# 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


**1 Introduction** **1**
- 1.1 Graphic adapter . . . 1
- 1.2 Colors . . . 2
  - 1.2.1 Physical phenomena . . . 2
  - 1.2.2 Color reproduction . . . 2
- 1.3 Image resolution . . . 3
  - 1.3.1 Coordinates . . . 3
  - 1.3.2 Normalized screen coordinates . . . 4
  - 1.3.3 Graphics primitives . . . 5

**2 Three-dimensional geometry** **7**
- 2.1 Homogeneous coordinates . . . 7
- 2.2 Affine transforms . . . 8
  - 2.2.1 Identity transform . . . 8
  - 2.2.2 Translation . . . 9
  - 2.2.3 Scaling . . . 9
  - 2.2.4 Rotation . . . 11
  - 2.2.5 Shear . . . 12
  - 2.2.6 Transformation matrix . . . 12
- 2.3 Transformation inverse . . . 14
- 2.4 Transformation composition . . . 15
  - 2.4.1 Transformations around an arbitrary axis . . . 15

**3 Projections** **17**
- 3.1 Introduction . . . 17
- 3.2 Parallel projections . . . 19
  - 3.2.1 Orthogonal projections . . . 19
  - 3.2.2 Aspect ratio . . . 20
  - 3.2.3 Axonometric projections . . . 22
  - 3.2.4 Oblique projections . . . 23
- 3.3 Perspective projections . . . 24
- 3.4 World coordinates . . . 28
  - 3.4.1 Look-in-direction . . . 28
  - 3.4.2 Look-at . . . 29
  - 3.4.3 Local coordinates and world matrix . . . 31
  - 3.4.4 World matrix definition . . . 31
- 3.5 Quaternions . . . 33

3.5.1 Rotation . . . 34
3.5.2 Usage . . . 35
3.5.3 Quaternions in GLM . . . 35
3.6 Summary . . . 36
3.6.1 Phases . . . 36

**4 Meshes** **38**
4.1 Introduction . . . 38
4.2 Mesh encoding . . . 39
4.3 Index primitives . . . 39
4.4 Smooth shading . . . 40
4.4.1 Vertex normal vectors . . . 41
4.4.2 Gouraud shading . . . 42
4.4.3 Phong shading . . . 42

**5 Optimization and visualization problems** **43**
5.1 Introduction . . . 43
5.2 Back-face culling . . . 43
5.3 Depth testing . . . 44
5.3.1 Z-buffer . . . 44
5.3.2 Stencil buffer . . . 45
5.4 Clipping . . . 45
5.4.1 Half spaces . . . 46
5.4.2 Points . . . 46
5.4.3 Triangles . . . 47
5.4.4 Remarks . . . 47
5.5 Screen synchronization update . . . 48
5.5.1 Double buffering . . . 48
5.5.2 Triple buffering . . . 48

**6 Rendering** **49**
6.1 Introduction . . . 49
6.2 Rendering equation . . . 50
6.2.1 Bidirectional reflectance distribution function . . . 50
6.2.2 The equation . . . 50
6.2.3 Lights basics . . . 52
6.3 Pipelines . . . 53
6.3.1 Taxonomy . . . 53
6.4 Vulkan pipeline . . . 54
6.4.1 Shaders . . . 56
6.5 Ray casting . . . 57
6.6 Ray tracing . . . 57
6.6.1 Ray-tracing pipeline . . . 58
6.6.2 Algorithm . . . 60
6.7 Radiosity . . . 61
6.7.1 Algorithm . . . 61
6.8 Montecarlo techniques . . . 62
6.9 Mesh shader pipeline . . . 62
6.10 Compute pipeline . . . 63

6.11 OpenGL Shading Language . . . 63
6.11.1 Vertex shader . . . 63
6.11.2 Fragment shader . . . 64
6.11.3 Shader-pipeline communication . . . 65
6.12 Textures . . . 65
6.12.1 UV coordinates . . . 66
6.12.2 Rasterization with textures . . . 66
6.12.3 Cube map textures . . . 67
6.12.4 Texture filtering . . . 67

