Computer Graphics
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The course outline is:

- Rendering pipeline, and hardware and software architectures for 3D graphics.

- Basic transformation: translation, rotation, scaling and projection.

- Basic of computational geometry, clipping and hidden surface removal.

- Lighting: light sources, materials, shaders, surface normal.

- Texture: projection, mapping, texture animation, alpha mapping, bump mapping, normal mapping.

- Advanced effects: reflection maps, BRDF models, environment maps, global illumination maps.

- Animation: scene graph, Bézier curves, quaternion.

Contents

Introduction

1.1 Graphic adapter

Graphics adapters partition the screen into a grid of discrete elements known as pixels, each capable of displaying a specific color. These pixels collectively render images sampled from spatial data, enabling the adapter to accurately display them on a monitor. To facilitate this process, the adapter is equipped with dedicated memory known as video memory (VRAM).
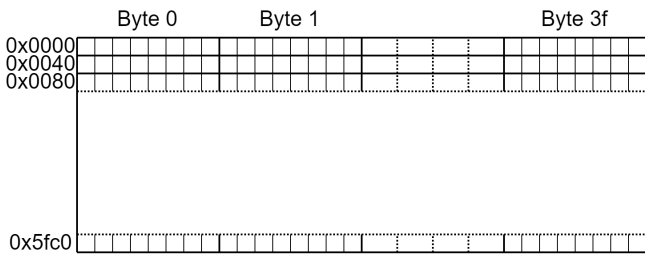
Figure 1.1: VRAM's structure

A segment of the VRAM, referred to as the screen buffer or frame buffer, contains encoded color information for all the pixels displayed on the screen. A specialized component on the graphics card, such as the RAMDAC for analog displays, utilizes this information from the VRAM to construct the image on the display. Users typically do not interact directly with the screen buffer; instead, they store images in various sections of the VRAM. Subsequently, users compose the on-screen image by issuing commands to the graphics adapter. These commands can include drawing points, lines, and other shapes, writing text, transferring raster images from the VRAM, performing 3D projections, and applying deformations and effects to the images. By combining these commands with the data stored in the VRAM, the adapter constructs the final image and transmits it to the display.

Performing complex operations involves various tasks such as interacting with multiple displays, managing multiple graphic adapters, or displaying multiple images simultaneously (e.g., stereoscopy), which can introduce significant complexity.

Vulkan Graphics adapters are typically developed in conjunction with their software drivers, ensuring that programmers interface with the hardware through libraries provided by the man-

ufacturer rather than directly accessing video card registers. Vulkan exemplifies a platform-independent standard set of procedures that streamlines the access of graphic card functionalities by application code.

1.2 Colors

Color coding is a fundamental aspect of computer graphics, dictating how colors displayed on-screen are represented as sequences of binary digits. Typically, on-screen colors are encoded using the RGB system, which stands for Red-Green-Blue.

1.2.1 Physical phenomena

In terms of physics, the color of light is determined by the wavelength of the photons it emits. When a light source emits photons of various wavelengths, objects interact with these photons based on their composition, either reflecting or absorbing them at different intensities. The wavelengths of reflected photons contribute to the primary colors of an object. These reflected photons are then focused by the lens of the human eye onto the retina, where specialized cells called rods and cones reside. Rods are sensitive to light intensity, while cones are sensitive to light color. Through these cells, the brain processes visual information. There are three distinct types of cones distributed evenly across the retina, each responsive to a specific portion of the light spectrum. The brain integrates signals from these cones to perceive a given color.
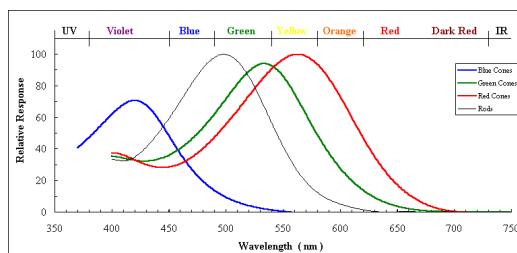
Figure 1.2: Human cones wavelength

1.2.2 Color reproduction

Color reproduction operates on the principle inverse to human vision, employing distinct emitters for each color perceptible by specific types of cones in the eye. As the wavelengths perceived by the three types of cones primarily correspond to red, green, and blue, different hues are created by blending the light from these three primary colors. When observing a display from a sufficient distance, the human eye naturally blends the primary hues, reproducing the stimuli associated with a combined color.

By mixing two of the three primary colors, such as red and green or blue and red, we produce secondary colors like yellow, cyan, or magenta. Mixing all three primaries yields white, while adjusting the proportions of the three colors enables the reproduction of various hues.

Comparing the wavelengths of photons sensed by cones with the frequency spectrum helps understand why cyan results from blending blue and green, and yellow emerges from mixing red and green. The visible spectrum to the human eye is broad, characterized by a parabolic shape.

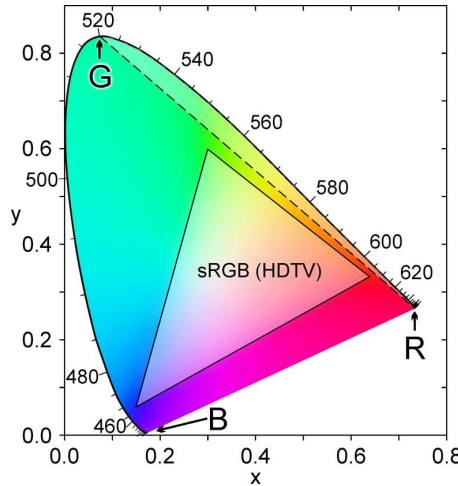Figure 1.3: Colors perceived by human eyes and reproduced by a monitor

The range of colors that a monitor can display corresponds to a cube, with the red, green, and blue components positioned along the x, y and z-axis, respectively.

The levels of these components translate into electrical signals that regulate the intensity of light emitted by the screen for each primary color. In digital systems, these signals are typically generated through DAC (Digital-to-Analog Converter) conversion, resulting in quantization, which reduces the number of available colors.

Different monitors may assign various frequencies of the spectrum to each primary color and encode their levels differently. Color profiles are used to compensate for these variations, ensuring consistent behavior across different adapters.

1.3   Image resolution

Image resolution refers to the pixel density within a given physical unit, typically measured in DPI (Dots Per Inch). In the context of raster graphic devices, resolution is relative to the size of the monitor rather than being an absolute metric as in printed images. Consequently, the resolution of a screen specifies the number of pixels displayed horizontally (s_w) and vertically (s_h). It's important to note that pixels may not have a square shape, and as a result, the horizontal resolution often differs from the vertical resolution.

1.3.1   Coordinates

The coordinate system utilized is Cartesian, with the origin situated in the top-left corner. The x-axis progresses horizontally from left to right, while the y-axis ascends vertically from top to bottom.
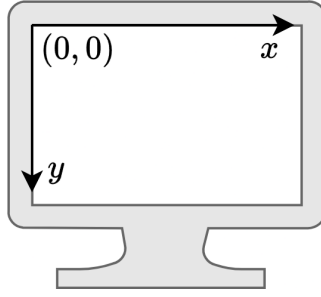
Figure 1.4: Pixel coordinates

The coordinate system adheres to a left-handed convention, signifying that the y-axis extends in the opposite direction compared to the conventional Cartesian system. As a consequence, the integer values of x and y coordinates fall within the range:

$$0 \leq x \leq s_w - 1$$

$$0 \leq y \leq s_h - 1$$

where s_w and s_h represent the horizontal and vertical dimensions of the screen, respectively. It's noteworthy that coordinates may exceed the boundaries of the screen.

Definition (*Clipping*). Clipping involves trimming the primitives to exhibit only their visible segments.

Failure to execute clipping may result in undesirable outcomes such as wrapping around or writing to unallocated memory space, often leading to irreparable errors depending on the hardware.

Contemporary displays offer diverse resolutions, sizes, and form factors. Applications strive to maintain consistent content presentation across varying resolutions, sizes, and screen shapes while harnessing the full potential of the display's capabilities.

1.3.2 Normalized screen coordinates

Normalized screen coordinates provide a standardized method for addressing points on a screen in a manner that is independent of the device's size and shape. As screens may possess varying aspect ratios and non-square pixels, the objective is to represent the same scene consistently, irrespective of the actual proportions or resolution, by adjusting or adding features based on available space. This concept extends to windows in conventional operating systems, which users can freely resize.

Many applications render images to memory, where the proportions of the display area can be arbitrary. Normalized screen coordinates utilize a Cartesian coordinate system, with x and y values ranging between two canonical values (commonly between -1 and 1, although other standards like 0 and 1 also exist), and axes oriented along specific directions.
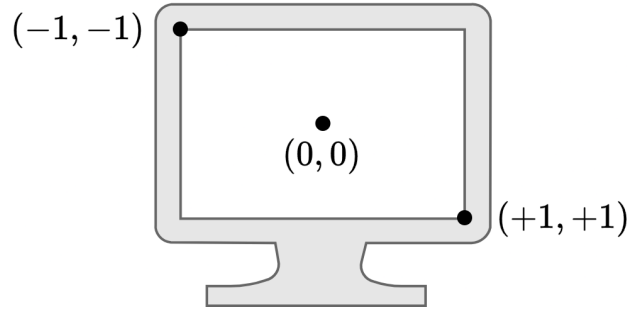
Figure 1.5: Normalized screen coordinates

For example, both OpenGL and Vulkan employ normalized screen coordinates within the $[-1, 1]$ range. However, OpenGL's y-axis increases upwards, while Vulkan adheres to the convention of pixel coordinates with the y-axis descending downwards.

