Applications of Linear Algebra: Computer Graphics

Yitong (Tony) Zhao
Linear Algebra, Period 7
Princeton International School of Mathematics and Science

Contents

1. Introduction 2
   1.1. The Rendering Pipeline 2
   1.2. MVP (Model-View-Projection) Transformation 3
2. Model Transformation 3
   2.1. Linearity of Translation Transformation 4
   2.2. Affine Space and Homogeneous Coordinates 5
      2.2.1. Definition 6
      2.2.2. Translation 6
      2.2.3. Addition 7
      2.2.4. Projection 8
   2.3. Arbitrary Rotation 9
      2.3.1. 2D 9
      2.3.2. 3D 10
3. View Transformation 11
4. Projection Transformation 13
   4.1. Orthographic 13
      4.1.1. Justifications 13
      4.1.2. Mathematical Representation 15
   4.2. Perspective 15
5. Summary 18
6. Appendix 18
   6.1. Code of Animation of 2D Shear Transformation 18
   6.2. Code of Animation of 2D Rotation Around Arbitrary Point 19
   6.3. Code of Graph of z Mapping in Perspective Projection 20
Bibliography 21

1. Introduction

Linear algebra, a branch of mathematics focusing on vector spaces and linear mappings between them, plays a foundational role in various scientific and engineering disciplines. One of its most profound applications is computer graphics, where it serves as the backbone for numerous processes and algorithms that enable the creation and manipulation of visual content and 3D models.

Computer graphics (CG) is a broad and diverse subject that includes multiple fields [1]: Geometry, ways to represent and process surfaces; animation, ways to represent and manipulate motion; rendering, algorithms to reproduce light transport; and imaging, image acquisition or image editing. However, the fundamental problem computer graphics is trying to solve can be easily described: transforming knowledge, or data, into images and animations in a more realistic and efficient way.

Rendering, as a field of CG, is the core of solving such transformation, as it transforms 3D models into 2D screen space. Two mainstream rendering techniques currently exist: rasterization and ray racing, the former being more efficient and the latter being more realistic. Although the former requires more linear algebra when transforming coordinates from 3D to 2D, both techniques rely heavily on linear algebra to perform transformations, such as projection, rotation, and scaling, on 3D models.

Due to the heavy involvement of linear algebra in rasterization-based rendering, this report will focus on the applications of linear algebra in rasterization-based rendering, specifically in the context of transforming 3D models into 2D screen space.

1.1. The Rendering Pipeline

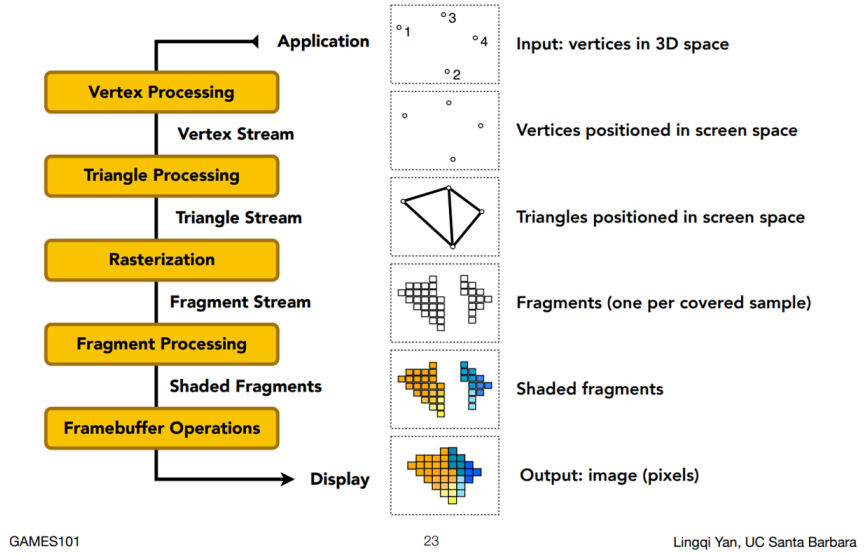


Figure 1: Figure 1: The rendering pipeline of rasterization-based rendering. [2]

Figure 1 shows the overall structure of the rasterization-based rendering pipeline, including steps like vertex processing, triangle processing, rasterization, fragment processing, and fame buffer operations. As a report on the applications of linear algebra, the first step before discussing the details of the maths involved is to identify the parts of the pipeline that require linear algebra.

Among all the steps in the pipeline, the vertex processing step is where most of the linear algebra operations take place. In this step, the 3D models are transformed into 2D screen space

through MVP (model, view, and projection) transformation. The later steps in the pipeline arrange the vertexes into triangles, which is the basic unit in CG, and then rasterize (or fill) the triangles into pixels on the screen. The fragment processing step then determines the color of each pixel, and the frame buffer operations step stores the final image in the frame buffer to be displayed on the screen.

1.2. MVP (Model-View-Projection) Transformation

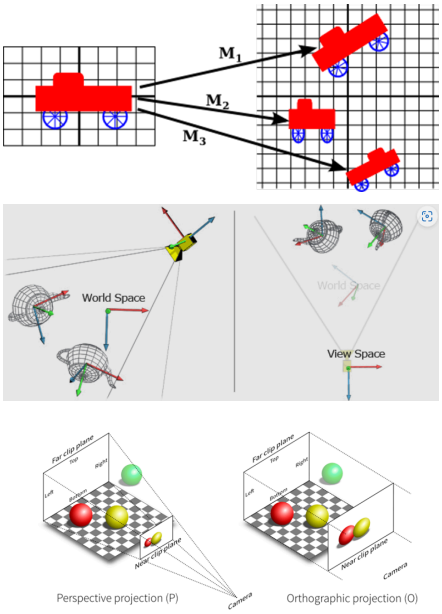


Figure 2: Illustration of the model, view, and projection transformations.

Figure 2 visually illustrated the MVP transformation. The model transformation is used to translate, rotate, and scale any 3D models in a particular scene. After that, we apply view transformation to adjust the world's coordinate system into camera's coordinate system so that the camera points to the $-z$ direction. Finally, we apply projection transformation to project the 3D models into canonical cube space, which is then transformed into screen space through viewport transformation.

This three-step transformation is analogous to the process of taking a group photo in the following process:

1. Arrange everyone into a good position (model transformation): in this step, people, or 3D models in the scene, are translated, rotated, and scaled to fit into the camera.
2. Adjust the camera's position and angle (view transformation): in this step, the camera's position and angle are adjusted to capture the scene from a particular perspective.
3. Project the 3D scene into a 2D image (projection transformation): in this step, the 3D scene is projected into a 2D image, which is then transformed into screen space.