**7 Lighting** **69**
7.1 Introduction . . . 69
7.1.1 Light color . . . 69
7.2 Direct light models . . . 70
7.3 Point light model . . . 70
7.4 Spot light model . . . 71
7.5 Area lights . . . 72
7.6 Bidirectional reflectance distribution function . . . 72
7.6.1 High Dynamic Range . . . 72
7.7 Diffuse reflection models . . . 73
7.7.1 Lambert reflection . . . 73
7.7.2 Toon shading . . . 74
7.7.3 Oren-Nayar . . . 74
7.8 Specular reflection . . . 75
7.8.1 Phong reflection model . . . 76
7.8.2 Blinn reflection model . . . 76
7.8.3 Ward anisotropic specular model . . . 77
7.8.4 The Cook-Torrance reflection model . . . 77
7.9 Emission and indirect lighting approximation . . . 79
7.9.1 Material emission . . . 79
7.9.2 Ambient lighting . . . 79
7.9.3 Image based lighting . . . 80

**8 Vulkan** **82**
8.1 Introduction . . . 82
8.1.1 Starter library . . . 82
8.1.2 Presentation surface . . . 83
8.2 Vulkan initialization . . . 83
8.2.1 Typical Vulkan application . . . 84
8.2.2 Presentation surface . . . 85
8.2.3 Physical devices . . . 85
8.2.4 Command buffer . . . 86
8.2.5 Vulkan swap chain . . . 86
8.3 Layout . . . 87
8.3.1 Data structure . . . 88
8.3.2 Vertex and fragments shaders . . . 89
8.3.3 Uniform buffers . . . 89
8.3.4 Descriptor Buffers . . . 91
8.3.5 Binding and Textures . . . 91

8.3.6 Texturing . . . . . 92

# 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).

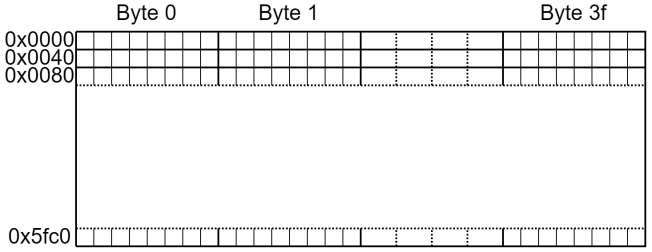



Figure 1.1: VRAM's structure

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 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.

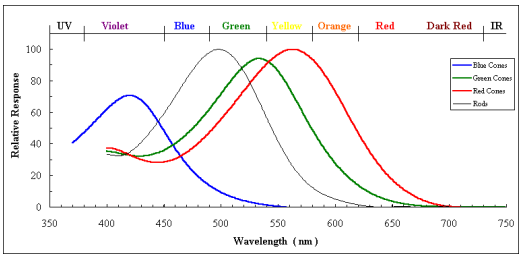



Figure 1.2: Human cones wavelength

### 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.

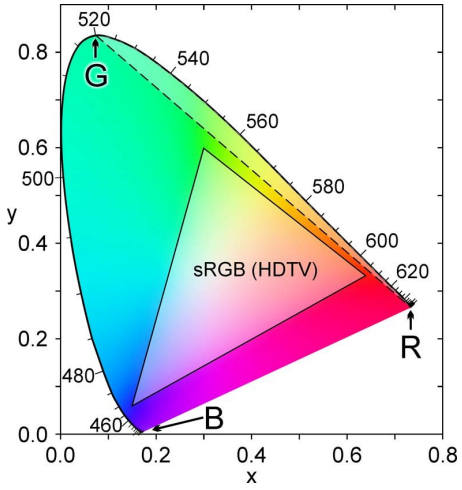



Figure 1.3: Colors perceived by human eyes and reproduced by a monitor

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.

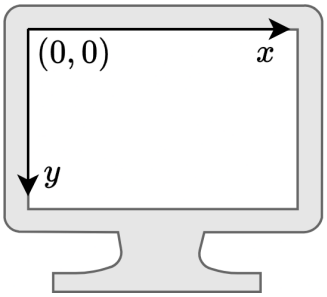



Figure 1.4: Pixel coordinates

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.