Given a screen (or window, or memory area) resolution of $s_w \times s_h$ pixels. Pixel coordinates (x_s, y_s) can be derived from normalized screen coordinates (x_n, y_n) using a straightforward relation. For Vulkan, the transformation is expressed as:

$$\begin{cases} x_s = (s_w - 1) \cdot \dfrac{x_n + 1}{2} \\ y_s = (s_h - 1) \cdot \dfrac{y_n + 1}{2} \end{cases}$$

As mentioned, screen buffers are typically accessed through their drivers and specific software libraries. Programs generally utilize normalized screen coordinates, except in cases where they operate at a low level and must directly interact with the frame buffer.

1.3.3   Graphics primitives

Procedures responsible for drawing simple geometric shapes on a screen, utilizing a 2D coordinate system, are known as 2D graphics primitives. Modern graphics adapters facilitate drawing operations by supporting three fundamental types of primitives: points, lines, and filled triangles.

These primitives work by manipulating and connecting points on the screen, identified by a pair of coordinates defined with a two-component vector. These coordinates are integer values measured in pixels. The primitives employed for screen drawing (or within windows) automatically handle the calculations to determine the appropriate pixels on the screen based on normalized screen coordinates.

Point   Drawing a point entails setting the color of a pixel at a specified position on the screen. The graphic primitive responsible for this action is typically called `plot()`. It requires parameters such as the coordinates of the pixel to be set (x, y) and its color (r, g, b).
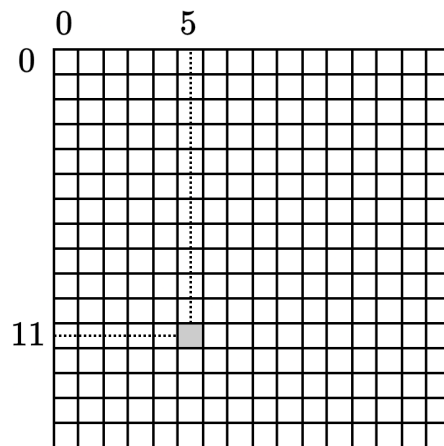
Figure 1.6: `plot(x:-0.312, y:0.562, r:0.8, g:0.8, b:0.8)`

Line The line primitive connects two points on the screen with a straight segment. It necessitates the coordinates (x_0, y_0) of the starting point and (x_1, y_1) of the end point, along with the color definition.
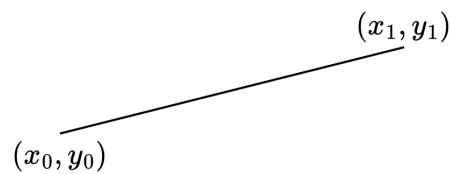
Figure 1.7: Line primitive

Filled triangle Filled triangles serve as the foundation of 3D computer graphics. They are defined by the coordinates of their three vertices (x_0, y_0), (x_1, y_1), and (x_2, y_2), along with their corresponding color (r, g, b).
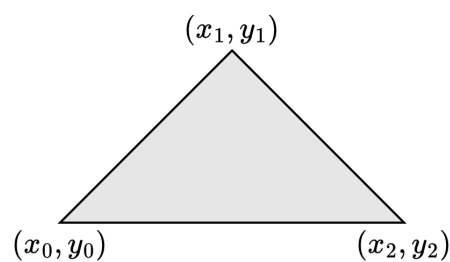
Figure 1.8: Filled triangle primitive

Three-dimensional geometry

2.1 Homogeneous coordinates

To depict objects in a 3D space effectively, we must select a suitable coordinate system. Homogeneous coordinates are employed for this purpose.

In homogeneous coordinates, a point within the 3D space is defined by four values: x, y, z, and w. The x, y, and z coordinates denote the point's position in the 3D space, while the w coordinate determines a scale, influencing the units of measurement utilized by the other three coordinates. In this system, all coordinates representing the same point (with varying w values) are linearly dependent.

The x, y, and z coordinates of the vector with $w = 1$ specify the actual position of the point in the 3D space. Given that all vectors representing the same point are linearly dependent, we can obtain the one with $w = 1$ by dividing the first three components (x, y, z) by the fourth component, w.

We can determine the Cartesian coordinates (x', y', z') corresponding to any point in homogeneous coordinates (x, y, z, w) as follows:

$$(x', y', z') = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w} \right)$$

Conversely, we can straightforwardly convert a point with Cartesian coordinates (x, y, z) into homogeneous coordinates by appending a fourth component $w = 1$:

$$(x', y', z') = (x, y, z, 1)$$
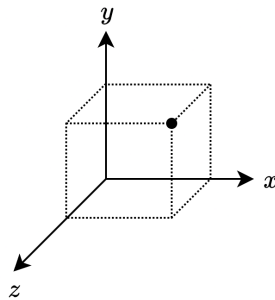
Figure 2.1: Coordinates

2.2 Affine transforms

The act of altering the coordinates of an object's points is termed a transformation. In three-dimensional space, transformations can be intricate, potentially relocating all points of the object. Nevertheless, a significant and extensive array of transformations can be encapsulated using a mathematical concept known as affine transforms.

Objects in 3D space are delineated by the coordinates of their points. Through the application of affine transformations to these point coordinates, objects can be translated, rotated, or scaled within the 3D space.

Affine transforms are typically categorized into four classes: translation, scaling, rotation, and shear. When translating, rotating, or scaling an object, the identical transformation is applied to all its points. The resultant transformed object is derived by reconstructing the geometric primitive using the updated points.

Matrix transforms In the realm of homogeneous coordinates, 4×4 matrices serve as the tool to express various geometrical transformations. The transformed vertex p' is obtained from the original point p by a straightforward matrix multiplication with the corresponding transformation matrix M:

$$p' = M \cdot p^T = (x', y', z', 1)$$

It's worth noting that there are two conventions in use:

- The transformation matrix M appears on the left side of the multiplication.

- The transformation matrix M appears on the right side.

For consistency, we adopt the convention where the matrix appears on the left.

2.2.1 Identity transform

The identity transformation leaves the points of an object unchanged. It is represented by a 4×4 identity matrix:

$$M = I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
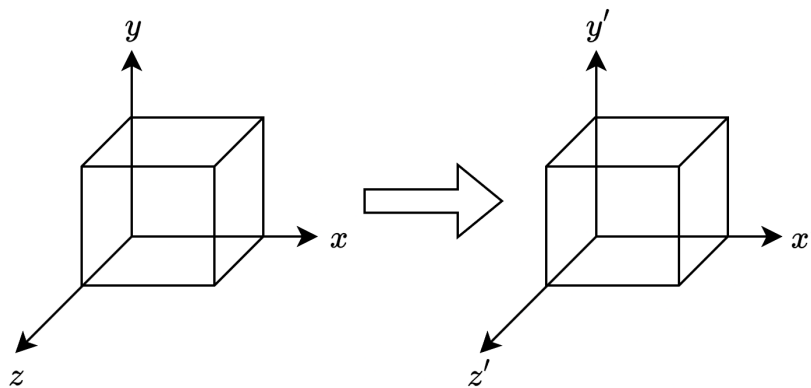
Figure 2.2: Identity transform

2.2.2 Translation

To move an object by distances d_x, d_y, and d_z along the x-axis, y-axis, and z-axis respectively, the new coordinates can be obtained by simply adding these distances to each corresponding axis:

$$\begin{cases} x' = x + d_x \\ y' = y + d_y \\ z' = z + d_z \end{cases}$$

In homogeneous coordinates derived from Cartesian coordinates, where the fourth component is always $w = 1$, the translation matrix $T(d_x, d_y, d_z)$ can be constructed by placing d_x, d_y, and d_z in the last column of the identity matrix:

$$M = T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
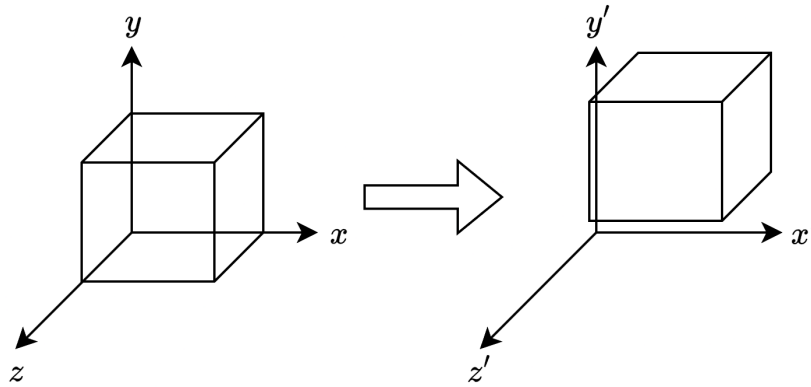
Figure 2.3: Translation transform

2.2.3 Scaling

Scaling alters the size of an object while preserving its position and orientation. It can be employed for various effects such as enlargement, shrinkage, deformation, mirroring, and flattening. Scaling transformations are characterized by a center, a fixed point that remains unchanged during the transformation. Initially, we assume the scaling center is at the origin of the 3D space.