2. Model Transformation

For model transformation, we would like to represent scale, shear, rotation, reflection, and translation in a matrix form. Except for translation, all other transformations can be represent by a 2×2 matrix for 2D space, which can be found in the following table:

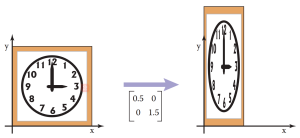
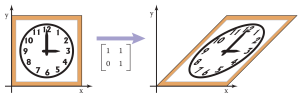
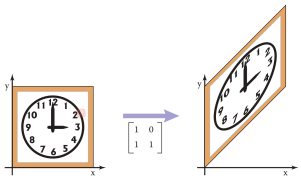
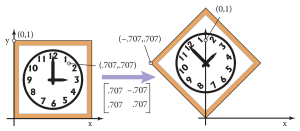
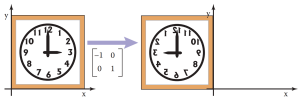
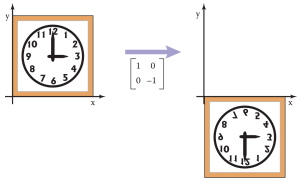
Transformation type	Illustration	Matrix representation	General form	explaination
Scale		$\begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix}$	$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$	where s_x and s_y are the scaling factors in the x and y directions, respectively.
x-axis Shear		$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$	where k is the shear factor. specifically, $k = \tan(\theta)$, where θ is the angle between the y-axis and the sheared y -axis.
y-axis Shear		$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$	where k is the shear factor. specifically, $k = \tan(\theta)$, where θ is the angle between the x-axis and the sheared x -axis.
rotation		$\begin{bmatrix} .707 & -.707 \\ .707 & .707 \end{bmatrix}$	$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$	where θ is the rotation angle.
reflection about y-axis		$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	This matrix reflects the x-axis about the y-axis.
reflection about x-axis		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	This matrix reflects the y-axis about the x-axis.

Table 1: matrix representation of 2D transformations, illustrations come from [3]

Table 1 shows different 2D transformations using matrices, and most of them can be easily extended into 3D space, which is commonly used in CG. For rotation, the matrix shown in the table is limited to rotation around the origin. However, rotation around arbitrary points is a necessary operation in CG. This part is thus introduced in Section 2.3. Translation is also a necessary part of model transformation, considering that any game characters or objects need to move around. However, when constructing a list of transformations like Table 1, one might find that it is especially hard to find a 2×2 matrix for 2D translations or 3×3 matrix for 3D translations.

2.1. Linearity of Translation Transformation

To validate that it is actually not possible to represent translations of some dimension with a matrix of the same dimension, we can check if a translation transformation is linear, as all linear transformations can be represented by matrices. A transformation T is linear if it satisfies the following property.

$$T(a\vec{x} + b\vec{y}) = aT(\vec{x}) + bT(\vec{y}) \tag{1}$$

where a and b are scalars, and $\vec{x}$ and $\vec{y}$ are vectors.

We define a translation transformation T to be verified as:

$$T(\vec{p}) = \begin{bmatrix} p_x + 1 \\ p_y \end{bmatrix} \tag{2}$$

Which basically means to translate a point p in the x-axis by 1 unit. We first evaluate the left side of Equation 1 by using $x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$:

$$\begin{aligned} a\vec{x} + b\vec{y} &= a\begin{bmatrix} 1 \\ 0 \end{bmatrix} + b\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \\ T(a\vec{x} + b\vec{y}) &= T\left(\begin{bmatrix} a \\ b \end{bmatrix}\right) = \begin{bmatrix} a + 1 \\ b \end{bmatrix} \end{aligned} \tag{3}$$

We then evaluate the right side of the equation:

$$\begin{aligned} aT(\vec{x}) &= aT\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = a\begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2a \\ 0 \end{bmatrix} \\ bT(\vec{y}) &= bT\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = b\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} b \\ b \end{bmatrix} \\ aT(\vec{x}) + bT(\vec{y}) &= \begin{bmatrix} 2a \\ 0 \end{bmatrix} + \begin{bmatrix} b \\ b \end{bmatrix} = \begin{bmatrix} 2a + b \\ b \end{bmatrix} \end{aligned} \tag{4}$$

By equating Equation 3, the left side of Equation 1, and Equation 4, the right side of Equation 1, we can see that the translation transformation is not linear as the two sides are not equal:

$$\begin{bmatrix} a + 1 \\ b \end{bmatrix} \neq \begin{bmatrix} 2a + b \\ b \end{bmatrix} \tag{5}$$

2.2. Affine Space and Homogeneous Coordinates

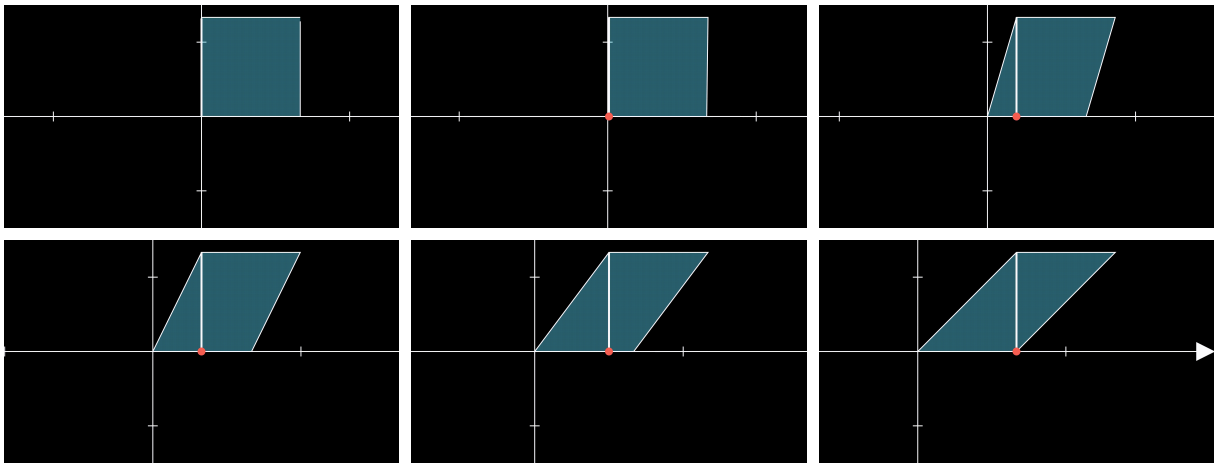


Figure 3: Animation (frame of) of 2D shear transformation $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ with a projection to x-axis. Code for this animation can be found at Section 6.1

Figure 3 shows the animation of a 2D shear transformation. Although the square as a whole is not translated but sheared, if we closely observe a part (for example, the point $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$) of it, we can see that it is actually translated in the x-axis. Hence, if we only consider the x-coordinate of this space or the projection to the x-axis, it is translated by 1 unit. With this idea in mind, we can

introduce the concept of affine space and homogeneous coordinates to represent translations linearly.

2.2.1. Definition

The following list shows how to represent points, vectors, and arbitrary coordinates in homogeneous coordinates [2]:

1. For a point, its coordinate is defined as $[x, y, z, 1]^T$
2. For a vector, its coordinate is defined as $[x, y, z, 0]^T$
3. For an arbitrary coordinate, $[x, y, z, w]^T = \left[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right]$, given $w \neq 0$.

