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

4 Camera Setting

In this chapter, we discuss two important aspects of a camera setting: 1) how to setup camera parameters, and 2) how to project objects into a 2D viewing space.

For the simplicity, we discuss these issues with a pinhole camera, one of simple camera setting. Modern cameras employ many different types of lenses and thus are much more complex than the pinhole camera. We also discuss how to extend such realistic cameras in other chapters .

4.1 Viewing Transformation

To see a particular portion of the world scene, it is natural to specify the camera. The camera is specified with its origin, and X, Y, and Z axis in the world space (Fig. 4.1), which define the affine frame of the camera space. The viewable image is then mapped to the X-Y space in the camera space. As a result, the goal of the viewing transformation is to convert the coordinates defined in the world space into those in the camera or viewing space.

Unfortunately, defining those parameters, e.g., X-axis of the camera, in the world space is neither an intuitive nor easy task. Instead, we would like to design an intuitive and easy way of defining those parameters. Following quantities are commonly adopted ones for defining the viewing space:

1. **Eye point, e.** This is simply the position of the camera.

2. **Look-at point, p.** We typically have a specific target that we want to look at. As a result, requiring such a look-at point is not a big burden to users.

3. **Up-vector, $\vec{u}_a$.** While we have the look-at point, the orientation of the camera is not specified. For example, we can look at the target point, while we maintain our head upward or downward.

Figure 4.1: To generate an image, we specify a camera in the world space, which consists of the origin and X, Y, and Z axis of the camera.

As a result, we require to specify an up-vector, $\vec{u}_a$, that define the orientation of the camera.

While we prepared an intuitive way of defining the camera, we still need to define the affine frame of the viewing space. The next goal is to define the affine frame from these parameters, as the following:

1. **Look-at vector, $\vec{l}$.** The Z-direction of the camera can be computed by computing the look-at vector, $\vec{l}$, which is computed by $p - e$ with a proper normalization, $\hat{l} = \frac{l}{|l|}$. Note that we use the hat notation, $\hat{}$, to denote a normalized vector, whose magnitude is one.

2. **Right vector, $\vec{r}$.** The X-axis of the camera is computed by the cross product between the look-at vector $\hat{l}$ and the given up-vector $vecu_a$:

$$\vec{r} = \vec{l} \times \vec{u}_a,$$
$$\hat{r} = \frac{\vec{r}}{|r|}. \tag{4.1}$$

r

3. **Adjusted up-vector, $\hat{u}$.**

   The given up-vector may not be perpendicular to the look-at and right vectors. As a result, we recompute a new up-vector, $\hat{u}$, that is perpendicular to both of them: $\hat{u} = \hat{r} \times \hat{l}$. Since it is difficult and cumbersome for users to specify the initial up-vector in this way, we adjust the up-vector in this way. Usually, this process is performed within a graphics library such as OpenGL.

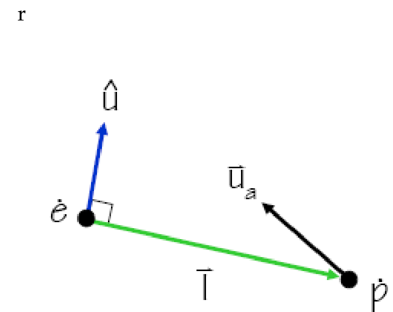

Figure 4.2: Adjusting the initial up-vector.

Let's consider how to transform coordinates in the world space to the viewing space defined in the camera space. This problem is exactly same one that we discussed for local and global frames of Sec. 3.4. As a result, we apply the concept of changing frames to this problem.

Suppose that the coordinate in the world space is c. What we want is to translate the camera origin such that the camera origin becomes the origin in the viewing space. This is represented by $\mathbf{T_{-e}}$. We then rotate the coordinate with a rotation matrix, $\mathbf{R_v}$, into the camera space. As a result, we have the following equation:

$$\mathbf{W}c = \mathbf{E}\mathbf{R_v}\mathbf{T_{-e}}c, \tag{4.2}$$

where $\mathbf{W}$ and $\mathbf{E}$ are frames of the world space and camera space. Therefore, the viewing matrix is defined as $\mathbf{R_v}\mathbf{T_{-e}}$ that convert the world space coordinate c into one in the camera space.

For the world space, we use canonical basis vectors and thus $\mathbf{W} = \mathbf{I}$. Also, the viewing space $\mathbf{E}$ is represented by the three basis vectors. As a result, we have the following relationship:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix} \mathbf{R_v} \tag{4.3}$$

$$\begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix}^{-1} = \mathbf{R_v} \tag{4.4}$$

The matrix of $\begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix} = M$ is an orthonormal matrix, whose columns are orthogonal to each other and unit normal vectors. In this case, $M^T M = I$ is satisfied and thus M^{-1} can be easily computed by M^T. As a result, the rotation matrix $\mathbf{R_v}$ is computed as the following:

$$\mathbf{R_v} = \begin{bmatrix} \hat{r}^T \\ \hat{u}^T \\ -\hat{l}^T \end{bmatrix} \tag{4.5}$$

