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

CHAPTER 1

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).

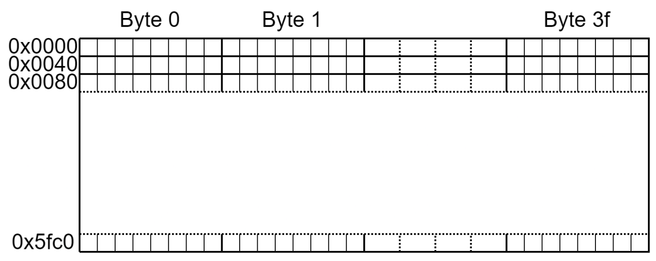

Figure 1.1: VRAM's structure

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 manufacturer 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.

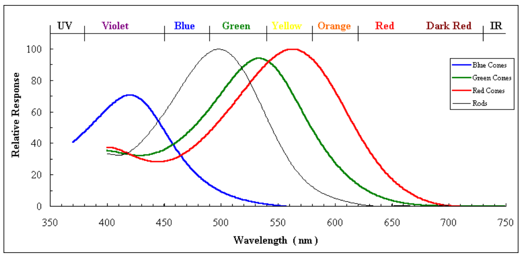

Figure 1.2: Human cones wavelength

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.

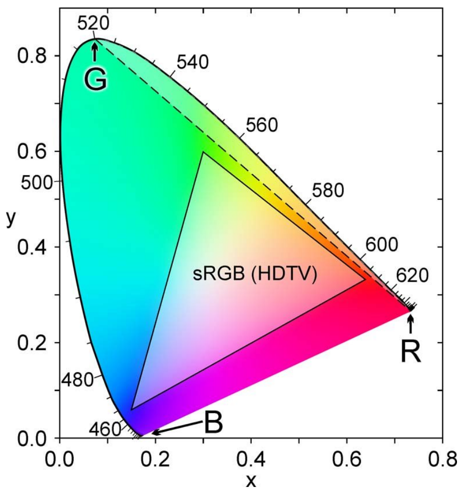

Figure 1.3: Colors perceived by human eyes and reproduced by a monitor

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.

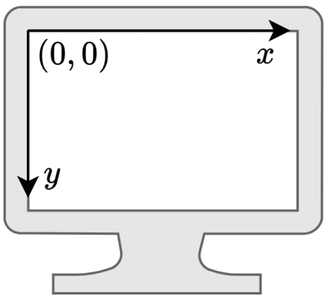

Figure 1.4: Pixel coordinates

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.

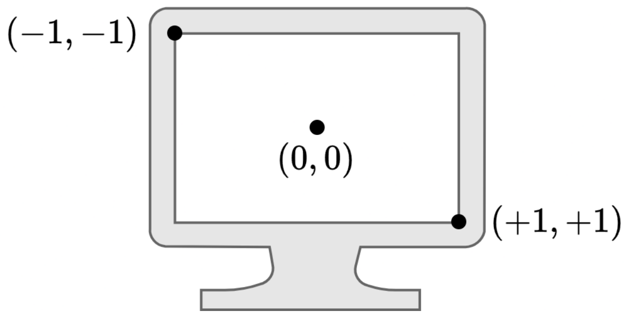

Figure 1.5: Normalized screen coordinates

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).

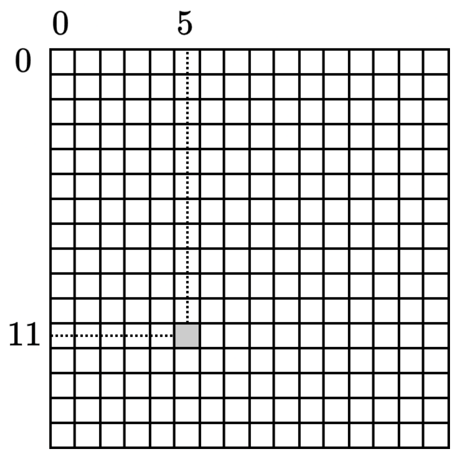

Figure 1.6: `plot(x:-0.312, y:0.562, r:0.8, g:0.8, b:0.8)`

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.

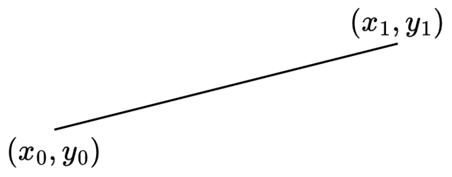

Figure 1.7: Line primitive

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).