With this definition, we can represent translations linearly by using a 4×4 matrix, as we're essentially defining a shear transformation in the 4D space. Then, it is projected into 3D space to obtain a translation transformation. The following shows how to represent a translation using matrices:

2.2.2. Translation

$$\mathrm{Tr}\left[t_x, t_y, t_z\right] = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Where t_x, t_y, and t_z are the amount of translation in the x, y, and z directions, respectively.

We can verify this matrix shown in Equation 6 by applying it to a point $\dot{p} = [x, y, z, 1]^T$:

$$\mathrm{Tr}\left[t_x, t_y, t_z\right]\dot{p} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} \tag{7}$$

Similarly, we can verify it by applying it to a vector $\vec{v} = [x, y, z, 0]^T$:

$$\mathrm{Tr}\left[t_x, t_y, t_z\right]\vec{v} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} \tag{8}$$

Interestingly, it can be observed from Equation 7 and Equation 8 that applying the same translation matrix to a point and a vector results in different outcomes. This is because points and vectors are inherently different in affine space (no fixed origin), as represented in the definition.

The result from Equation 7 and Section 2.3 can also be intuitively understood by considering the meaning of a vector is to represent a direction regardless of starting points, which should not be affected by translations, while a point is a position, which can be translated, or moved.

To ensure that the problem introduced in Section 2.1 is completely solved, we can verify if the translation defined by Equation 6 is linear by using Equation 1. To simplify the procedure, we can use a 2×2 translation.

$$\mathrm{Tr}[t_x, t_y] = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$\text{Evaluate the left side} : T(ax + by)$$

$$ax + by = a\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} ax_1 + by_1 \\ ax_2 + by_2 \\ ax_3 + by_3 \end{bmatrix}$$

$$T(ax + by) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ax_1 + by_1 \\ ax_2 + by_2 \\ ax_3 + by_3 \end{bmatrix} = \begin{bmatrix} ax_1 + by_1 + t_x(ax_3 + by_3) \\ ax_2 + by_2 + t_y(ax_3 + by_3) \\ ax_3 + by_3 \end{bmatrix}$$

$$\text{Evaluate the right side} : aT(x) + bT(y) \tag{9}$$

$$aT(x) = a\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} ax_1 + at_x x_3 \\ ax_2 + at_y x_3 \\ ax_3 \end{bmatrix}$$

$$bT(y) = b\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} by_1 + bt_x y_3 \\ by_2 + bt_y y_3 \\ by_3 \end{bmatrix}$$

$$aT(x) + bT(y) = \begin{bmatrix} ax_1 + at_x x_3 \\ ax_2 + at_y x_3 \\ ax_3 \end{bmatrix} + \begin{bmatrix} by_1 + bt_x y_3 \\ by_2 + bt_y y_3 \\ by_3 \end{bmatrix} = \begin{bmatrix} ax_1 + by_1 + at_x x_3 + bt_x y_3 \\ ax_2 + by_2 + at_y x_3 + bt_y y_3 \\ ax_3 + by_3 \end{bmatrix}$$

From Equation 9, we thus verified that translations are linear under homogeneous coordinates.

2.2.3. Addition

Since points and vectors are treated differently in homogeneous coordinates, the operation between them will also be different from normal vector arithmetics. The following is a list of different situation and their justification:

1. $\text{point}(1) + \text{vector}(0) \to \text{point}(1)$: The parenthesis after "point" or "vector" represents the value of the last component of a point and vector in homogeneous representation. Here, we found that the addition of a point with a vector is a point. This is reasonable as a vector represents displacement in a certain direction. After such displacement, a point will be moved to another point.
2. $\text{vector}(0) + \text{vector}(0) \to \text{vector}(0)$: the addition of two vectors will result in a new vector according to the triangle sum rule (head to tail).
3. $\text{vector}(0) - \text{vector}(0) \to \text{vector}(0)$: same as vector plus vector as the subtraction of two vectors is equivalent to the addition of the first vector and the negative of the second vector.
4. $\text{point}(1) - \text{point}(1) \to \text{vector}(0)$: the subtraction of two points will result in a vector. This is reasonable as the subtraction of two points is essentially the displacement between two points.
5. $\text{point}(1) + \text{point}(1) \to \text{midpoint}(2)$: This is a little more complicated than the previous case. According to the definition introduced in Section 2.2.1, we know that $[x, y, z, w]^T = \left[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right]^T$. Hence $[a + b, 2]^T = \left[\frac{a+b}{2}, 1\right]^T$. Alternatively, as the addition of two points cannot be well understood geometrically, we can express such additions in another way:

$$\begin{aligned}\dot{a}+\dot{b} &= \frac{1}{2}\dot{a}+\frac{1}{2}\dot{b} \quad \text{Note: } \tfrac{1}{2}\dot{a}=\dot{a} \text{ as } [x,w]^T=[kx,kw]^T \\ &= \dot{a}-\frac{1}{2}\dot{a}+\frac{1}{2}\dot{b} \\ &= \dot{a}+\frac{1}{2}\big(\dot{b}-\dot{a}\big)=\dot{a}+\frac{1}{2}\vec{ab}\end{aligned} \tag{10}$$

From Equation 10, we expressed the addition of two points into the addition of point and vector, making it geometrically meaningless. Extending from Equation 10, we can generalize the addition of two points x and y as calculating some points located in the line connected by x and y.

$$\begin{aligned}a\dot{x}+b\dot{y} &= (a+b)\dot{x}+b(\dot{y}-\dot{x}) \\ &= \dot{x}+\frac{b}{a+b}\overrightarrow{xy} \quad \begin{array}{l}\text{Scale by constant will not affact} \\ \text{since } [x,w]^T=[kx,kw]^T\end{array}\end{aligned} \tag{11}$$

2.2.4. Projection

Our world is in 3D space, but our visions are in 2D space. Hence, we view our world through projection to convert a space in 3D to 2D. Projection, although useful, also brought complications in comparison with regular Euclidean Geometry.

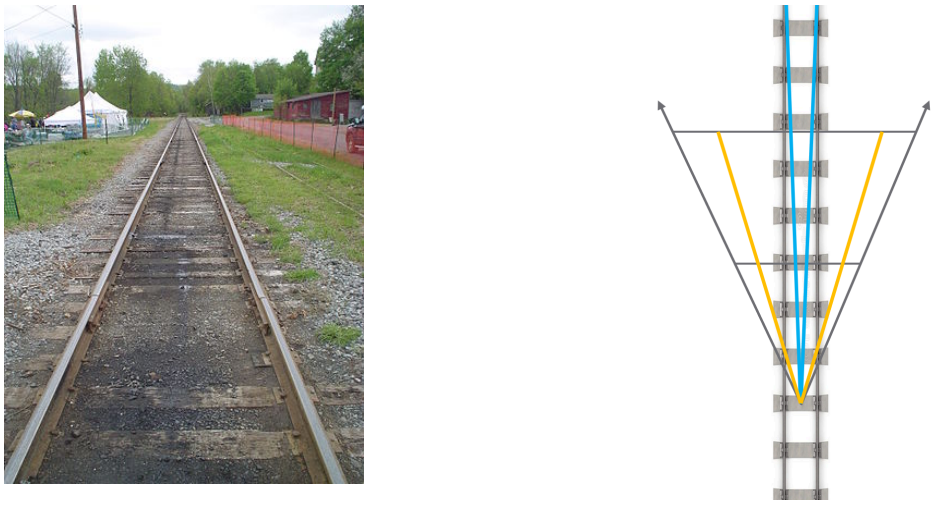


Figure 4: Illustration of railroad vanishes at a point in the distance [4].