Given the rotation matrix and translation matrix, the viewing matrix $\mathbf{V}$ is computed as the following:

$$\mathbf{V} = \mathbf{R_v}\mathbf{T_{-e}} = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & 0 \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\ -\hat{l}_x & -\hat{l}_y & -\hat{l}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.6}$$

Connections to OpenGL. In an old version of OpenGL, the viewing transformation is setup by calling ”gluLookAt (·)". This function simply constructs the viewing matrix (Eq. 4.6) and composes it with the current matrix that OpenGL maintains. In a recent OpenGL version, e.g., 3.0, gluLookAt is no longer available, and one needs to maintain their own viewing transformation in a vertex shader . Fortunately, there are many available codes to implement equivalent functions in recent versions of OpenGL.

4.2 Projection

Figure 4.3: Orthographic projection.

Projection occurs right after viewing transformation. Projection maps 3D points defined in the camera or eye space into 2D points in the image space. There are two common projection methods: orthographic and perspective projection.

The orthographic projection simply flattens 3D objects into the 2D image space. It preserves parallel lines before and after the projection. It is used for top and side views in various modeling tools (e.g., 3ds Max). It can, however, appear unnatural due to the lack of perspective foreshortening.

In a simplest form, the orthographic projection is defined as the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \quad (4.7)$$

As an additional details to the viewing and projection transformation, we also define a view volume for each camera. Fig. 4.4 shows an example of the view volume for the orthographic projection with related parameters defining the view volume. After the orthographic projection, we map those 3D coordinates into ones in the NDC space (Sec. 3.1).

In this context, the orthographic projection mapping to the NDC space is computed as the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{-(r+l)}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{-(t+b)}{t-b} \\ 0 & 0 & \frac{2}{f-n} & \frac{-(f+n)}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \quad (4.8)$$

where r, l, t, b, f, n indicates right, left, top, bottom, far, and near, respectively. As a sanity check, when we have a coordinate of $(l, 0, 0, 1)$, it should give us -1 after the orthographic projection. This is verified as the following:

$$x'(l) = \frac{2l}{r-l} - \frac{r+l}{r-l} = -\frac{r-l}{r-l} = -1. \quad (4.9)$$

Note that we do not cancel the Z-coordinate even after the orthographic projection. We actually use the Z-coordinate for an important rendering task, visibility check using the depth buffer (Ch. 7.4).

4.2.1 Perspective Projection

Perspective projection is very common in modern computer animation. It, however, takes a long history to be fully understood and

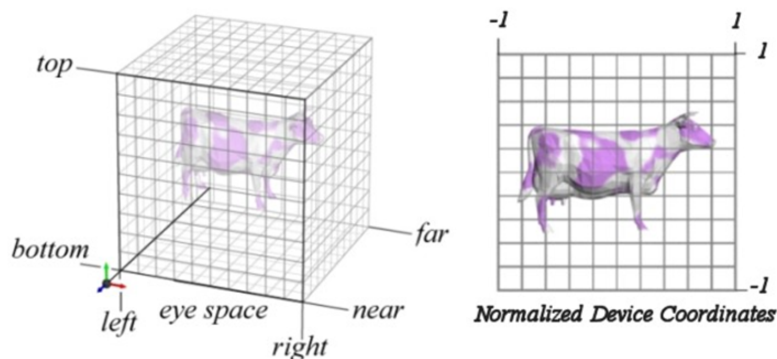

Figure 4.4: The left figure shows a view volume for the orthographic projection. After the orthographic projection, we map 3D coordinates into ones in the NDC space.

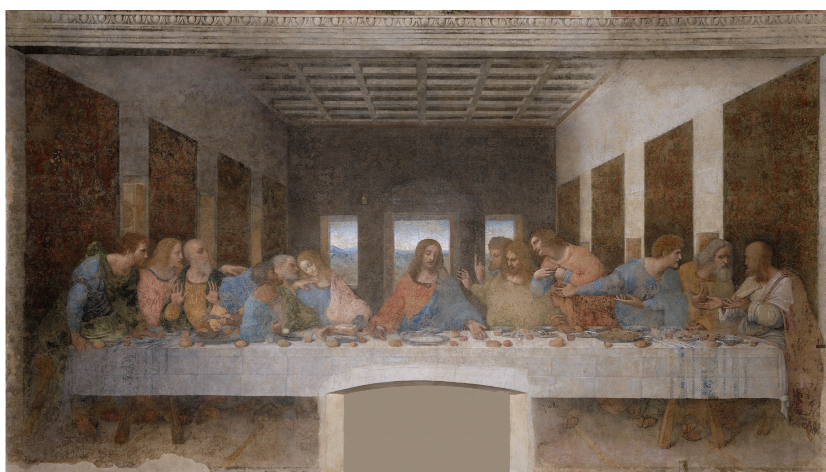

Figure 4.5: This shows the last supper drawn by Leonardo da Vince. This painting shows that objects are drawn under the perspective projection. Furthermore, the vanishing point is located at the Jesus to emphasize the theme of the paining. In other words, perspective projection may be intentionally used for artistic expression.

used in arts. Fig. 4.5 shows an early example of a painting adopting the perspective projection and its intentional use for artistic expression.