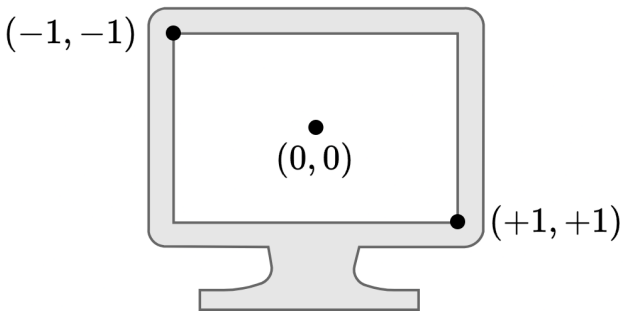



Figure 1.5: Normalized screen coordinates

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 $(r, g, b)$.

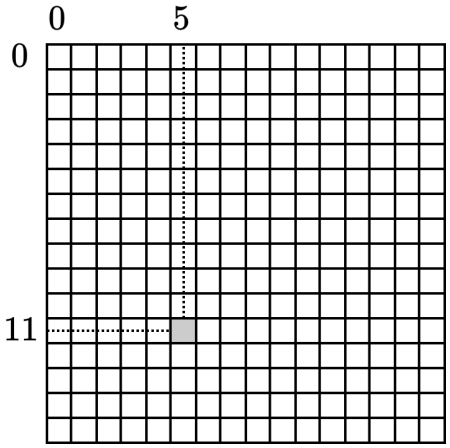



Figure 1.6: `plot (x:-0.312, y:0.562, r:0.8, g:0.8, b:0.8)`

**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.

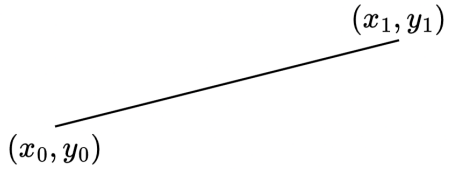



Figure 1.7: Line primitive

**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 $(r, g, b)$.

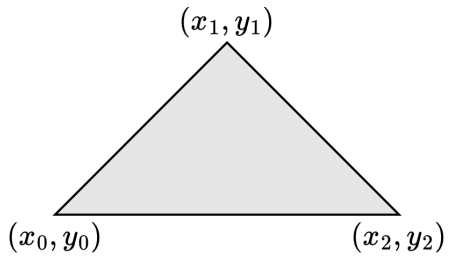



Figure 1.8: Filled triangle primitive

# 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 $w = 1$:

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

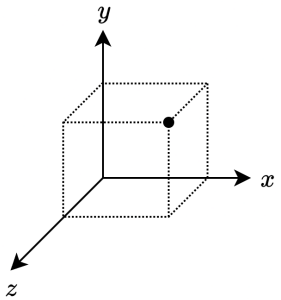



Figure 2.1: Coordinates

## 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, $4 \times 4$ 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 $4 \times 4$ identity matrix:

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

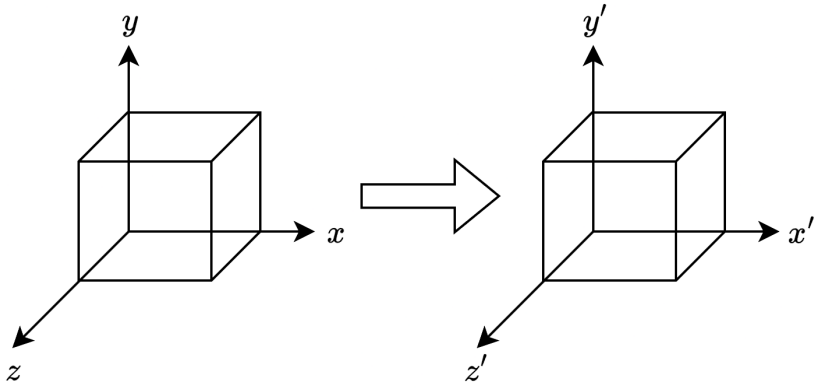



Figure 2.2: Identity transform

### 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:

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

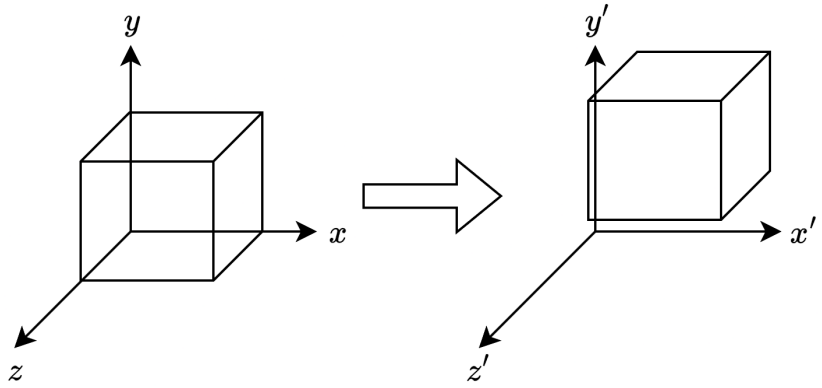



Figure 2.3: Translation transform

### 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 $s = s_x = s_y = s_z$.

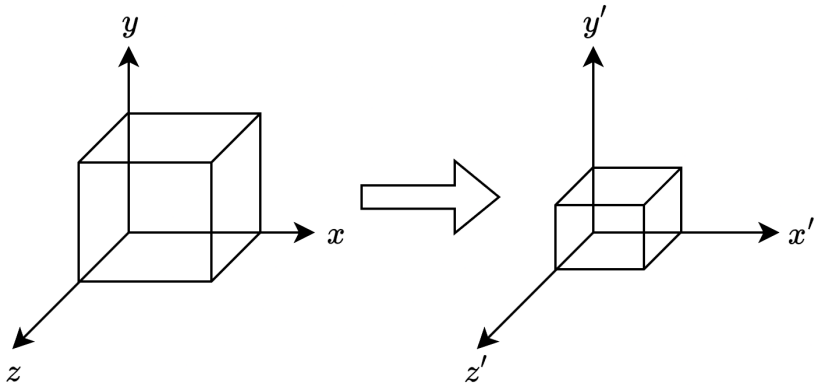



Figure 2.4: Scaling transform

**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 $\alpha$ 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 $\alpha$ 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:

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

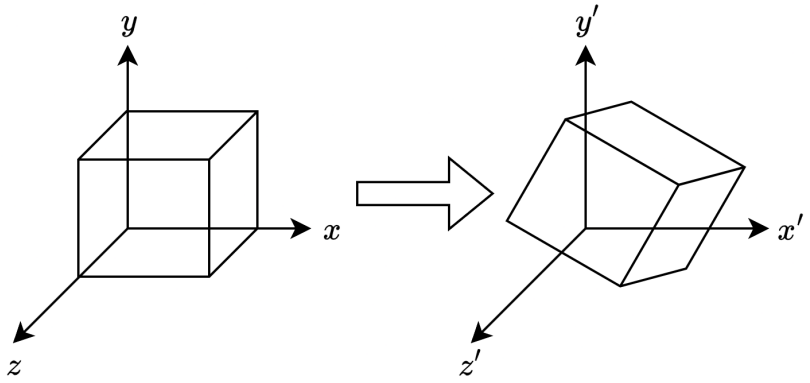



Figure 2.5: Rotation transform

### 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:

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

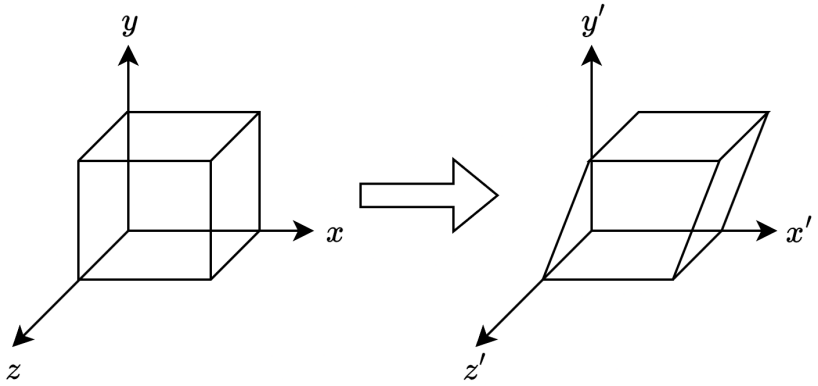



Figure 2.6: Shear transform

### 2.2.6 Transformation matrix

It's important to note that in all the $4 \times 4$ 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 $3 \times 3$ 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.

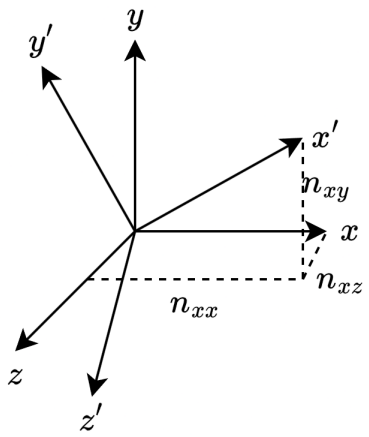



Figure 2.7: Transformation matrix $M_R$

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

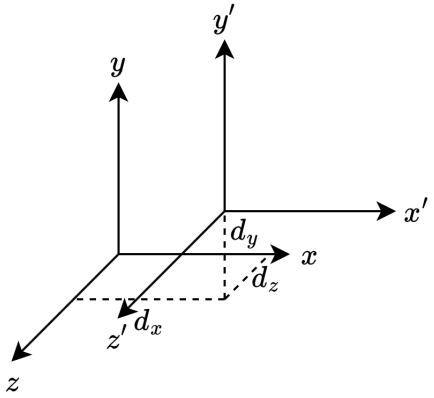



Figure 2.8: Transformation matrix $d_t$

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:

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

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 $\alpha$ 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 $\alpha$ 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 $\gamma$ around the $z$-axis, followed by an angle $\beta$ around the $y$-axis.

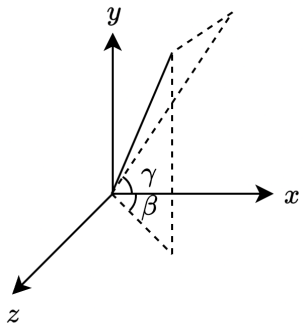



Figure 2.9: Rotation with an arbitrary axis

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 $\alpha$ about an arbitrary axis passing through the point $(p_x, p_y, p_z)$, aligning the $x$-axis by rotating $\gamma$ around the $z$-axis and $\beta$ 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:

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

# 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 ±45° around the $y$-axis, followed by a rotation of ±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.

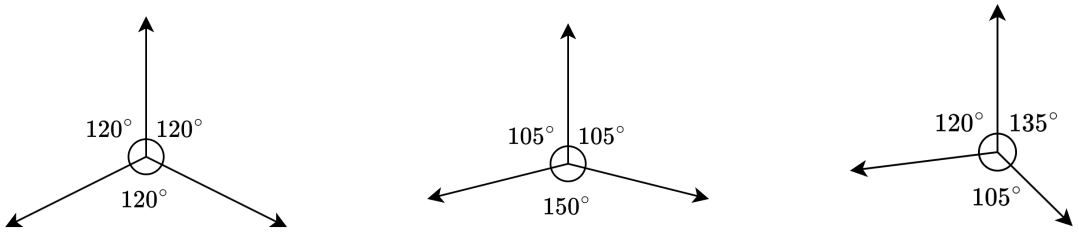
- *Dimetric*: in dimetric projection, distinct units are used for the $x$ and $z$-axes compared to the $y$-axis. Its popularity in the 1980s stemmed from its simplicity in implementation through integer arithmetic. Even today, dimetric projection remains prevalent in retro-style games and applications.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

- 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.