For example, the two rails on a railroad track are parallel, which, according to Euclidean Geometry, should never meet. However, when we look at or take a picture of a railroad track, the two rails will meet at a point in the distance, as shown on the left side of Figure 4. This is because of the projection of 3D space into 2D space.

This phenomenon can be partly understood by the right side of Figure 4. In human vision, any points located on the same line that extend from our eye will become the same point. For example, anything on the yellow line on the right side of Figure 4 will be projected to the same point on the screen or our eye. As a result, the angle between the verticle line and the line connecting the railroad and our eye will become smaller and smaller as the distance increases, making the two sides of the railroad indistinguishable.

To reflect such special property in projective geometry, we would like to make all points on the same line to be essentially the same. A line in 2D space can be represented by the following equation [5] (the following also works for a plane in 3D):

$$ax + by + c = 0 \tag{12}$$

Which can be re-written in the form of dot products:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0 \tag{13}$$

The left side will be the normal vector of the line, denoted as $\vec{n}$, and the right side will be a point $\dot{p}$. If we scale the point by some constant, it will still be on the line:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} kx \\ ky \\ k \end{bmatrix} = 0 \tag{14}$$

We conclude that if $\vec{n} \cdot \dot{p} = 0$, then $\vec{n} \cdot \dot{kp} = 0$. Thus, the points $\dot{p}$ and $\dot{kp}$ are equivalent under homogeneous coordinates as they're all on the same line.

With such an understanding of the homogeneous coordinate, we can now interpret the railroad vanishing problem. A rail in a railroad track can be represented as a parameterized vertical line like:

$$R(t) = \begin{bmatrix} c \\ t \end{bmatrix} \tag{15}$$

Here, c is a constant. As the rail extends to the distance, the t value will increase:

$$\lim_{t \to \infty} R(t) = \begin{bmatrix} c \\ \infty \end{bmatrix} = \begin{bmatrix} \frac{c}{\infty} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{16}$$

That is, regardless of the value of c, the rail will be projected to the same point on the screen as t goes to infinity (the rail becomes further and further away).

2.3. Arbitrary Rotation

2.3.1. 2D

For 2D rotation around arbitrary point (x, y), we can decompose it into the following steps:

1. Translate the point (x, y) to the origin by applying the translation matrix $\mathrm{Tr}[-x, -y]$
2. Rotate the point around the origin by applying the rotation matrix $R[\theta]$
3. Translate the point back to the original position by applying the translation matrix $\mathrm{Tr}[x, y]$

The following animation shows such steps (around point $(2, 3)$) and the comparison with direct rotation around the point $(2, 3)$:

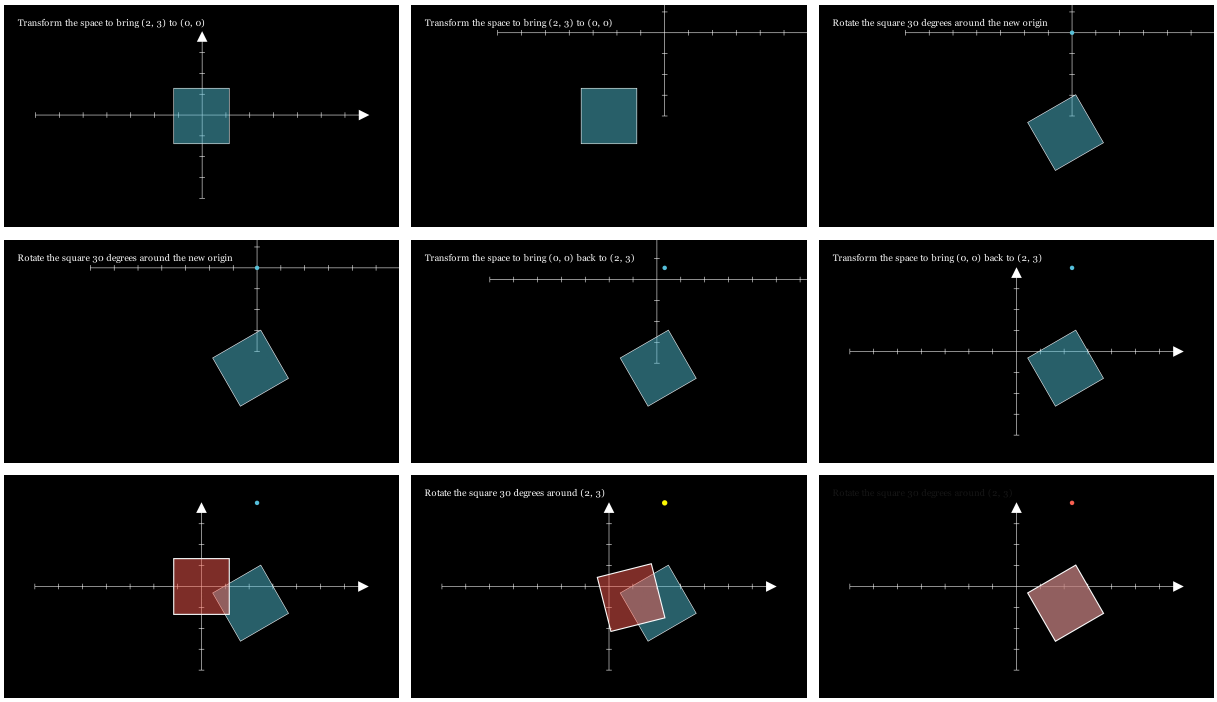


Figure 5: Animation of 2D rotation around point $(2, 3)$. The first six frames show the 3-steps method while the later three frames show directly rotating around the point. Code for this animation can be found at Section 6.2

We can explicitly write the 3-step method as follows in matrix form:

$$\begin{aligned} R[\theta, x, y] &= \mathrm{Tr}[x, y] R[\theta]\ \mathrm{Tr}[-x, -y] \\ &= \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{17}$$

2.3.2. 3D

In 3D spaces, rotations are defined by an axis $\vec{k} = \left[k_x, k_y, k_z\right]^T$ and an angle θ. Rodrigues' rotation formula is a common way to represent rotations in 3D space. The formula is as follows. Proof of the formula can be found here.

$$\begin{aligned} R &= I + \sin\theta K + (1 - \cos\theta) K^2 \\ K &= \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix} \end{aligned} \tag{18}$$

Although the proof of the formula is complicated, we can use some ways to show that it makes sense. When rotating some object, the location of the object will be changed, except for points on the rotation axis. To check if Rodrigues' rotation formula satisfies this property, we can evaluate if the eigenvector of the rotation matrix is the rotation axis, with an eigenvalue of 1.

