotation.

3D translation is extended straightforwardly from the 2D translation:

$$c' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_y \\ 0 & 0 & 0 & 1 \end{bmatrix} c. \tag{3.26}$$

The rotation in the 3D space along the canonical axis is easily extended from the 2D case. For example, the rotation along the X axis is computed as the following:

$$\mathbf{R}_{\grave{\mathbf{X}}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.27}$$

The 3D rotation against an arbitrary vector requires additional treatments. Nonetheless, the affine frame we studied in Sec. 3.2.3 simplifies this process and we thus discuss this approach here in this section.

Suppose that we would like to rotate a vector, $\vec{x}$, given a rotation axis vector, $\vec{a}$. When the rotation axis aligns with one of canonical X, Y, or Z axis, we can easily extend the 2D rotation matrix to 3D rotation matrix. Unfortunately, the rotation axis may not be aligned with those canonical axes, complicating the derivation of the rotation matrix. We now approach this problem in the perspective of the affine frame. The vector $\vec{x}$ can be considered to be defined in the frame of three basis vectors consisting of $\vec{a}$, the red one, and two other orthogonal vectors, the black and green vectors in the figure.

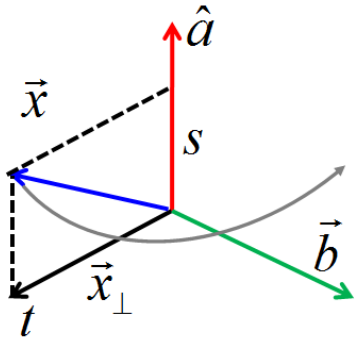

Figure 3.8: Geomtry for 3D rotation.

Let's first compute the black vector $\vec{x}_\perp$, which is orthogonal to $\vec{a}$, and the plane spanned by these two vectors contains the rotation vector $\vec{x}$. We can decompose two coordinates, s and t, of $\vec{x}$ in the plane defined by $\vec{a}$ and $\vec{x}_\perp$, respectively. To compute such coordinates, we can apply the dot product. s and t, and $\vec{x}_\perp$ are then computed by canceling the coordinate of $\vec{a}$, as follows:

$$\begin{aligned} s &= \vec{x} \cdot \vec{a}, \\ \vec{x}_\perp &= \vec{x} - s\vec{a}, \\ t &= \vec{x} \cdot \vec{x}_\perp. \end{aligned}$$

The green vector $\vec{b}$ that is orthogonal to both $\vec{a}$ and $\vec{x}_\perp$ is computed by the cross product between $\vec{a}$ and $\vec{x}_\perp$; i.e., $\vec{b} = \vec{a} \times \vec{x}_\perp$.

So far, we have computed three basis vectors of a local affine frame that can define the vector $\vec{x}$. Specifically, the vector $\vec{x}$ is defined as the

following:

$$\begin{bmatrix} \vec{a} & \vec{x}_\perp & \vec{b} & \dot{o} \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \\ 0 \end{bmatrix}, \tag{3.28}$$

where $\dot{o}$ is a virtual origin of our local affine frame. The rotation in the amount of θ along the rotation axis $\vec{a}$ is transformed to the rotation along the X axis in the local affine frame. As a result, coordinates of the rotated vector are computed as the following:

$$\begin{bmatrix} \vec{a} & \vec{x}_\perp & \vec{b} & \dot{o} \end{bmatrix} \mathbf{R}_{\grave{\mathbf{X}}} \begin{bmatrix} s \\ t \\ 0 \\ 0 \end{bmatrix}. \tag{3.29}$$

Quaternion is an popular alternative for the 3D rotation, and many tutorials are available for the topic.