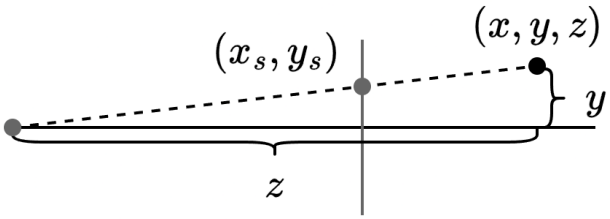



Figure 3.4: Perspective projection with respect to $y$-axis

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.

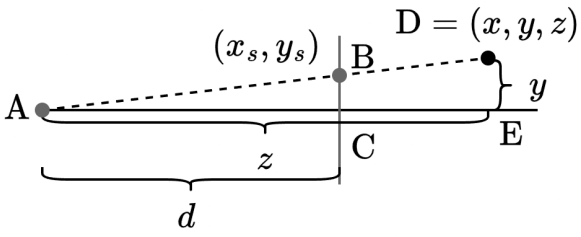



Figure 3.5: Perspective projection with respect to $y$-axis with triangles

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 $\Theta$ 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 $\alpha$), the elevation (angle $\beta$), and the roll over the viewing direction (angle $\rho$). 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 $4 \times 4$ 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 $\rho$ 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 $\phi$, represents the rotation of the object along its facing axis (the $x$-axis). A positive roll turns the object clockwise.

The Pitch, denoted as $\theta$, 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 $\psi$, 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:

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

- 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 $\theta$ 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 $\Delta q$, 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 $4 \times 4$ 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.

## 3.6 Summary

To translate the local coordinates defining a 3D model (as exported from tools like Blender) into pixel positions on screen, five sequential steps must be executed: world transform, view transform, projection, normalization, and screen transform. Each step facilitates the transformation of coordinates from one space to another.

The initial three steps (and potentially the final one) can be executed through a matrix-vector product. However, Normalization mandates a distinct procedure that cannot be merged with the others.

### 3.6.1 Phases

The steps needed to have the pixels on screen are:

1. An object is initially modeled in local coordinates $p_M$. Typically, these local coordinates are 3D Cartesian, and they are first converted into homogeneous coordinates $p_L$ by appending a fourth component equal to one. The world transform then transitions the coordinates from local space to global space by multiplying them with the world matrix $M_W$.

2. The View transform enables visualization of the 3D world from a specific viewpoint in space. It converts coordinates from global space to camera space using the view matrix $M_V$, often generated using either the look-in-direction or look-at techniques.

3. The Projection Transform readies the coordinates for display on the screen by executing either a parallel or perspective projection. In the case of parallel projections, it employs a parallel projection matrix $M_{P-ort}$ to convert camera space coordinates to normalized screen coordinates. For perspective projections, a perspective projection matrix $M_{P-persp}$ is utilized. In this scenario, the outcomes are not yet normalized screen coordinates but rather an intermediate space known as clipping coordinates.

4. In many cases, the world, view, and projection matrices are combined into a single matrix, known as the world-view-projection matrix (MWVP). During perspective projections, the Normalization step converts clipping coordinates into normalized screen coordinates. Unlike the other transformations, this step involves converting homogeneous coordinates describing points in clipping space into Cartesian coordinates. Specifically, each coordinate is divided by the fourth component of the homogeneous coordinate vector, and then the last component (which is always one) is discarded. This step isn't required in parallel projections because the $M_{P-ort}$ matrix already provides normalized screen coordinates. In this case, it suffices to discard the last component, which should already be one.

5. In a typical 3D application, the world-view-projection operation is performed, and the resulting clipping coordinates, which define the primitives to be displayed, are sent to the underlying framework. The video card driver then converts these clipping coordinates, if required, first into normalized screen coordinates and then into pixel coordinates for visualizing the objects. This conversion process is conducted in a manner that is transparent to the end user.