$$\text{Evaluate } K\vec{k}:$$

$$K\vec{k} = \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix} \begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix} = \begin{bmatrix} -k_z k_y + k_y k_z \\ k_z k_x - k_x k_z \\ -k_y k_x + k_x k_y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{19}$$

$$\text{Evaluate } K^2\vec{k}:$$

$$K^2 = \begin{bmatrix} -k_y^2 - k_z^2 & k_x k_y & k_x k_z \\ k_y k_x & -k_x^2 - k_z^2 & k_y k_z \\ k_z k_x & k_z k_y & -k_x^2 - k_y^2 \end{bmatrix} \quad K^2\vec{k} = \begin{bmatrix} k_x(-k_y^2 - k_z^2) + k_x k_y^2 + k_x k_z^2 \\ \cdots \\ \cdots \end{bmatrix}$$
$$= \begin{bmatrix} -k_x k_y^2 - k_x k_z^2 + k_x k_y^2 + k_x k_z^2 \\ \cdots \\ \cdots \end{bmatrix} = 0 \tag{20}$$

$$R\vec{k} = \vec{k} + \sin\theta \cdot 0 + (1 - \cos\theta) \cdot 0$$
$$= \vec{k}$$

We can see that the rotation matrix does not change the rotation axis, which is an expected property.

3. View Transformation

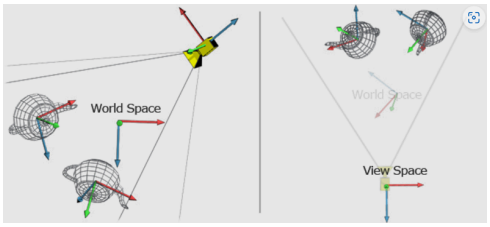


Figure 6: Illustration of the view transformation.

As stated in Section 1.2, we would like to transform the coordinate system so that the camera is located at the origin and points to the $-z$ direction (we're using $-z$ instead of z because this is a convention in computer graphics software packages like OpenGL). Such operations have the following significance:

1. Simplify the calculation when performing projection transformation: After the view transformation, we're essentially projecting the 3D scene into the xy-plane.
2. Simplify visibility test: When performing projection, we'd like to make sure that only the nearest object appears on our screen. Thus, we'll need to determine the objects' depth (distance) from the camera. After performing view transformation, this can be simply done by comparing the z coordinate.

To define the view transformation, we need to first define a camera. The following three vectors/ points are used for this purpose:

1. $\dot{e}$: a point representing the position of the camera. In Figure 7, this is represented by the cyan point.
2. $\vec{g}$: this is the gaze-at/look-at vector, which defines the direction the camera is pointing at. In Figure 7, this is represented by the cyan arrow.
3. $\vec{t}$: this is the up vector, which defines the orientation of the camera (upward orientation). In Figure 7, this is represented by the cyan arrow.

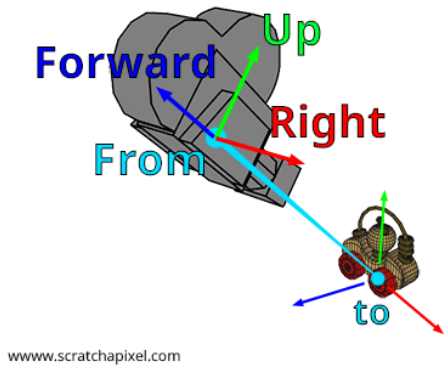


Figure 7: The three values that define a camera.

From Figure 7 and the three values we used to define a camera, we can see that the way we define a camera is similar to the way we define a coordinate system. Then, we can reframe the goal of the view transformation in a mathematical way:

1. Translate the camera to the origin: This is done by applying the translation matrix $\mathrm{Tr}\left[-e_x, -e_y, -e_z\right]$
2. Rotate the camera to align with the $-z$ direction: This is done by applying the rotation matrix that rotates the $\vec{g}$ vector to the $-z$ direction.
3. Rotate the camera to align with the y direction: This is done by applying the rotation matrix that rotates the $\vec{t}$ vector to the y direction.
4. Here, we need to align the other base vector (the red vector in Figure 7) in our camera coordinate system to the x direction. However, this is not directly defined by the three values we used to define a camera. Knowing that the third base vector is perpendicular both to $\vec{g}$ and $\vec{t}$, we can calculate it by taking the cross product of $\vec{g}$ and $\vec{t}$. In conclusion, we'd like to rotate $\vec{g} \times \vec{t}$ to x direction.

It will be easy to express the first step of the view transformation in matrix form as follows:

$$\mathrm{Tr}_{\mathrm{view}}\left[-e_x, -e_y, -e_z\right] = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{21}$$

It is not trivial to represent operations 2-4 in a rotation matrix, as it requires us to rotate a non-standard coordinate system to fit the standard one. However, the reverse process, which is to rotate the standard coordinate system to fit the non-standard one, is easy to represent. We can find the matrix for this process, then take the inverse of it:

$$R_{\mathrm{view}}^{-1} = \begin{bmatrix} \left(\vec{g} \times \vec{t}\right)_x & t_x & -g_x & 0 \\ \left(\vec{g} \times \vec{t}\right)_y & t_y & -g_y & 0 \\ \left(\vec{g} \times \vec{t}\right)_z & t_z & -g_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{22}$$

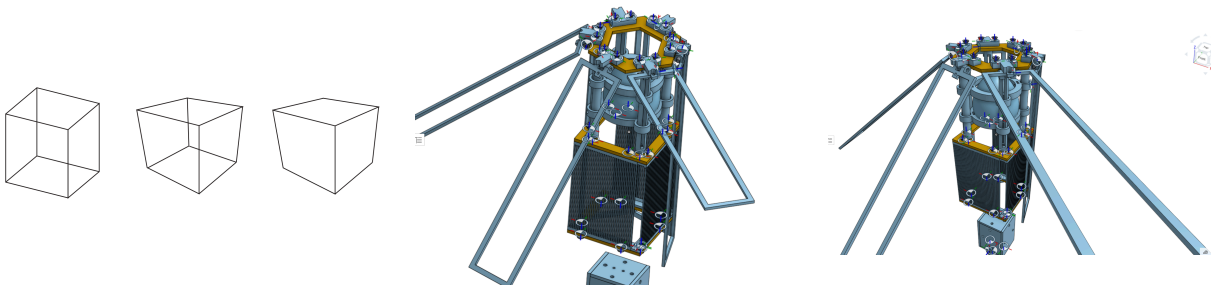
The first row represents rotating x axis to $\vec{g} \times \vec{t}$, the second row represents rotating y axis to t, and the third row represents rotating z axis to $-g$.