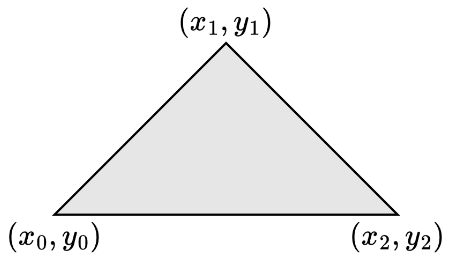

Figure 1.8: Filled triangle primitive

CHAPTER 2

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 $w = 1$:

$$(x', y', z') = (x, y, z, 1)$$

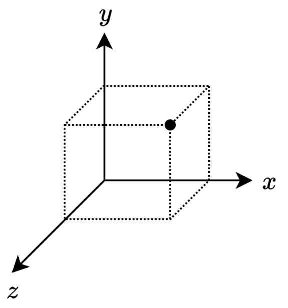

Figure 2.1: Coordinates

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:

$$M = I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

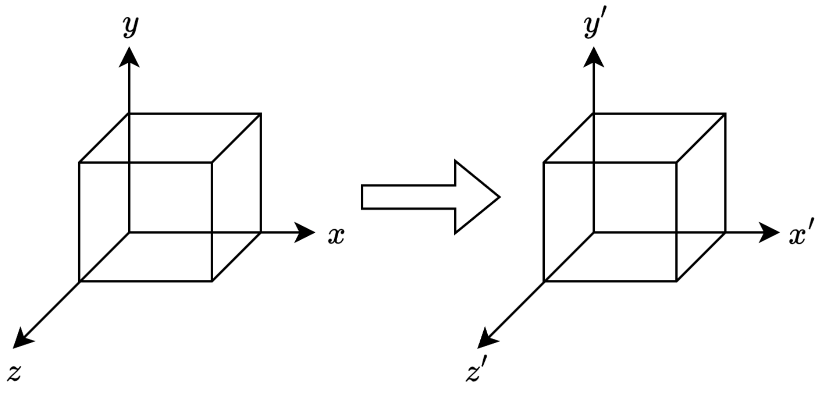

Figure 2.2: Identity transform

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:

$$M = T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

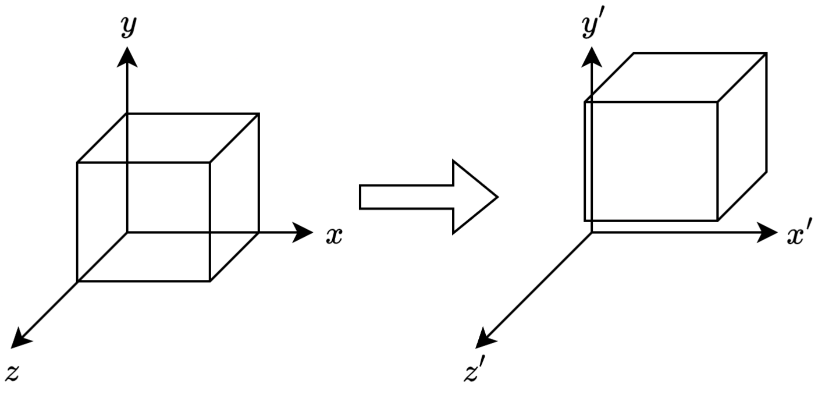

Figure 2.3: Translation transform

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 $s = s_x = s_y = s_z$.

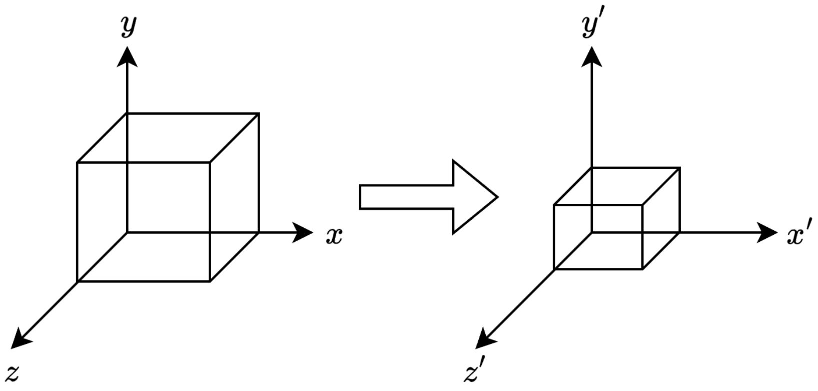

Figure 2.4: Scaling transform

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:

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

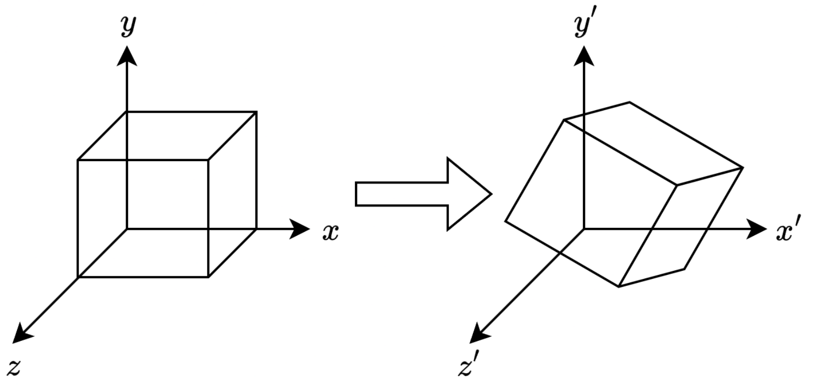

Figure 2.5: Rotation transform

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:

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

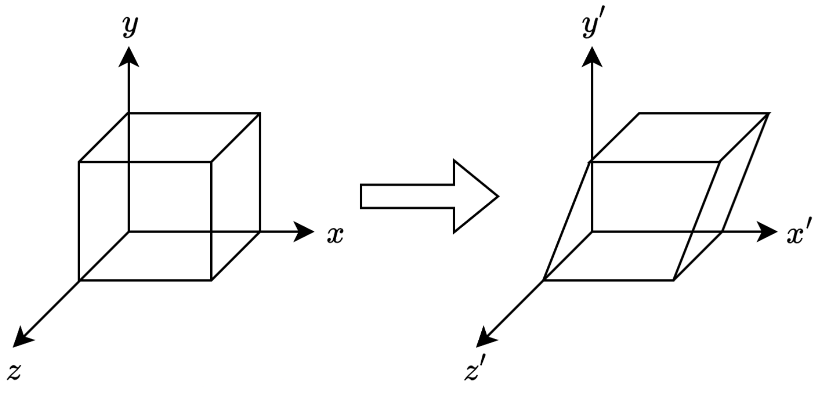

Figure 2.6: Shear transform

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.

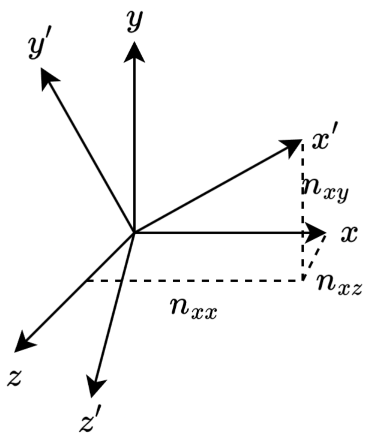

Figure 2.7: Transformation matrix M_R

The vector d^T represents the position of the origin of the new coordinate system in the old one.

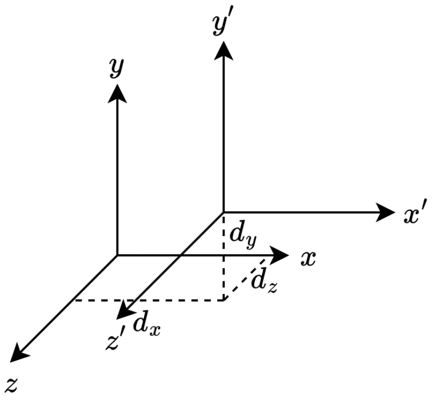

Figure 2.8: Transformation matrix d_t

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, 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.

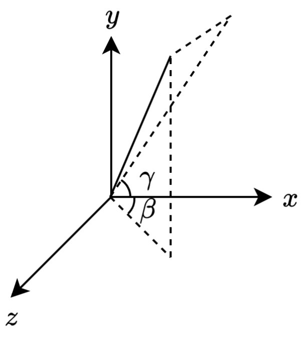

Figure 2.9: Rotation with an arbitrary axis

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:

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

2.5 Transformation properties

The associativity property holds for the product of two matrices and for the product of a matrix and a vector:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

$$A \cdot (B \cdot p) = (A \cdot B) \cdot p$$

Exploiting this property allows us to factorize all transformations into a single matrix. By computing the product of all transformations, a composed matrix is obtained, which can then be used to apply all transformations in a single operation.