Proportional scaling uniformly magnifies or diminishes an object by the same factor s in all directions. Consequently, it preserves the object's proportions while changing its size. The coordinates of points are modified by multiplying each coordinate by the scaling factor s:

$$\begin{cases} x' = s \cdot x \\ y' = s \cdot y \\ z' = s \cdot z \end{cases}$$

Depending on the value of s, the transformation can either enlarge (for $s > 1$) or shrink (for $0 < s < 1$) the object. Non-proportional scaling distorts an object using different scaling factors

s_x, s_y and s_z for each axis, allowing enlargement or shrinkage in only one direction. Initially, non-proportional scaling is considered along the three main axes. The new coordinates are computed as:

$$\begin{cases} x' = s_x \cdot x \\ y' = s_y \cdot y \\ z' = s_z \cdot z \end{cases}$$

The scaling transformation matrix $S(s_x, s_y, s_z)$ is formed by placing the scaling factors s_x, s_y and s_z on the diagonal:

$$M = S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It's worth noting that proportional scaling is a special case achieved when using identical scaling factors $s = s_x = s_y = s_z$.
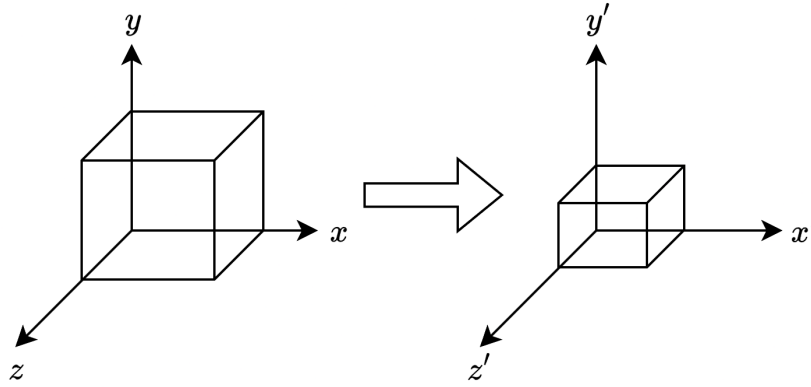
Figure 2.4: Scaling transform

Mirroring   Mirroring can be achieved by utilizing negative scaling factors. Initially, we assume the mirror occurs around a plane or axis passing through the origin, aligned with the x, y or z axes. Three types of mirroring are possible:

- *Planar*: this creates a symmetric object with respect to a plane. It's accomplished by setting -1 as the scaling factor for the axis perpendicular to the plane (x for yz-plane):

$$\begin{cases} s_x = -1 \\ s_y = 1 \\ s_z = 1 \end{cases}$$

- *Axial*: this creates a symmetric object with respect to an axis. It's achieved by setting -1 as the scaling factor for all axes except the one corresponding to the axis of symmetry (x and z for the y-axis):

$$\begin{cases} s_x = -1 \\ s_y = 1 \\ s_z = -1 \end{cases}$$

- *Central*: this creates a symmetric object with respect to the origin. It's accomplished by setting -1 as the scaling factor for all axes:

$$\begin{cases} s_x = -1 \\ s_y = -1 \\ s_z = -1 \end{cases}$$

Flattening When a scaling factor of 0 is applied in any direction, it flattens the image along that axis. However, this operation must be approached with caution as it effectively reduces the dimensionality of the objects. For the sake of simplicity in our discussion, we will typically assume that the scaling coefficients are non-zero.

2.2.4 Rotation

Rotation alters the orientation of an object while keeping its position and size unchanged. It is performed along an axis, a line where points remain unaffected by the transformation. Initially, we will focus on rotations about the three reference axes passing through the origin.

A rotation of an angle α about the z-axis can be computed as follows:

$$\begin{cases} x' = x \cdot \cos \alpha - y \cdot \sin \alpha \\ y' = x \cdot \sin \alpha - y \cdot \cos \alpha \\ z' = z \end{cases}$$

Expressed in matrix form, this becomes:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Utilizing homogeneous coordinates, rotations of an angle α around the z-axis can be represented by matrices. Rotations about the x-axis and the y-axis follow similar principles and are expressed by the following matrices:

$$R_y(\alpha) = \begin{bmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_z(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
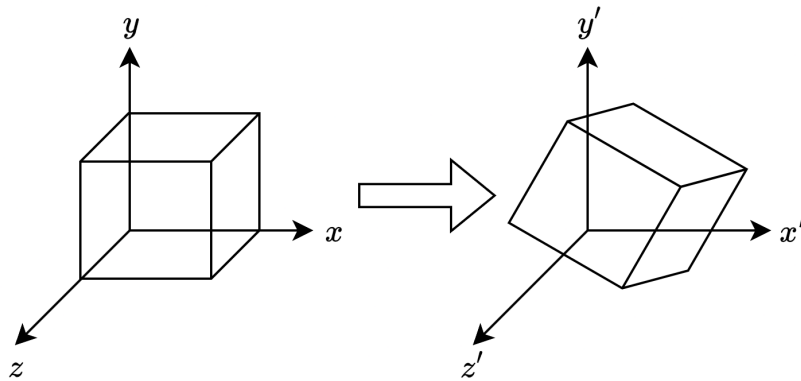
Figure 2.5: Rotation transform

2.2.5 Shear

The shear transform bends an object in one direction and is performed along an axis with a center. Initially, we focus on the y-axis passing through the origin. As the value of the y-axis increases, the object is linearly bent into a direction specified by a 2D vector (defined by h_x and h_z). The transformed point coordinates are computed as follows:

$$\begin{cases} x' = x + y \cdot h_x \\ y' = y \\ z' = z + y \cdot h_z \end{cases}$$

In matrix form, this becomes:

$$H_x(h_y, h_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ h_y & 1 & 0 & 0 \\ h_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Similarly, shear transforms can be applied along the x-axis and the z-axis:

$$H_y(h_x, h_z) = \begin{bmatrix} 1 & h_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad H_z(h_x, h_y) = \begin{bmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
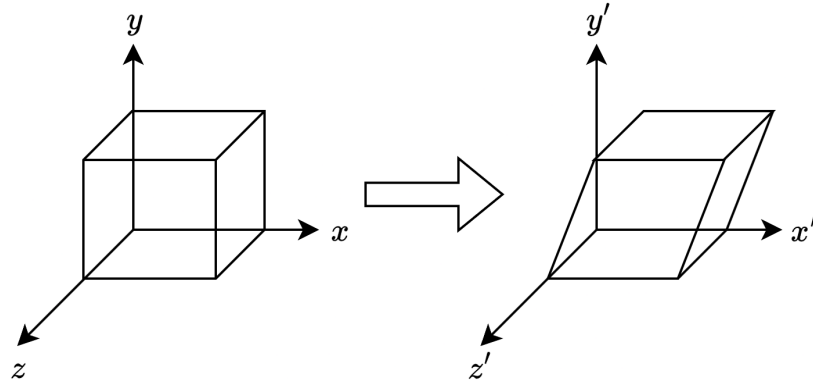
Figure 2.6: Shear transform

2.2.6 Transformation matrix

It's important to note that in all the 4×4 transformation matrices we've discussed, the last row is always $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$. This ensures that the w coordinate remains unchanged by the transformation.

The upper part of a transformation matrix can be split into a 3×3 sub-matrix M_R, representing the rotation, scaling, and shear factors of the transform, and a column vector d^T encoding the translation:

$$M = \begin{bmatrix} n_{xx} & n_{yx} & n_{zx} & d_x \\ n_{xy} & n_{yy} & n_{zy} & d_y \\ n_{xz} & n_{yz} & n_{zz} & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} M_R & d^T \\ 0 & 1 \end{bmatrix}$$

Specifically, the matrix product exchanges the three Cartesian axes of the original coordinate system with three new directions. The columns of M_R represent the directions and sizes of the new axes in the old reference system.
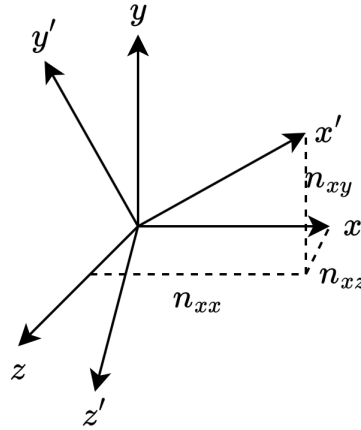
Figure 2.7: Transformation matrix M_R

The vector d^T represents the position of the origin of the new coordinate system in the old one.
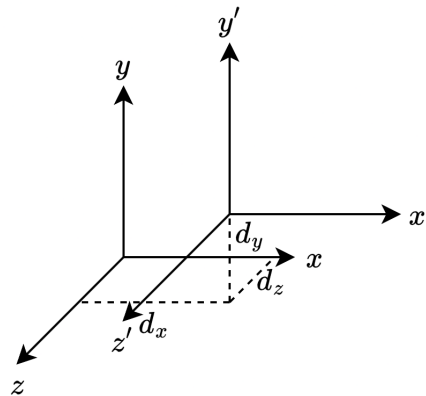
Figure 2.8: Transformation matrix d_t

The transformation introduces the following changes to the axes:

- Rotations maintain the size and angles of the axes constant but change their directions.

- Scaling increases or decreases the size of the axes while maintaining their original directions.

- Shear bends the axes along which the transform is performed.

In many cases, it's simpler to define a transformation by specifying its new center and the new directions of its axes.

Conventions   It's important to highlight that under the matrix-on-the-right convention, all transformation matrices are transposed. A simple way to identify which convention is being used is by inspecting a non-zero translation transform:

- If the matrix has the last column $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$, then the matrix-on-the-right convention is employed.

- Conversely, if the last row is $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$, then the matrix-on-the-left convention is being utilized.

2.3   Transformation inverse

Transformations can be reversed to return an object to its original position, size, or orientation. One advantage of using matrices is that the inverse transformation can be easily computed by inverting the corresponding matrix: if point p' is the result of applying the transformation encoded in matrix M to a point p, then point p can be retrieved from p' by multiplying it with M^{-1}, the inverse of matrix M:

$$p = M^{-1} \cdot p'$$

It can be shown that a transformation matrix M is invertible if its sub-matrix composed of the first 3 rows and 3 columns is invertible. This is generally true, except in cases where:

- One or two of the projected axes degenerate to zero length.

- Two axes perfectly overlap.

- One axis aligns with the plane defined by the other two.

The inverse of a general matrix M can be computed as:

$$M^{-1} = \frac{1}{\det(M)} \operatorname{adj}(M)$$

Here, $\operatorname{adj}(M)$ denotes the adjugate of a square matrix M, which is the transpose of its cofactor matrix.