Noting that all rotation matrices are orthogonal, using the property of $A^{-1} = A^T$ for orthogonal matrices, we can easily take the inverse of R_{view}^{-1}, hence finding the rotation part of view transformation:

$$R_{\mathrm{view}} = \begin{bmatrix} \left(\vec{g} \times \vec{t}\right)_x & \left(\vec{g} \times \vec{t}\right)_y & \left(\vec{g} \times \vec{t}\right)_z & 0 \\ t_x & t_y & t_z & 0 \\ -g_x & -g_y & -g_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{23}$$

4. Projection Transformation

Projection is a mapping from 3D to 2D space. In computer graphics, mainly two types of projection are used: orthographic and perspective.



(a) Comparison between two types of projections when applied to a cube. The left side is the cube with orthographic projection, while the right two images are perspective.

(b) The default orthographic projection mode in Onshape when viewing my egg drop capsule model from engineering class

(c) The same egg drop capsule viewed under perspective projection in Onshape.

Figure 8: Illustration of orthographic and perspective projection.

Figure 8 shows the difference between orthographic and perspective projection. From subfigure (a), we can see that if two lines are parallel in 3D space, orthographic projection will keep them parallel in 2D space, while perspective projection will not preserve such property (explained in Section 2.2.4). Also, observing the long rectangle shown in subfigure (c), one can see that the width of the triangle seemed to be larger when it is closer to the viewer, and vice versa.

4.1. Orthographic

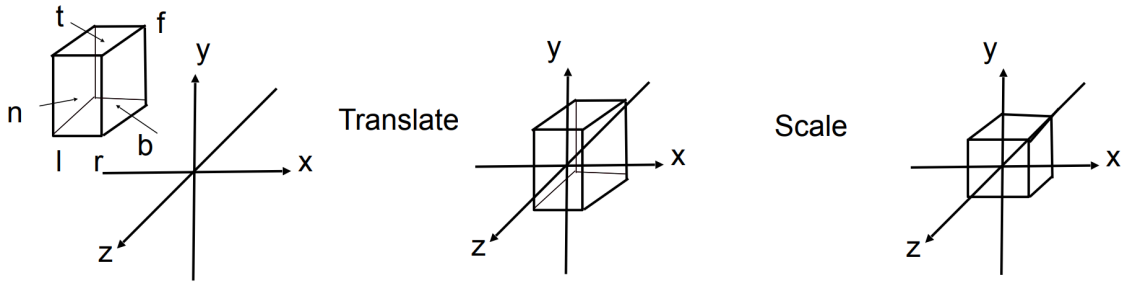


Figure 9: Illusgtration of different steps in orthographic projection [2]

Although projections should be a mapping from 3D to 2D, the projection transformations in CG are not defined in this way. In orthogonal projection, we select a cuboid $[l(\text{left}), r(\text{right})] \times [b(\text{bottom}), t(\text{top})] \times [f(\text{far}), n(\text{near})]$, then translate its center to the origin. In the end, we scale this cuboid to a canonical cube of $[-1, 1]^3$.

4.1.1. Justifications

Generally, computer science is a subject concerning practical usage. Due to the limitation of computing power and memory space, computer scientists have developed multiple approaches in CG pipeline to reduce calculation, and the adoption of orthogonal projection with canonical cubes is one such example.

In the first step, we select a cuboid for scaling. Such a cuboid should include everything that we'd like to render later in the pipeline. Very often, some objects do not need to be rendered, such as objects behind the camera or objects we don't want to show, such as objects too far away that are not visible. By selecting a cuboid, we can reduce the number of objects that need to be processed in the later pipeline, thus improving performance.

The process of excluding unwanted objects is called clipping. This process is not trivial. The smallest unit in CG is triangles; sometimes, the cuboid we selected cuts the triangles into pieces. In such cases, we need to determine which part of the triangle should be rendered and which part should be discarded. To improve the performance of such operations, dedicated circuits have been integrated into hardware like GPU (graphics processing unit). Usually, hardware performance can be significantly optimized for very specific tasks. To make the optimization for clipping enough, we make the problem very specific – making the cuboid a canonical cube.

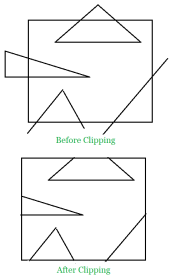


Figure 10: Illustration of clipping triangles.

The remaining operation to be justified in orthographic projection is translating the center of the cuboid to the origin. Besides simplifying and standardizing operations performed for dedicated hardware, this step helps preserve precision when scaling the selected cuboid to a canonical cube.

Since computer systems are based on digital circuits that express numbers in binary form, the precision of representing real numbers is limited. Specifically, computer systems use floating-point representation to approximate real numbers. For IEEE standard, floating point numbers are represented as:

$$V = (-1)^s \times M \times 2^E \tag{24}$$

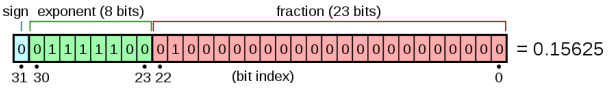


Figure 11: Illustration of floating-point number representation.

Here, s is the sign bit, M is the mantissa, and E is the exponent. Overall, this is similar to the scientific notation, where M is the significant digits and E is the power of 10. Since the digit of M does not change as the exponent increases, the precision of the number decreases as the absolute value of the number becomes more extensive. The specific precision can be found on the following table:

exponent	*range*	*half*	*float*	*double*
0	$[1, 2)$	0.0009765625	0.00000011920929	0.00000000000000002220446
1	$[2, 4)$	0.001953125	0.000000238418579	0.00000000000000044408921
2	$[4, 8)$	0.00390625	0.000000476837158	0.00000000000000088817842
9	$[512, 1024)$	0.5	0.00006103515	0.00000000000011368684
10	$[1024, 2048)$	1	0.00012207031	0.00000000000022737368
11	$[2048, 4096)$	2	0.00024414062	0.00000000000045474735
12	$[4096, 8192)$	4	0.00048828125	0.00000000000009094947
15	$[32768, 65536)$	32	0.00390625	0.00000000000072759576
16	$[65536, 131072)$	N/A	0.0078125	0.00000000000014551915
17	$[131072, 262144)$	N/A	0.015625	0.00000000002910383
18	$[262144, 524288)$	N/A	0.03125	0.00000000058207661
19	$[524288, 1048576)$	N/A	0.0625	0.00000000011641532
23	$[8388608, 16777216)$	N/A	1	0.00000000186264515
52	$[4503599627370496, 9007199254740992)$	N/A	536870912	1

Figure 12: Precision of floating-point numbers.

It can be seen that the precious would be the highest when the number is close to 1. By translating the center of the cuboid to the origin, we can ensure that the objects we'd like to render are close to the origin, thus preserving the precision of the floating-point numbers.