Furthermore, the inverse of the product of two matrices follows a specific pattern:

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

This property enables us to compute the inverse of a composed transformation.

It's crucial to note that matrix multiplication is not commutative, meaning the order of transformations matters. Swapping two transformations results in a different outcome.

2.5.1 OpenGL Mathematics

To address the issue of non-commutativity in matrix operations, we can utilize GLM, a straightforward linear algebra library commonly employed in computer graphics.

GLM employs the same type names as those defined in the GLSL shading language. Being a header-only library simplifies compilation and linking processes. Leveraging C++ operator overloading capabilities, complex mathematical operations can be expressed using a simple syntax.

The necessary headers to include are:

- `#define GLM_FORCE_RADIANS`
- `#include glm/glm.hpp`
- `#include glm/gtc/matrix_transform.hpp`
- `#include glm/gtx/matrix_transform2.hpp`

The initial `define` ensures that all angles are interpreted as radians. The first `include` imports the entire library, while the other two imports the four principal transformations.

Matrix and vectors definitions The following commands utilize GLM's functionality to define and manipulate matrices and vectors in the context of computer graphics:

- To define 4×4 matrices:
  `glm::mat4`
- To create a matrix from its elements (passed per column):
  `glm::mat4(m11, m21, m31, m41, ..., m14, m24, m34, m44)`
- To create an identity matrix:
  `glm::mat4(1)`
- To define three-component vectors:
  `glm::vec3(m1, m2, m3)`
- To define four-component vectors:
  `glm::vec4(m1, m2, m3, m4)`

Note that to access the elements of a matrix we have to specify the index of the column followed by the index of the row:
`matrix[column_index][row_index]`

Matrix and vectors operations The possible operations, that needs to respect the dimensionality of them, are:

- To compute matrix product:
  `matrix1 * matrix2`
- To invert a matrix:
  `glm::inverse(matrix)`
- To transpose a matrix:
  `glm::transpose(matrix)`
- To perform algebraic operations between matrices and vectors:
  ```
  matrix + matrix
  matrix - matrix
  matrix * matrix
  ```

Transposition The transposition of a matrix can be done with the following command:
`glm::mat4 T = glm::translate(glm::mat4(1), glm::vec3(dx, dy, dz))`
This command creates a 4×4 translation matrix (`T`) with displacements `dx`, `dy`, and `dz`.

Scaling The scaling of a matrix can be done with the following command:
`glm::mat4 S = glm::scale(glm::mat4(1), glm::vec3(sx, sy, sz))`
This command generates a 4×4 scaling matrix (`S`) with scaling factors `sx`, `sy`, and `sz`. A shortcut for proportional scaling is: `glm::mat4 Sp = glm::scale(glm::mat4(1), glm::vec3(s))`

Rotation The rotation of a matrix can be done with the following command:
`glm::mat4 R = glm::rotate(glm::mat4(1), ang, glm::vec3(ax, ay, az))`
This command produces a 4×4 rotation matrix (`R`) of an angle `ang` along an axis specified by vector `ax`, `ay`, and `az`. Angles can be specified in radians using `glm::radians()` function. For rotations along the x, y, and z axes:

- `glm::mat4 Rx = glm::rotate(glm::mat4(1), ang, glm::vec3(1, 0, 0))`
- `glm::mat4 Ry = glm::rotate(glm::mat4(1), ang, glm::vec3(0, 1, 0))`
- `glm::mat4 Rz = glm::rotate(glm::mat4(1), ang, glm::vec3(0, 0, 1))`

Shear The shear of a matrix can be done with the following commands:

- `glm::mat4 Rx = glm::shearX3D(glm::mat4(1), hy, hz)`
- `glm::mat4 Rx = glm::shearY3D(glm::mat4(1), hx, hz)`
- `glm::mat4 Rx = glm::shearZ3D(glm::mat4(1), hx, hy)`

These commands create 4×4 shear matrices along the x, y, and z axes with shear factors `hx`, `hy`, and `hz`.