However, for some transformations presented earlier, their inverses can be computed using simple matrix patterns:

- For translation:

$$M = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- For scaling:

$$M = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- For rotation, by changing the sign of the sine function:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.4 Transformation composition

When constructing a scene, an object undergoes multiple transformations, and combining these transformations in a sequence is termed composition. This typically involves translating and rotating the object in various directions to accurately position it within the scene. Additionally, achieving rotations around arbitrary axes and scaling with different centers entails combining diverse transformations. The efficient application of transformation composition is facilitated by the properties of matrix multiplication.

To apply composition, we begin by placing the object's center at position (p_x, p_y, p_z) and orienting its direction at an angle α around the y-axis. Following this initial step, each transformation is performed in order. In the case of rotation and translation, it's essential to perform the translation first to avoid complications.

It's noteworthy that, akin to functional programming, matrices appear inside the expression in reverse order concerning the transformations they represent.

2.4.1 Transformations around an arbitrary axis

Now, let's consider the rotation of an object by an angle α about an arbitrary axis passing through the origin.

Suppose we're focusing on a scenario where the direction of the arbitrary axis can be defined using a pair of angles. Specifically, we can align the x-axis with the arbitrary axis by first rotating by an angle γ around the z-axis, followed by an angle β around the y-axis.
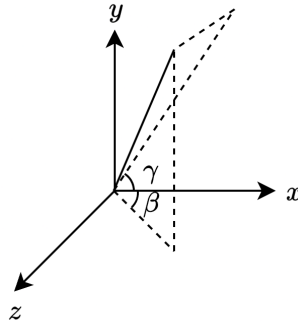
Figure 2.9: Rotation with an arbitrary axis

To position the axis correctly, we need to perform the following transformations: R_y^{-1}, R_z^{-1}, R_x, R_z, and R_y.

If the axis doesn't pass through the origin but through a point (p_x, p_y, p_z), we must apply a translation $T^{-1}(p_x, p_y, p_z)$ to move it to the origin and then use $T(p_x, p_y, p_z)$ at the end to return the axis to its initial position:

$$R_y R_z R_x R_z^{-1} R_y^{-1} T^{-1}$$

In summary, a rotation of α about an arbitrary axis passing through the point (p_x, p_y, p_z), aligning the x-axis by rotating γ around the z-axis and β around the y-axis, can be computed as:

$$p' = T(p_x, p_y, p_z) R_y(\beta) R_z(\gamma) R_x(\alpha) R_z(\gamma)^{-1} R_y(\beta)^{-1} T(p_x, p_y, p_z)^{-1} p$$

Similar procedures can be followed for different rotation sequences to align another main axis with the arbitrary one.

These considerations also apply to scaling an object along arbitrary directions and with an arbitrary center. They can be extended to generalize shear and perform symmetries about arbitrary planes, axes, or centers.

In many cases, the rotation axis can be represented by a unit vector $n = (n_x, n_y, n_z)$ where $n_x^2 + n_y^2 + n_z^2 = 1$. In this scenario, the rotation matrix can be determined using the following pattern:

$$\begin{bmatrix} \cos\alpha + n_x^2\left(1 - \cos\alpha\right) & n_x n_y\left(1 - \cos\alpha\right) - n_z\sin\alpha & n_x n_z\left(1 - \cos\alpha\right) + n_y\sin\alpha & 0 \\ n_x n_y\left(1 - \cos\alpha\right) + n_z\sin\alpha & \cos\alpha + n_y^2\left(1 - \cos\alpha\right) & n_y n_z\left(1 - \cos\alpha\right) - n_x\sin\alpha & 0 \\ n_x n_z\left(1 - \cos\alpha\right) - n_y\sin\alpha & n_y n_z\left(1 - \cos\alpha\right) + n_x\sin\alpha & \cos\alpha + n_z^2\left(1 - \cos\alpha\right) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

CHAPTER **3**

Projections

3.1 Introduction

In the realm of 3D computer graphics, the objective is to portray a three-dimensional environment on a flat screen, which inherently possesses only two dimensions. Even in the realm of stereoscopic imagery, the illusion of depth arises from the brain's processing of distinct 2D images presented to each eye. Utilizing geometrical primitives, 3D computer graphics encapsulates objects within a three-dimensional realm, subsequently rendering a 2D depiction of this space for display on the screen.

The process of generating a 2D image from a 3D scene is known as projection. This technique defines the 2D depiction of a 3D object by the intersection of a set of projection rays with a surface.

Typically, this surface is a plane known as the projection plane. On this plane, a rectangular area corresponds to the screen. An essential characteristic of planar projections is that linear segments in the 3D scene maintain their straightness when projected onto the screen. It's important to note that this characteristic no longer holds true when projections are made onto surfaces other than a plane.

Because the projected segments link the projections of their endpoints, it's enough to project the vertices and connect them to reconstruct a 2D representation of the corresponding 3D objects.

We will consider two types of planar projections:

- *Parallel projection*: all rays are parallel to the same direction.

- *Perspective projection*: all the rays pass through a point, called the center of projection.

(a) Parallel   (b) Perspective

Figure 3.1: Types of projections

In both parallel and perspective projections, a point on the screen corresponds to an infinite number of coordinates in space. This results directly from losing a component when transitioning from a 3D spatial system to a 2D screen system.

Properties   In parallel projections, all points that lie along a line parallel to the projection ray are projected onto the same pixel. In perspective projection, all points that align with both the projected pixel and the center of projection are mapped to the same location.

From the world to the screen   In 3D computer applications, projections are realized by converting 3D coordinates between two reference systems. Specifically, projections transform world coordinates into 3D normalized screen coordinates.

The coordinate system that describes objects in 3D space is known as World Coordinates (or global coordinates).

World Coordinates represent a right-handed Cartesian coordinate system, where the origin is mapped to the center of the screen. In the y-up version considered in this course, the x and y-axes align with the horizontal and vertical edges of the screen, respectively. The z-axis extends out of the screen, toward the viewer.

It's worth noting that certain applications, such as Blender, employ a different convention for World Coordinates known as z-up. In this case, the z-axis aligns with the vertical screen direction, and the y-axis points inside the screen. Some applications may even use a left-handed system, where the y-axis extends toward the viewer.

(a) y-up   (b) z-up

Figure 3.2: World coordinates

As previously discussed, Normalized Screen Coordinates provide a means to specify point positions on a screen or window in a device-independent manner.

Despite the screen being a 2D surface, pixels originating from 3D images require a distance from the viewer to correctly sort surfaces and avoid generating unrealistic images.

In 3D Normalized Screen Coordinates, a third component is introduced, which, in the context of Vulkan, falls within the $[0, 1]$ range. Points with smaller z-values are considered closer to the viewer. It's important to note that in this context, all normalized screen coordinates are considered to be 3D.

It's worth mentioning that other low-level graphics engines, such as OpenGL, employ completely different systems for normalized screen coordinates.

3.2 Parallel projections

Parallel projections are a type of graphical projection where all projection lines are parallel to each other, resulting in objects appearing the same size regardless of their distance from the observer. In parallel projections, the projection plane is positioned perpendicular to the line of sight, and there is no foreshortening or perspective distortion.

3.2.1 Orthogonal projections

Orthogonal projections are characterized by projecting onto planes that align with the xy, yz, or zx planes, with the projection rays perpendicular to these planes. They find primary use in technical drawings, as segments of equal length parallel to an axis retain their identical properties.

It's important to recognize that each plane has two distinct projection directions, resulting in differing orderings of the z-component of normalized screen coordinates. This can sometimes produce entirely different images.

We'll initially focus on a projection plane corresponding to the xy-plane, which is perpendicular to the z-axis. In this setup, visible objects possess negative values in their z coordinate.

The screen boundaries naturally constrain the portion of the 3D world visible on a screen along the horizontal and vertical axes. However, it's also necessary to limit the range of a scene along the depth axis for several reasons:

1. Prevent displaying objects behind the observer.

2. Avoid showing objects that are excessively distant from the viewer's perspective.

3. Enable the z-axis of the normalized screen coordinates to be confined within the 0 to +1 range.

These constraints define two planes:

- The plane with the closest z-component (maximum signed value, minimum absolute value) is termed the near plane.

- The one with the farthest z-component (minimum signed value, maximum absolute value) is referred to as the far plane.

After projection, only points contained within the box bounded by the left, right, top, and bottom screen borders, as well as the near and far planes along the depth axis, can be visualized. Normalized Screen Coordinates are defined such that the near plane, the left side, and the bottom side of the screen are projected to $z_n = 0$, $x_l = -1$, and $y_t = -1$, respectively. Conversely, the far plane, the right side, and the top side are projected to $z_f = 1$, $x_r = 1$, and $y_b = 1$, respectively.

Orthogonal projection matrix Orthogonal projections can be implemented by normalizing the x, y, z coordinates of the projection box within the ranges $(-1, 1)$, $(-1, 1)$, and $(0, 1)$ respectively. This normalization process, starting from a valid coordinate, can be achieved by multiplying with a suitable matrix. In particular, we define the projection matrix, P_{ort}, as the matrix that, when multiplied with a world coordinate p_W, computes the corresponding normalized screen coordinate p_N:

$$P_N = P_{ort} \cdot P_W$$

To obtain normalized coordinates, we must determine the coordinates of the screen borders in 3D space. We denote l and r as the x-coordinates in 3D space corresponding to locations displayed on the left and right borders of the screen. Anything to the left of l will be excluded from the screen, and nothing to the right of r will be visualized. Similarly, we use t and b for the y-coordinates of the top and bottom borders of the screen in 3D space. P_{ort} We also denote $-n$ and $-f$ as the z-coordinates of the near and far planes in 3D space. Since the z-axis is oriented opposite to the viewer's perspective, positive distances n and f are generally preferred over negative coordinates where the two planes intersect.