Preserving the precision of the floating point value is important when conducting z-tests, which are used to tell if an object is in front of another. If an object is behind, it should not be displayed on the screen. Without enough precision, a phenomenon called z-fighting will occur, where two objects are flickering on the screen as the computer wrongly determined their distance to the camera (z coordinate):

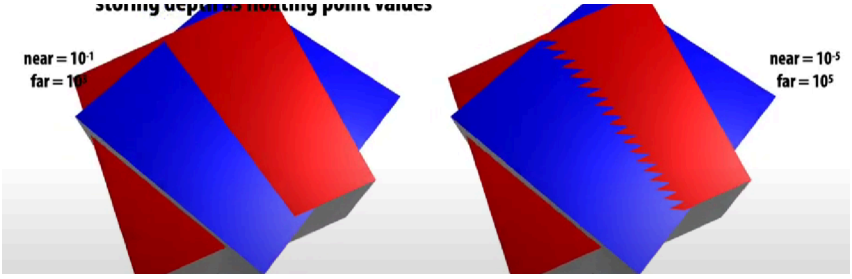


Figure 13: Illustration of z-fighting. The left side represents the case where enough precision is preserved, while the right side is the case where precision is lost. It can be observed that at the intersection point of the two cubes, a clear zig-zag line occurred.

4.1.2. Mathematical Representation

Overall, the steps after selecting the cuboid can be expressed as:

$$P_{\text{ortho}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{25}$$

The matrix at the right translates the center of the selected cuboid to the origin, while the matrix at the left scales the cuboid to a canonical cube.

4.2. Perspective

As stated in Section 2.2.4, in perspective projection, all light will converge at a certain point. This phenomenon can be illustrated by a view frustum:

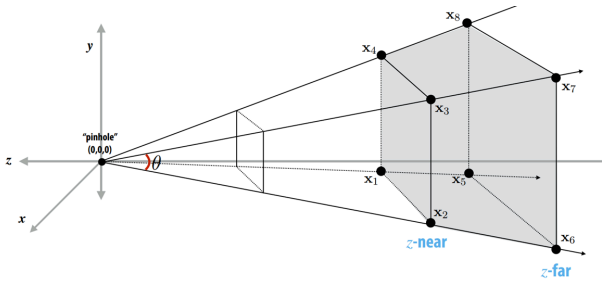


Figure 14: Illustration of the view frustum. Only objects within z-near and z-far in later steps of rendering.

The view frustum indicates the region that might appear on the screen after rendering so that its cross-sectional area becomes larger when the distance becomes further from observers.

As discussed in Section 4.1, transforming a cuboid into a canonical cube provides advantages, including clipping unnecessary objects, standardization for hardware optimization, preserving depth, etc. We also want to keep those advantages in perspective projection, so the basic idea of developing a projection matrix is to first "squeeze" the view frustum into a cuboid and then perform the orthographic projection transformation.

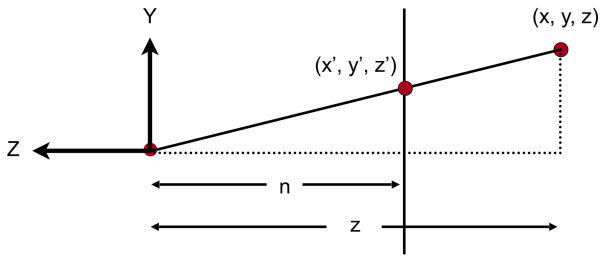


Figure 15: Side view of a view frustum. [2]

In Figure 15, n marks the near plane of the view frustum, and z marks an arbitrary plane with depth z. Our goal here is to project the point (marked by $[x, y]^T$) on the plane z to the near plane (marked by $[x', y']^T$). Notice that we want to preserve the z coordinate of the point, as it is used to determine the depth of the object.

From the illustration, we can see a similar triangle described by the following equation:

$$y' = y\left(\frac{n}{z}\right) \tag{26}$$

This works also for the x coordinate. Then, we can describe our transformation in the following form:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \xrightarrow{\text{perspective projection}} \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ z \\ 1 \end{bmatrix} \tag{27}$$

Notice that division by z, a variable, is not linear. We'll have to change the transformation if we want to represent it in a matrix form. Fortunately, with the property of homogeneous coordinates, we can represent the division by z by multiplying every component by z.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \xrightarrow{\text{perspective projection}} \begin{bmatrix} nx \\ ny \\ z^2 \\ z \end{bmatrix} \tag{28}$$

However, z^2 of the third component is still not linear, but we can first try to write the matrix representation without it:

$$P = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & m_1 & m_2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{29}$$

Here, m_1 and m_2 are the values that we need to determine as they're related to z (the first and second components of the third row correspond to x and y, thus having to be zero). To determine m_1 and m_2, we consider the following equation:

$$
\begin{aligned}
[0 \;\; 0 \;\; m_1 \;\; m_2] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} &= z^2 \\
m_1 z + m_2 &= z^2 \\
z^2 - m_1 z - m_2 &= 0
\end{aligned} \tag{30}
$$

Since this is a quadratic equation, we know that no matter how we adjust the coefficients m_1 and m_2, we can only have a maximum of two places where z coordinate is the same after projection.

Naturally, we think about the z coordinate of the near and far planes since we only render the objects between the two ends. Hence, we have the following equation:

$$
\begin{aligned}
m_1 n + m_2 &= n^2 \\
m_1 f + m_2 &= f^2
\end{aligned} \tag{31}
$$

Solving this equation we get:

$$
\begin{aligned}
m_1 &= f + n \\
m_2 &= -fn
\end{aligned} \tag{32}
$$

The complete matrix for perspective projection is:

$$
P_{\text{persp}} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & f+n & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{33}
$$

We know that the mapping of the z coordinate is not the same after the projection. With the following figure, we can better understand this mapping:

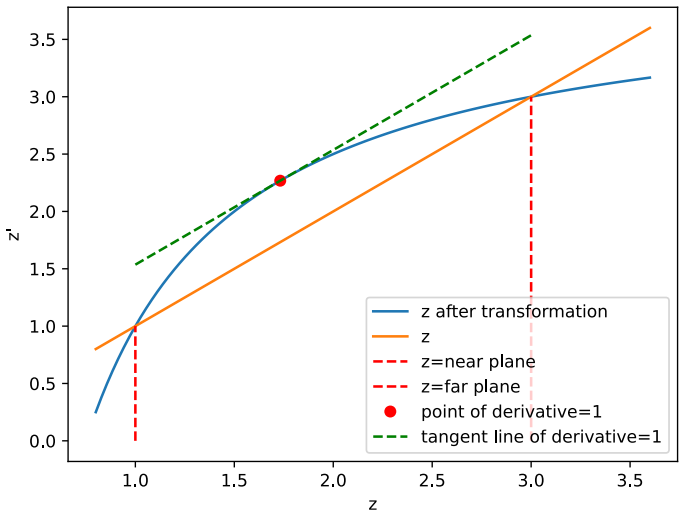


Figure 16: Mapping of the z coordinate after perspective projection. $n = 1$, $f = 3$

From Figure 16, we can first see that the order is reserved after the projection, enabling us to perform z-tests. Secondly, the z coordinate after the projection will have higher precision near the near plane. The figure shows that on the left side of the red dot, the z coordinate is

mapped to a larger range, providing more precision or distinguishability for close values, while the opposite happens on the right side of the red dot.

This property is especially favorable for CG applications since it mitigates the z-fighting problem near the observer (near plane). Although the z-fighting problem might be more severe near the far plane, it is less noticeable as objects near the far plane (far away from the observer) are usually smaller and less detailed.