A key characteristic of the perspective projection is foreshortening of far-away objects compared to close objects. Another characteristic of perspective projection is that parallel lines in perspective projection always intersect at a point, i.e., vanishing point.

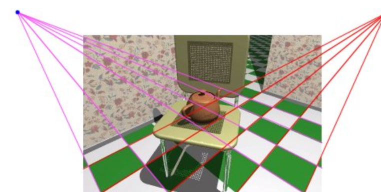

Figure 4.6: Vanishing points.

In this section, we discuss such a perspective projection under a simplistic camera model, pinhole camera. Fig. 4.7 shows a 2D schematic illustration of a point into a view plane under a pinhole camera. The point, p, has (y, z) coordinates in the Y-Z world space. Under the pinhole, we can see the point by observing on the ray that is reflected from the point and passes through the pinhole. We draw the ray in the blue color.

In a camera we commonly have some kind of sensors (e.g., camera sensors or film) to capture the light energy that the ray carries at the

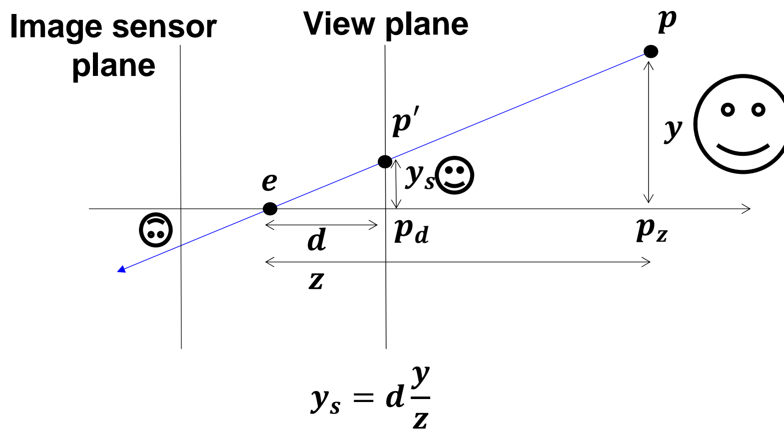

Figure 4.7: This figure illustrates how a point maps in the world space maps to one in the view plane space.

end of the optical systems behind of the eye point, i.e., focal point. In computer graphics, we, however, have such image recording plane in front of the eye position, i.e., the camera center.

Given this configuration of the view plane, our goal is to compute coordinates of the point, p', in the view plane that is projected from the 3D point p. Since the projected point p' is in the view plane, its Z-coordinate is d, which is the distance from the camera origin e to the view plane. The unknown of p' is its Y-coordinate.

To derive this, we apply properties of similar triangles between $\triangle p' e p_d$ and $\triangle p e p_z$, and we then have the following relationship based on the same proportion of same sides:

$$\frac{y_s}{d} = \frac{y}{z} \Rightarrow y_s = d\frac{y}{z}, \tag{4.10}$$

where d and z are Z-coordinates of points p_d and p_z, respectively.

The next question is how to represent this equation in a matrix form. The bottom line is how to represent $\frac{1}{z}$ in a matrix form. We address this problem by utilizing the homogeneous coordinate with the following simple matrix form:

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{4.11}$$

The trick is on the homogeneous coordinate. In this case, the homogeneous coordinate after applying the perspective matrix is set to the depth of the point, i.e., $w = z$. We then have the following the homogeneous divide and accomplish the perspective projection:

$$w = z, x' = \frac{x}{w} = \frac{x}{z}, y' = \frac{y}{w} = \frac{y}{z}, z' = 0. \tag{4.12}$$

The final homogeneous coordinate after the homogeneous divide is $1 = \frac{w}{w}$.

We also define a view volume for the perspective projection and convert it to the unit view volume, followed by mapping to the NDC space. Based on this, we can also setup a perspective projection matrix for the NDC space. In an old OpenGL version, this function is supported by call $glFrustum(\cdot)$ or $gluPerspective(\cdot)$.

4.2.2 Common Questions

Can we support other projections than orthographic and perspective projections? For example, a projection simulating the image observed from bug's eyes? What if this projection is not represented as a simple matrix? Yes, we can support many other projections that are represented as some mathematical equations. Also, current GPU can support arbitrary projections although the projection is not represented as a simple matrix.

I felt that there are something missed in the image generated by using perspective projection. Then, I realized that those images do not have effects like out-of-focusing and in-focusing. How can we support these effects? To correctly simulate these kinds of effects, we need to simulate a lens that we are using in camera. This can be supported by using ray tracing, but may take long computation time. Instead, we can mimic similar effects by considering depth values of rasterized objects. For example, the depth values of the rasterized objects are far away from the user-defined focal depth, we blur the image of the object. This is not a correct solution, but a hacky solution that can run very fast in the rasterzation rendering mode.