The orthogonal projection matrix P_{ort} is computed in three steps. First, we translate the center of the projection box to the near plane at the origin using a translation T_{ort}:

$$T_{ort} = \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -(-n) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The second step involves normalizing the coordinates between -1 and 1 (for the x-axis and y-axis) or 0 and 1 (for the z-axis) through a scale transformation S_{ort}:

$$S_{ort} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The last step corrects the direction of the axes with the mirror matrix M_{ort}, by inverting the y-axis and z-axis:

$$M_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By applying the composition of transformations, we can define the orthogonal projection matrix P_{ort} as follows:

$$M_{ort} = M_{ort} \cdot S_{ort} \cdot T_{ort} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{l-r} \\ 0 & \frac{2}{b-t} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.2.2 Aspect ratio

The ratio between the horizontal and vertical dimensions of the physical screen is known as the aspect ratio:

$$a = \frac{D_x}{D_y}$$

In this definition, the measures for the horizontal and vertical sizes, D_x and D_y, are specified in metric units rather than pixels. When pixels are square, both metric units and pixels yield the same outcome. However, if pixels are not square, only the actual display proportions can ensure images are not distorted.

Normalized screen coordinates do not incorporate the aspect ratio. Instead, the projection matrix adjusts for this factor, appropriately scaling the images to ensure they appear with the correct proportions.

To generate images with accurate proportions, the values of l, r, t, and b must be consistent with the aspect ratio a of the monitor:

$$a = \frac{r - l}{t - b} \rightarrow r - l = a \cdot (t - b)$$

In many cases, the projection box is centered at the origin both horizontally and vertically. When this occurs, only the half-width of the box w, the near plane n, the far plane f, and the aspect ratio a are required.

$$\begin{cases} l = -w \\ r = w \\ t = \frac{w}{a} \\ b = -\frac{w}{a} \end{cases}$$

As a result the transformation matrix becomes:

$$P_{ort} = \begin{bmatrix} \frac{1}{w} & 0 & 0 & 0 \\ 0 & -\frac{a}{w} & 0 & 0 \\ 0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For orthogonal projections where the near plane is positioned at $n = 0$, the projection matrix can be further simplified into a non-uniform scaling matrix with factors $S(\frac{1}{w}, -\frac{a}{w}, -\frac{1}{f})$:

$$P_{ort} = \begin{bmatrix} \frac{1}{w} & 0 & 0 & 0 \\ 0 & -\frac{a}{w} & 0 & 0 \\ 0 & 0 & -\frac{1}{f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

While World Coordinates are represented using Homogeneous coordinates, Normalized Screen Coordinates are Cartesian coordinates. Therefore, a conversion procedure from one system to the other must be performed (division of the first three components by the fourth).

For parallel projection, since it's implemented using a sequence of conventional transform matrices, the last element of P_N is always one. This implies that Normalized Screen Coordinates can be obtained by simply discarding the fourth element of the vector resulting from the product of the homogeneous coordinates with the projection matrix.

Orthogonal projections in GLM GLM gives the `ortho()` function to compute the orthographic projection matrix specifying the boundaries:

```
glm::mat4 Port = glm::ortho(l, r, b, t, n, f);
```

Here, l, r, b, t, n, f are the positions in world coordinates respectively of the left, right, bottom, top, near and far boundaries of the visible region. Keep however in mind that such procedure was created for the Normalized Screen Coordinates conventions of OpenGL, and the result will not work correctly in Vulkan without further modifications. In order to make it work, the following additions must be done:

```
#define GLM_FORCE_DEPTH_ZERO_TO_ONE

Port = glm::scale(glm::mat4(1.0f), glm::vec3(1,-1,1)) *
                  glm::ortho(l, r, b, t, n, f);
```

3.2.3   Axonometric projections

Many objects conform to shapes that align with the sides of a box. In parallel projections, faces perpendicular to the projection plane are concealed. When a cube is aligned with the three main axes, it presents as a square, restricting depth perception. Axonometric projections resolve this constraint by simultaneously displaying all cube faces. Axonometric projections enable the viewing of all faces of box-shaped objects by either rotating the projection plane relative to the three main axes or by adjusting the angle of the projection plane so that it is no longer perpendicular to the projection rays.

Orthographic projections   Orthographic projections are a type of axonometric projection where rays are perpendicular to the projection plane. The principal orthographic axonometric projections include:

- *Isometric*: the three axes are oriented at angles of 120° from one another. Segments of equal lengths aligned parallel to the main axis in three-dimensional space maintain equal lengths when projected onto a two-dimensional plane. Isometric projections are achieved by first rotating $\pm 45°$ around the y-axis, followed by a rotation of $\pm 35.26°$ around the x-axis, before applying the parallel projection as previously described. It's worth noting that in this scenario, the projection matrix needs to be specified with the half-width and the aspect ratio, as the box's borders displayed on the screen are no longer aligned with the main axis.

$$
\begin{bmatrix} \frac{1}{w} & 0 & 0 & 0 \\ 0 & -\frac{a}{w} & 0 & 0 \\ 0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(35.26°) & \sin(35.26°) & 0 \\ 0 & -\sin(35.26°) & \cos(35.26°) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} \cos(45°) & 0 & \sin(45°) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(45°) & 0 & \cos(45°) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

  The rotations around the x and y axes, with their respective signs, yield four distinct yet equally valid axonometric projections.

- *Dimetric*: in dimetric projection, distinct units are used for the x and z-axes compared to the y-axis. Its popularity in the 1980s stemmed from its simplicity in implementation through integer arithmetic. Even today, dimetric projection remains prevalent in retro-style games and applications. To obtain dimetric projections, a rotation of $\pm 45°$ around the y-axis is first applied, followed by an arbitrary rotation α around the x-axis, before

executing the basic parallel projection:

$$
\begin{bmatrix}
\frac{1}{w} & 0 & 0 & 0 \\
0 & -\frac{a}{w} & 0 & 0 \\
0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\alpha) & \sin(\alpha) & 0 \\
0 & -\sin(\alpha) & \cos(\alpha) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos(45°) & 0 & \sin(45°) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(45°) & 0 & \cos(45°) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

- *Trimetric*: in trimetric projection, each axis has a different unit. To obtain trimetric projections, an arbitrary rotation β around the y-axis is first applied, followed by an arbitrary rotation α around the x-axis, before executing the parallel projection:

$$
\begin{bmatrix}
\frac{1}{w} & 0 & 0 & 0 \\
0 & -\frac{a}{w} & 0 & 0 \\
0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\alpha) & \sin(\alpha) & 0 \\
0 & -\sin(\alpha) & \cos(\alpha) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos(\beta) & 0 & \sin(\beta) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\beta) & 0 & \cos(\beta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

(a) Isometric (b) Dimetric (c) Trimetric
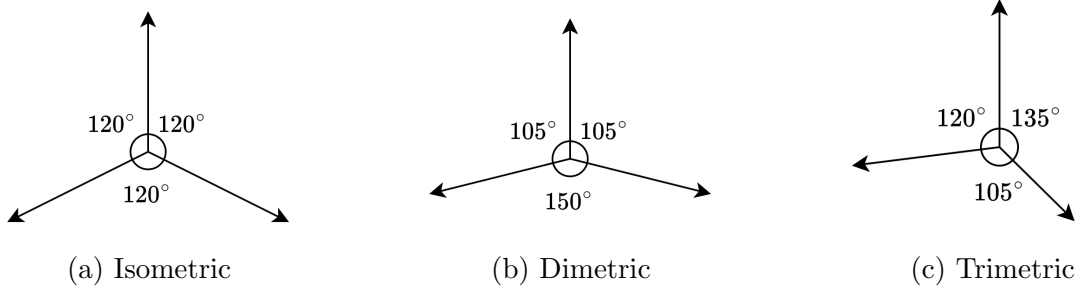
Figure 3.3: Orthographic projections

A fundamental characteristic of axonometric projections is that they preserve the upward axis (the y-axis in our context) parallel to the vertical orientation of the screen.

3.2.4 Oblique projections

In oblique projections, rays are parallel but oblique in relation to the projection plane. Consequently, two of the three axes (x and y) run parallel to the screen, while the third axis (z) is inclined at an angle to the other two. The z-axis is commonly angled at 45°, 30°, or 60°, and it can be oriented in either direction. The length of the z-axis can either match that of the other two axes or be halved. If the length is preserved, the projection is termed Cavalier; otherwise, it's referred to as Cabinet.

Oblique projections, valued for their simplicity in implementation using only integer arithmetic, found utility in some arcade and PC games.

These projections are achieved by applying a shear along the z-axis before the orthogonal projection:

$$
\begin{bmatrix}
\frac{1}{w} & 0 & 0 & 0 \\
0 & -\frac{a}{w} & 0 & 0 \\
0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -\rho\cos(\alpha) & 0 \\
0 & 1 & -\rho\sin(\alpha) & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

The shear factor determines the projection angle and whether it will be Cavalier or Cabinet. Specifically, it can be defined with respect to the axis angle, denoted as α, and the corresponding reduction factor, denoted as ρ:

- Cavalier: $\alpha = 45°$, $\rho = 1$.

- Cabinet: $\alpha = 45°$, $\rho = 0.5$.

3.3 Perspective projections

Parallel projections maintain the apparent size of an object regardless of its distance from the observer, making them commonly employed in technical drawings. On the other hand, perspective projections depict objects with varying sizes based on their distance from the projection plane, making them better suited for immersive visualizations.

The magnification effect occurs because all projection rays converge at a single point. For simplicity, let's consider a side view where the projection plane is reduced to a line.