To find the point where precision magnification stopped, we first express the equation for z mapping:

$$z_p = f + n - \frac{fn}{z} \tag{34}$$

We then find the place where the derivative of z' is 1:

$$\begin{aligned} \left(z_p\right)' &= \frac{fn}{z_m^2} = 1 \\ z_m &= \sqrt{fn} \end{aligned} \tag{35}$$

Before z_m, the derivative will be larger than 1:

$$\begin{aligned} \left(z_p\right)' &= \frac{fn}{z^2} > 1 \\ z < z_m &= \sqrt{fn} \end{aligned} \tag{36}$$

After z_m, the derivative will be less than 1.

5. Summary

From the previous sections, we derived the following matrices for linear transformation in rasterization-based CG:

$$p' = P_{\text{ortho}} P_{\text{persp}} V M p \tag{37}$$

Where P_{ortho} and P_{persp} are the projection matrices for orthographic and perspective projection, respectively, V is the view transformation matrix, M is the model transformation matrix, and p (in homogeneous coordinate) is the point to be transformed.

6. Appendix

6.1. Code of Animation of 2D Shear Transformation

```
from manim import *


class ShearTransformation(MovingCameraScene):
    def construct(self):
        # Define the initial square
        self.camera.frame.set_width(8)
        square = Square()
        square.set_fill(BLUE, opacity=0.5)
        square.set_stroke(WHITE, width=2)

        # Define the shearing matrix
        shear_matrix = [[1, 1], [0, 1]]
```

```
        # Move the square so that the left bottom corner is at the origin
        square.shift(UP + RIGHT)

        # Add a coordinate system
        axes = Axes(x_range=[-2, 2, 1], y_range=[-2, 2, 1])

        # Add a vertical line at y in [0, 1]
        vertical_line = Line(start=ORIGIN, end=UP * 2)
        point = Dot(vertical_line.get_start(), color=RED)
        # Add the coordinate system and square to the scene
        self.add(axes)
        self.play(Create(square), Create(vertical_line), run_time=1)
        self.wait(.5)

        # Apply the shearing transformation
        self.play(square.animate.apply_matrix(shear_matrix),
                  vertical_line.animate.shift(RIGHT * 2),
                  point.animate.shift(RIGHT * 2),
                  self.camera.frame.animate.shift(RIGHT * 2),
                  run_time=3
                  )

        self.wait(1)
```

6.2. Code of Animation of 2D Rotation Around Arbitrary Point

```
from manim import *


class RotateAroundPoint(Scene):
    def construct(self):
        # Define the initial square
        square = Square()
        square.set_fill(BLUE, opacity=0.5)
        square.set_stroke(WHITE, width=2)

        # Initial position of the square's center
        initial_position = np.array([2, 3, 0])

        # Rotation point (same as initial position of the square)
        rotation_point = initial_position

        # Add coordinate system
        axes = Axes()

        # Show initial setup
        self.play(Create(axes), run_time=0.5)
        self.play(Create(square), run_time=0.5)
        self.wait(1)

        # Transform the space to bring (2, 3) to (0, 0)
        text_transform_1 = Text(
            "Transform the space to bring (2, 3) to (0, 0)", font_size=24).to_edge(UL)
        self.play(Write(text_transform_1), run_time=1)
        # self.play(square.animate.shift(-rotation_point))
        self.play(axes.animate.shift(rotation_point))
```

```
        self.wait(1)
        self.play(FadeOut(text_transform_1))

        # Rotate the square 30 degrees around the new origin
        text_rotate_1 = Text(
            "Rotate the square 30 degrees around the new origin", font_size=24).to_edge(UL)
        rot_point_1 = Dot(rotation_point, color=BLUE)
        self.play(Create(rot_point_1))
        self.play(Write(text_rotate_1))
        self.play(Rotate(square, angle=PI/6,
                  about_point=rotation_point), Indicate(rot_point_1))
        self.wait(1)
        self.play(FadeOut(text_rotate_1))

        # Transform the space to bring (0, 0) back to (2, 3)
        text_transform_2 = Text(
            "Transform the space to bring (0, 0) back to (2, 3)", font_size=24).to_edge(UL)
        self.play(Write(text_transform_2), run_time=1)
        self.play(axes.animate.shift(-rotation_point))
        self.wait(1)
        self.play(FadeOut(text_transform_2))

        square = Square()
        square.set_fill(RED, opacity=0.5)
        self.play(Create(square), run_time=0.5)

        # Directly rotate the square around (2, 3)
        text_rotate_2 = Text(
            "Rotate the square 30 degrees around (2, 3)", font_size=24).to_edge(UL)
        rot_point_2 = Dot(rotation_point, color=RED)
        self.play(Create(rot_point_2))
        self.play(Write(text_rotate_2))
        self.play(Rotate(square, angle=PI/6,
                  about_point=rotation_point), Indicate(rot_point_2))
        self.wait(1)
        self.play(FadeOut(text_rotate_2))
```

6.3. Code of Graph of z Mapping in Perspective Projection

```
import numpy as np
import matplotlib.pyplot as plt
NEAR_PLANE = 1
FAR_PLANE = 3
def z_transform(orig_z):
    return FAR_PLANE + NEAR_PLANE - (FAR_PLANE * NEAR_PLANE) / orig_z
x_point_of_deriv_1 = np.sqrt(NEAR_PLANE * FAR_PLANE)

ORIG_Z = np.linspace(NEAR_PLANE * .8, FAR_PLANE * 1.2, 100)
Z = z_transform(ORIG_Z)
plt.plot(ORIG_Z, Z, label = 'z after transformation')
plt.plot(ORIG_Z, ORIG_Z, label='z')
plt.vlines(NEAR_PLANE, 0, NEAR_PLANE, colors='r', linestyles='dashed', label='z=near
plane')
plt.vlines(FAR_PLANE, 0, FAR_PLANE, colors='r', linestyles='dashed', label='z=far
plane')
plt.plot(x_point_of_deriv_1, z_transform(x_point_of_deriv_1), 'ro', label='point of
derivative=1')
```

```
# draw tangent line
TAN_X = np.linspace(NEAR_PLANE, FAR_PLANE, 100)
TAN_Y = TAN_X + z_transform(x_point_of_deriv_1) - x_point_of_deriv_1
plt.plot(TAN_X, TAN_Y, 'g--', label='tangent line of derivative=1')
plt.legend()
plt.savefig('./img/z_transform.svg', dpi=300)
```

Bibliography

[1] Wikipedia contributors, "Computer graphics (computer science)." [Online]. Available: https://en.wikipedia.org/wiki/Computer_graphics_(computer_science)

[2] L. Yan, "GAMES101: 现代计算机图形学入门 (Introduction to Modern Computer Graphics)." [Online]. Available: https://sites.cs.ucsb.edu/~lingqi/teaching/games101.html

[3] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.

[4] Wikipedia contributors, "Vanishing point." [Online]. Available: https://en.wikipedia.org/wiki/Vanishing_point

[5] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.