Rays intersect the projection plane at varying points based on the object's distance. Comparing segments from objects of equal size, but different distances reveals that closer objects to the plane exhibit larger projections.

A point with coordinates (x, y, z) in space is projected onto the plane, resulting in a point with Normalized Screen Coordinates (x_s, y_s, z_s), where z_s is necessary to order points based on their distance from the viewer, as in the case of parallel projections. Initially, we'll concentrate on y_s, followed by x_s, and finally on z_s. Specifically, when focusing on the y-axis, we can calculate the normalized screen coordinate y_s as the projection of the vertical component of the point (x, y, z) onto the plane.
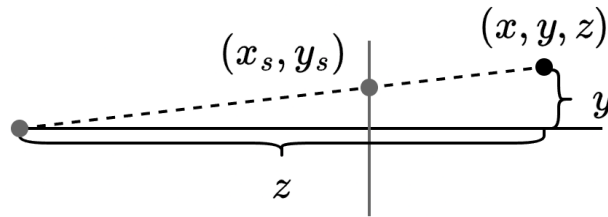
Figure 3.4: Perspective projection with respect to y-axis

For ease of computation, we place the center of projection at the origin $(0, 0, 0)$. The projection plane is positioned at a distance d along the z-axis from the projection center. Examining the image below, we notice two similar triangles: ABC and ADE. Here, y_s represents the height of the smaller triangle, while the world coordinate y represents the height of the larger one.
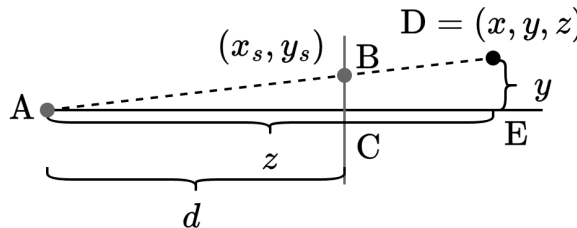
Figure 3.5: Perspective projection with respect to y-axis with triangles

Due to the similarity of the two triangles, there exists a linear relationship among the lengths of their edges. Specifically, we have:

$$y_s = \frac{d \cdot y}{z}$$

The same reasoning applies to x_s as well:

$$x_s = \frac{d \cdot x}{z}$$

Focal length Parameter d signifies the distance between the center of projection and the projection plane. It can be utilized to simulate the focal length of a camera lens. Notably, adjusting d results in a zoom effect.

A short d corresponds to a wide-lens effect: it accentuates the distances of objects from the plane. Additionally, it enables the capture of a larger number of objects in the view, resulting in smaller objects in the images.

A large d corresponds to a tele-lens effect, minimizing the differences in size for objects at various distances. It also decreases the number of objects visible in the scene, resulting in enlarged views.

Parallel (orthographic) projections can be derived from perspective projections as d approaches infinity.

Perspective projections matrices With the aid of homogeneous coordinates, perspective projections can also be achieved through a matrix-vector product. It's important to recognize that the world coordinate system under consideration is oriented oppositely along the z-axis, meaning the z-coordinates are negative:

$$x_s = \frac{d \cdot x}{-z} \qquad y_s = \frac{d \cdot y}{-z}$$

The formula under consideration requires a sign change to accommodate the actual direction of the z-axis. In Vulkan, the y-axis should also be mirrored, but it will be easier to perform this in a separate final step. The projection matrix for perspective, with center in the origin and projection plane at distance d on the z-axis, can be defined as:

$$P_{persp} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note that the last row of this transform matrix is no longer equal to $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$. This also makes the product of matrix P_{persp} with a homogeneous coordinate resulting in a vector with component $w \neq 1$.

The previous method comes with a drawback: the z component of the resulting Cartesian coordinate is consistently $-d$, resulting in the loss of information regarding the distance from the projection plane. Consequently, it is not possible to define accurate 3D Normalized Screen Coordinates with a z_s component that accurately represents the point's distance from the view plane. Since z_s is only utilized for sorting primitives, it suffices to insert an element equal to 1 in the third row of the fourth column of the matrix to achieve a z_s component that consistently

varies with the distance of the point:

$$
P_{persp} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}
$$

Similar to parallel projections, the visible area of the 3D world corresponds to normalized screen coordinates within the range $[-1, +1]$ for x and y, and $[0, 1]$ for z. Additional transforms are then applied to ensure this range corresponds to the specific area of the world we aim to display. Let n represent the distance from the origin to the near plane. We denote l, r, t, and b as the coordinates of the left, right, top, and bottom edges of the projection plane in world space at the near plane. Lastly, f denotes the distance from the origin to the far plane. It's important to note that, because of perspective, the coordinates of the screen borders at the far plane differ from those at the near plane. Consequently, the visible area is not a box but rather a frustum. Additionally, it's worth noting that the coordinates of the screen borders don't necessarily need to be symmetric. This flexibility can be utilized to calculate shifted viewports, enabling the sharing of a projection over two adjacent screens, for instance. Given that the coordinates of the screen border are defined at the near plane, the value of n corresponds to the distance of the projection plane, denoted as d. It's important to emphasize that for perspective projections, n must be greater than 0. The initial step involves writing the previously introduced projection matrix while substituting d with n:

$$
U_{persp} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}
$$

To derive Normalized Screen Coordinates, additional transformations need to be sequentially applied. Starting with the given values of l, r, t, and b at the near plane, we first calculate the projections of the top-left and bottom-right corners at the near plane.

$$
\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} l \\ t \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} nl \\ nt \\ -n^2 + 1 \\ n \end{bmatrix} = \begin{bmatrix} l \\ t \\ -n + \frac{1}{n} \\ 1 \end{bmatrix}
$$

$$
\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} r \\ b \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} nr \\ nb \\ -n^2 + 1 \\ n \end{bmatrix} = \begin{bmatrix} r \\ b \\ -n + \frac{1}{n} \\ 1 \end{bmatrix}
$$

Additionally, we need to calculate the projected coordinates of a point at the far plane ($z = -f$) to establish the appropriate normalization for the z-axis.

$$
\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ -n^f + 1 \\ f \end{bmatrix} = \begin{bmatrix} \frac{n \cdot x}{f} \\ \frac{n \cdot y}{f} \\ \frac{1}{f} - 1 \\ 1 \end{bmatrix}
$$

We have also that:

$$T_{persp} = \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\left(\frac{1}{n} - n\right) \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad S_{persp} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{nf}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combining all the elements, we obtain;

$$P_{persp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{b-t} & \frac{t+b}{b-t} & 0 \\ 0 & 0 & \frac{f}{n-f} & \frac{nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

As for parallel projection, the values l, r, t and b must be consistent with the aspect ratio a of the monitor.

$$l - r = a \cdot (t - b)$$

Camera In many scenarios, a set of parameters resembling those found in a camera is employed. These typically include the distances n and f of the near and far planes, the angle Θ at the top of the frustum (referred to as the field of view or fov_y) along the y-axis, and the aspect ratio a of the screen. From the previous definitions we have:

$$t = n \tan \frac{\Theta}{2} \qquad b = -n \tan \frac{\Theta}{2} \qquad l = -a \cdot n \tan \frac{\Theta}{2} \qquad r = a \cdot n \tan \frac{\Theta}{2}$$

Plugging the values l, r, t and b in the P_{persp} matrix we obtain:

$$P_{persp} = \begin{bmatrix} \frac{1}{a \cdot \frac{\Theta}{2}} & 0 & 0 & 0 \\ 0 & -\frac{1}{\frac{\Theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{f}{n-f} & \frac{nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective projections in GLM GLM provides the `frustum()` function for computing the perspective projection matrix, defining the boundaries of the visible region:

```
glm::mat4 Port = glm::frustum(l, r, b, t, n, f);
```

Here, l, r, b, t, n, f represent the left, right, bottom, top, near, and far boundaries in world coordinates, respectively.

The `perspective()` function computes the perspective projection matrix based on the field of view (fov) and aspect ratio (a):

```
glm::mat4 Port = glm::perspective(fov, a, n, f);
```

Where fov, a, n, and f denote the vertical field of view, aspect ratio, near plane distance, and far plane distance, respectively.

As for the `ortho()` function, originally designed for OpenGL, adjustments are necessary for Vulkan compatibility. To achieve this, the following steps are required:

```
#define GLM_FORCE_DEPTH_ZERO_TO_ONE

M2glm = glm::scale(glm::mat4(1.0), glm::vec3(1,-1,1)) *
        glm::frustum(l, r, b, t, n, f);
M1glm = glm::perspective(fovy, a, n, f);
M1glm[1][1] *= -1;
```

3.4 World coordinates

To simplify positional and directional references, we'll establish the origin at the center of the 3D world and utilize a compass for directional definitions. Specifically, we'll designate the negative z-axis as North and the positive x-axis as east.

The projection matrices we've encountered assume a projection plane parallel to the xy-plane. For parallel projections, we assume the projection rays align with the negative z-axis. In perspective, the projection center is at the origin, akin to a camera facing north.

In practical applications, generating a 2D view for a plane positioned anywhere in space involves additional transformations before projection.

Though our focus is on perspective, the same principles apply to parallel projection. The projection plane resembles a camera sensor viewing the scene from the projection center. This camera is defined by its position, aiming direction, and angle around this direction. The projection matrices we've discussed compute the view from a camera initially at the origin, facing the negative z-axis. This camera can be conceptualized as a 3D object.

We can then define a transformation matrix M_C that relocates this camera object to its target position, aligning the negative z-axis with the desired direction. This matrix is termed the camera matrix. By applying the inverse of M_C to all scene objects, we transform them into a new 3D space where the projection plane aligns as needed.

In this transformed space, the view from the arbitrarily oriented camera can be computed by applying the projection matrices.

The inverse of matrix M_C, denoted as M_V, is known as the view matrix:

$$M_V = M_C^{-1}$$

The view matrix precedes the projection matrix M_{prj}, enabling the determination of normalized screen coordinates of points in space as observed by the specified camera. This combined matrix is often referred to as the view-projection matrix:

$$M_{VP} = M_{prj} \cdot M_V$$

It transforms a point from world coordinates (in homogeneous coordinates) to 3D normalized screen coordinates (in Cartesian coordinates) at a specific location and direction in space according to matrix M_V. Various techniques exist for creating a view matrix in a user-friendly manner, with the two most popular being:

- *Look-in-direction* technique.

- *Look-at* technique.

3.4.1 Look-in-direction

This technique is commonly employed in first-person applications where the user directly controls the camera's position and view direction. In the look-in-direction model, the camera's position (c_x, c_y, c_z) is given in world coordinates. The direction where the camera is facing is specified using three angles: the compass direction (angle α), the elevation (angle β), and the roll over the viewing direction (angle ρ). The roll parameter is seldom used.

Specifically, with $\alpha = 0°$, the camera faces north, while $\alpha = 90°$ implies the camera is facing west. South corresponds to $\alpha = 180°$, and East is $\alpha = 270°$. A positive angle $\beta > 0°$ makes the camera look up, while a positive angle $\rho > 0°$ rotates the camera counterclockwise. In

terms of the camera object, roll corresponds to rotation around the z-axis, elevation (or pitch) along the x-axis, and direction (or yaw) around the y-axis. Rotations must follow a specific order, discussed later: roll first, then elevation, and finally direction. Translation is applied after rotations. The camera matrix is composed as follows:

$$M_C = T(c_x, c_y, c_z) \cdot R_y(\alpha) \cdot R_x(\beta) \cdot R_z(\rho)$$

The view matrix, being the inverse of the camera matrix, follows the rules for inverting a chain of transformations:

$$M_V = M_C^{-1} = R_z(-\rho) \cdot R_x(-\beta) \cdot R_y(-\alpha) \cdot T(-c_x, -c_y, -c_z)$$

Look-in in GLM GLM does not offer specific support for constructing a look-in-direction matrix. However, due to its simplicity, it can be easily implemented as follows:

```
glm::mat4 Mv = glm::rotate(glm::mat4(1.0), -rho, glm::vec3(0,0,1)) *
               glm::rotate(glm::mat4(1.0), -beta, glm::vec3(1,0,0)) *
               glm::rotate(glm::mat4(1.0), -alpha, glm::vec3(0,1,0)) *
               glm::translate(glm::mat4(1.0), glm::vec3(-cx, -cy, -cz));
```

3.4.2 Look-at

The look-at model is typically utilized in third-person applications, where the camera tracks a specific point or object. In this scenario, the camera's position (c_x, c_y, c_z) is given in world coordinates. This technique also requires the definition of an up vector, denoted as $u = (u_x, u_y, u_z)$, perpendicular to the ground plane of the scene. In y-up coordinate systems, the up vector is commonly set to $u = (0, 1, 0)$, aligning the camera's y-axis perpendicular to the horizon.

The view matrix is computed by first determining the direction of its axes in world coordinates, and then constructing the camera matrix accordingly. Initially, we establish the transformed (negative) z-axis as the normalized vector originating from the point the camera is looking at and ending at the camera center:

$$v_z = \frac{c - a}{|c - a|}$$

where a represents the point the camera is aimed at.

The new x-axis must be perpendicular to both the new z-axis and the up vector. This can be calculated using the normalized cross product of the two vectors:

$$v_x = \frac{u \cdot v_z}{|u \cdot v_z|}$$

It's important to note that the cross product yields zero if the vectors u and v_z are collinear, making it impossible to determine v_x. This situation arises when the viewer is perfectly vertical, hindering the alignment of the camera with the ground. In such cases, the simplest solution is to use the previously computed matrix or select a random orientation for the x-axis.

Finally, the new y-axis should be perpendicular to both the new z-axis and the new x-axis, and it can be computed via the cross product of the two previously obtained vectors:

$$v_y = v_z \cdot v_x$$

The camera matrix, denoted as M_C, is then constructed by arranging vectors v_x, v_y, and v_z in the first three columns, and the position of the camera center c in the fourth column:

$$M_C = \begin{bmatrix} v_x & v_y & v_z & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The view matrix can be obtained by inverting M_C. Since the vectors are orthogonal, the inversion of a look-at camera matrix can be computed easily with a transposition and a matrix-vector product:

$$M_V = M_C^{-1} = M_C = \begin{bmatrix} R_C^T & -R_C^T \cdot c \\ 0 & 1 \end{bmatrix}$$

Look-at in GLM GLM provides convenient functions for vector operations and matrix construction, simplifying the computation of view matrices.

To compute the cross product of two vectors, GLM offers the `cross()` function:

```
glm::vec3 uXvz = glm::cross(u, vz);
```

To normalize a vector, use the `normalize()` function:

```
glm::vec3 vx = glm::normalize(uXvz);
```

Constructing a 4×4 matrix from four column vectors can be achieved using the appropriate constructor:

```
glm::mat4 Mc = glm::mat4(vx, vy, vz, glm::vec4(c.x, c.y, c.z, 1));
```

This construction can be implemented as follows:

```
glm::vec3 vz = glm::normalize(c - a);
glm::vec3 vx = glm::normalize(glm::cross(u, vz));
glm::vec3 vy = glm::cross(vz, vx);
glm::mat4 Mc = glm::mat4(glm::vec4(vx), glm::vec4(vy),
                         glm::vec4(vz), glm::vec4(c, 1));
```

Here, c, a, and u are three `glm::vec3` vectors representing, respectively, the position of the camera, the target point, and the up vector. To obtain the view matrix, simply compute the inverse of the camera matrix:

```
glm::mat4 Mv = glm::inverse(Mc);
```

Additionally, GLM provides the `lookAt()` function to create a look-at matrix directly from three vectors representing the camera center, target point, and up vector:

```
glm::mat4 Mv = glm::lookAt(glm::vec3(cx, cy, cz), glm::vec3(ax, ay, az),
glm::vec3(ux, uy, uz));
```

Or using vectors directly:

```
glm::mat4 Mv = glm::lookAt(c, a, u);
```

Roll in GLM   Roll in look-at matrices can be implemented in two ways:

1. Rotating the u vector.

2. Adding a rotation of an angle ρ around the z-axis.

For the second method, it can be implemented as follows:

```
glm::mat4 Mv = glm::rotate(glm::mat4(1.0), -Roll, glm::vec3(0,0,1)) *
               glm::lookAt(c, a, u);
```

3.4.3   Local coordinates and world matrix

A fundamental aspect of 3D computer graphics is the capability to position and manipulate objects within the virtual environment. Object manipulation is typically achieved using a transformation matrix known as the World Matrix, denoted as M_W.

Each object possesses a set of local coordinates, representing the positions of its points within its creation space. When assembling a scene, these object points undergo a transformation from their original modeled positions to the positions where they are displayed. These transformed coordinates are referred to as the object's global or world coordinates.

The World Matrix M_W facilitates this transformation from local to global coordinates. It applies a sequence of translations, rotations (and possibly scaling or shears) to the object's local coordinates. As discussed previously, the order of transformations is crucial as matrix multiplication is not commutative. While there may not be a single solution, adhering to best practices generally yields the desired outcome.

3.4.4   World matrix definition

When working with objects described in local coordinates, users typically aim to:

- Position the object.

- Rotate the object.

- Scale or mirror the object.

Advanced transformations like shearing or scaling along arbitrary axes are generally avoided as they are not easily generalized. If such transformations are required, custom procedures must be developed to handle them.

Positioning   When positioning an object in 3D space, the user aims to define coordinates that remain consistent regardless of the object's size or orientation. Thus, specifying the placement at point $p = (p_x, p_y, p_z)$ necessitates applying translation $T(p_x, p_y, p_z)$ as the final transformation. This ensures that the object's location remains unchanged despite any prior rotations or scaling. The parameters $p = (p_x, p_y, p_z)$ of the translation $T(p_x, p_y, p_z)$ determine the position where the origin of the object, in its local space, will be located after it has been translated in the world space.

It's crucial that the object being positioned is modeled with its origin set at the point most accurately representing its location. This specific point varies for each object and should be carefully chosen during the modeling process.

If the object is to have dimensions s_x, s_y, s_z, rotations should be applied afterward. Otherwise, the scaling direction would be altered.

Therefore, scaling by $S(s_x, s_y, s_z)$ must be the initial transformation applied. When scaling by a factor s_x, it means that a unitary segment in local coordinates (along the x-axis) becomes s_x units in global coordinates. This principle similarly applies to scaling factors s_y and s_z.

Rotation   Rotation should be conducted between scaling and translation operations. In order to establish an orientation in 3D space, the fundamental rotations along the three axes, namely x, y, and z, need to be combined.

There are various methods to compute the rotation matrix for an object. One objective in constructing the rotation component of the world matrix is to streamline the specification of parameters needed by the application to define the direction of an object. Euler Angles, comprising three parameters that describe an object's orientation in 3D space, are commonly referred to as Roll, Pitch, and Yaw.

Euler's angles were initially introduced for z-up coordinate systems, making them easier to remember in that context. Therefore, we will initially explain them in a z-up coordinate system, where the World's North aligns with the positive y-axis. Objects should be modeled in a way that they face along the positive x-axis, their side is aligned with the y-axis, and their vertical line is oriented along the z-axis.

The Roll, denoted as ϕ, represents the rotation of the object along its facing axis (the x-axis). A positive roll turns the object clockwise.

The Pitch, denoted as θ, defines the elevation of the object and corresponds to a rotation around its y-axis (the side axis). A positive pitch turns the head of the object facing down.

The Yaw, denoted as ψ, defines the direction of the object and corresponds to a rotation along the z-axis (the vertical axis). It indicates the direction where the object is heading: $\psi = 0°$ represents East, while $\psi = 90°$ corresponds to North.

Following this convention, rotations are performed in alphabetical order: x-axis, y-axis, and finally z-axis. For different conventions, the axes must always be rotated in the roll, pitch, yaw order (RPY). When considering a y-up system, where the object faces the positive z-axis, rotations should be performed in the z, x, and y order.

Final world matrix   With this convention, positioning an object in space requires nine parameters: three for position, three for Euler angles, and three for scaling factors. Each parameter holds an intuitive physical significance.

The World Matrix M_W is then computed by factoring the five transformations in the specified order:
$$M_W = T(p_x, p_y, p_z) \times R_y(\phi) \times R_x(\theta) \times R_z(\varphi) \times S(s_x, s_y, s_z)$$

World matrix in GLM   GLM does not provide dedicated functions for constructing a World matrix directly from basic position, scales, and rotations. However, within a specific package of the library, there exists a specialized function for generating a rotation matrix from Euler angles:

```
#include <glm/gtx/euler_angles.hpp>

glm::mat4 MEa = glm::eulerAngleYXZ(glm::radians(yaw),
                                   glm::radians(pitch),
                                   glm::radians(roll));
```

It is typically more practical to define the entire sequence of transformations ourselves:

```
glm::mat4 Mw = glm::translate(glm::mat4(1.0), glm::vec3(px, py, pz)) *
               glm::rotate(glm::mat4(1.0), yaw , glm::vec3(0,1,0)) *
               glm::rotate(glm::mat4(1.0), pitch , glm::vec3(1,0,0)) *
               glm::rotate(glm::mat4(1.0), roll , glm::vec3(0,0,1)) *
               glm::scale(glm::mat4(1.0), glm::vec3(sx, sy, sz));
```

Conventions   In a real application, modelers and developers collaborate to define the convention that should be used to create assets suitable for integration into the project. An adaptation transformation matrix M_A can be appended before the world matrix to adapt the convention followed by third-party assets to the one required by the application:

$$M_W = T(p_x, p_y, p_z) \times R_y(\phi) \times R_x(\theta) \cdot R_z(\varphi) \times S(s_x, s_y, s_z) \cdot M_A$$

3.5   Quaternions

A rotation characterized by Euler angles is ideal for planar tasks, such as driving simulations or first-person shooters (FPS). However, they are not suitable for applications like flight or space simulators due to the potential issue of gimbal lock.

A gimbal is a circular ring capable of rotating around its diameter. To freely orient an object in space, a physical system requires a minimum of three interconnected gimbals. During rotations, the pitch movement affects the roll axis, and the yaw movement affects both the pitch and roll axes. When the pitch rotates exactly 90°, the roll and yaw axes align, resulting in a loss of one degree of freedom. This phenomenon is known as gimbal lock, where certain natural movements become impossible, necessitating complex combinations of the three basic rotations. One common solution to this issue is to represent object rotations using quaternions, a mathematical device. However, Euler angles are frequently preferred in many applications, particularly in virtual reality (VR), where gimbal lock rarely occurs, and using quaternions introduces additional mathematical complexity to the process.

Quaternions are an extension of complex numbers, featuring three imaginary components represented as:

$$a + ib + jc + kd$$

These imaginary components, collectively referred to as the vector part, adhere to specific relationships:

$$i^2 = j^2 = k^2 = ijk = -1$$

With this specification, a comprehensive algebraic system can be established, incorporating various operations:

- Addition:

$$(a_1 + ib_1 + jc_1 + kd_1) + (a_2 + ib_2 + jc_2 + kd_2) = (a_1 + a_2) + i(b_1 + b_2) + j(c_1 + c_2) + k(d_1 + d_2)$$

- Scalar product:

$$\alpha(a + ib + jc + kd) = \alpha a + i\alpha b + j\alpha c + k\alpha d$$

- Quaternion product:

$$(a_1 + ib_1 + jc_1 + kd_1)(a_2 + ib_2 + jc_2 + kd_2) = (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) +$$
$$+ i(a_1b_2 - b_1a_2 - c_1d_2 - d_1c_2) + j(a_1c_2 - c_1a_2 - d_1b_2 - b_1d_2) +$$
$$+ k(a_1d_2 - d_1a_2 - b_1c_2 - c_1b_2)$$

- Norm:
$$\|a + ib + jc + kd\| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

- Phase:
$$\theta = \arccos \frac{a}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

- Power:

$$(a + ib + jc + kd)^\alpha = \|a + ib + jc + kd\|^\alpha \left(\cos(\alpha\theta) + \frac{ib + jc + kd}{\sqrt{b^2 + c^2 + d^2}} \sin(\alpha\theta) \right)$$

3.5.1 Rotation

A unit quaternion q possesses a norm of $\|q\| = 1$. These unit quaternions are employed to encode 3D rotations.

Let's examine a rotation of an angle θ around an axis aligned with a unit vector $v = (x, y, z)$. This rotation can be succinctly represented using the following quaternion:

$$q = \cos \frac{\theta}{2} + \sin \frac{\theta}{2}(ix + jy + kz)$$

Given that v is unitary, q also is unitary. Quaternions can be directly transformed into rotation matrices:

$$R(q) = \begin{bmatrix} 1 - 2c^2 - 2d^2 & 2bc + 2ad & 2bd - 2ac & 0 \\ 2bc - 2ad & 1 - 2b^2 - 2d^2 & 2cd + 2ab & 0 \\ 2bd + 2ac & 2cd - 2ab & 1 - 2b^2 - 2c^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If q_1 and q_2 represent two distinct unit quaternions encoding separate rotations, their product encodes the composite transformation:

$$M_1 \cdot M_2 \Leftrightarrow q_1 \cdot q_2$$

With these definitions established, we can convert a set of Euler angles to a quaternion:

$$R = R_y(\phi) \cdot R_x(\theta) \cdot R_z(\varphi) \rightarrow q = \left(\cos \frac{\phi}{2} + j \sin \frac{\phi}{2} \right) \left(\cos \frac{\theta}{2} + i \sin \frac{\theta}{2} \right) \left(\cos \frac{\varphi}{2} + k \sin \frac{\varphi}{2} \right)$$

It's important to note that quaternion multiplication is non-commutative. As the order of rotations matters, the sequence of quaternion multiplication should mirror that of the corresponding matrices.

To derive Euler angles from a quaternion, the process begins by computing its corresponding rotation matrix. Subsequently, the angles are extracted from this matrix.

3.5.2   Usage

When dealing with complex rotations, an object's orientation is typically stored in memory using a quaternion, represented as q. When there's a need to calculate the world matrix, this quaternion is converted into the corresponding rotation matrix. Subsequently, this matrix is utilized for both the translation and scaling components via multiplication. In essence, the World Matrix M_W can be computed as follows:

$$M_W = T(p_x, p_y, p_z) \cdot R(q) \cdot S(s_x, s_y, s_z)$$

The application consistently conducts rotations using quaternion operations: any relative changes in an object's direction are encoded with a quaternion Δq, expressing both the direction and the amount of rotation.

   When the rotation quaternion is applied first, the rotation takes place in world space:

$$q = \Delta q \cdot q$$

When the rotation quaternion is positioned at the end, local space is employed:

$$q = q \cdot \Delta q$$

3.5.3   Quaternions in GLM

GLM provides quaternion support through both base and extended functions. To utilize quaternion functions, include the specific part of the GLM library:

```
#include <glm/gtc/quaternion.hpp>
```

Quaternions can be created in three ways:

1. From Euler angles, specified in the pitch, yaw, roll order:

   ```
   glm::quat qe = glm::quat(glm::vec3(0, glm::radians(45.0f), 0));
   ```

   Note: This constructor is rarely used due to the pitch, yaw, roll order. It's mainly suitable for building base quaternions corresponding to the three main rotations. To obtain a quaternion representing a proper rotation matrix with given Euler angles in the considered zxy order, compose the building blocks manually:

   ```
   glm::quat qe = glm::quat(glm::vec3(0, glm::radians(yaw), 0)) *
                  glm::quat(glm::vec3(glm::radians(pitch), 0, 0)) *
                  glm::quat(glm::vec3(0, 0, glm::radians(roll)));
   glm::mat4 MQ = glm::mat4(qe);
   ```

2. Specifying the scalar part first, followed by the i, j, k vector components:

   ```
   glm::quat qv = glm::quat(cos(glm::radians(22.5f)), 0,
                            sin(glm::radians(22.5f)), 0);
   ```

3. From the rotation angle and the axis direction, using the `rotate()` function:

```
glm::quat qa = glm::rotate(glm::quat(1,0,0,0), glm::radians(45.0f),
                           glm::vec3(0,1,0));
```

The scalar component can be accessed with the `.w` field, and the i, j, k vector components with the `.x`, `.y`, and `.z` fields, respectively:

```
std::cout << qe.x << "\t" << qe.y << "\t"
          << qe.z << "\t" << qe.w << "\n" ;
```

Algebraic operations, including the product among quaternions, can be performed using the usual symbols. For example, for the product:

```
glm::quat qp = qb * qa;
```

Compute the equivalent 4×4 rotation matrix by passing it as a constructor parameter to the `glm::mat4` matrix type:

```
glm::mat4 R = glm::mat4(qe);
```

Since rotations are encoded with unit quaternions, ensure this property using the `normalize` function starting from an arbitrary element:

```
glm::quat qn = glm::quat(-1, 0, 2, 3);
glm::quat qnn = glm::normalize(qn);
```

The extended functions offer various additional features such as Euler angle extraction, interpolation, and faster construction and conversion procedures.