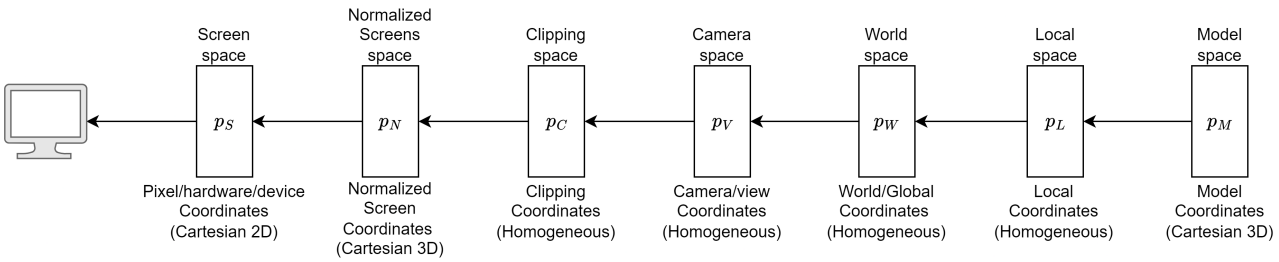



Figure 3.6: World-view projection matrices

# CHAPTER 4

# Meshes

## 4.1 Introduction

Natural objects exhibit a blend of glossy, bumpy, linear, and curved surfaces. In the digital realm, 3D assets serve as virtual renditions of these objects, encoded digitally. While we've discussed techniques for determining on-screen positions of points in 3D space, objects typically aren't represented as disjointed point clouds due to the immense resources such a representation would demand to create an illusion of solidity.

Instead, solid objects are initially stored solely by their boundaries, with their internal content often disregarded. This initial step simplifies the storage and processing requirements for digital representations of objects.

The encoding of object geometry relies on mathematical models that articulate surfaces using a series of parameters. Computational geometry delves into the optimal methods for mathematically articulating these surfaces. Numerous methodologies have been outlined in scholarly literature, with the most prevalent being: meshes (polygonal surfaces), hermite surfaces, NURBS (Non-Uniform Rational B-Splines), HSS (Hierarchical Subdivision surfaces), and metaballs. 3D authoring tools such as Blender typically offer users the flexibility to choose from various techniques for encoding the models they create. Each technique necessitates distinct sets of tools and modeling proficiencies. Every mathematical model possesses unique characteristics and constraints. Nonetheless, when it comes to rendering, all models are ultimately converted into meshes, or polygonal surfaces. Therefore, our focus will be primarily on meshes.

**Polygonal surfaces** Polygonal surfaces represent objects described by a series of connected polygons. With specialized rendering features, these surfaces can effectively approximate curved shapes. Each polygon describing a planar section of the object's surface is termed a face, while the edges denote the sides of the polygons, typically where two faces intersect. Vertices mark the beginning and end points of edges, with each edge being delimited by exactly two vertices. In a proper solid, at least three faces intersect at a vertex.

**Two-manyfold** In the context of topology, surfaces where every edge is adjacent to precisely two faces exhibit a unique structure known as a 2-manyfold. Surfaces that deviate from this pattern usually depict non-physical objects, and if utilized, necessitate careful attention to ensure accurate rendering.

At times, non-2-manifold objects are employed to minimize the polygon count required for encoding extremely thin objects or to achieve specific visual effects. However, algorithms such as backface-culling are incompatible with non-2-manifold objects and may not function as intended.

**Triangles and meshes** Each polygon can be subdivided into a collection of triangles that share certain edges. This assemblage of adjacent triangles forms what is known as a mesh.

Given that three non-collinear points define a unique plane in space, and likewise three non-collinear points define a triangle, it follows that polygons with more than three vertices may not necessarily represent a single planar surface. Consequently, there exist various possible arrangements for connecting more than three points to delineate a surface in space. This necessitates the conversion of every polygon, whether planar or not, into a collection of triangles through a process known as polygon tessellation. Notably, polygon tessellations are not singular; multiple tessellations can be generated for each polygon with more than three sides.

In a mesh representation of an object, its surfaces are stored using the collection of polygons that form its boundaries. These boundary polygons are subsequently transformed into a series of interconnected triangles that share certain edges.

## 4.2 Mesh encoding

Mesh encoding relies on sets of vertices, which serve as the building blocks for defining the geometry. The rendering engine utilizes these vertices to establish the endpoints of the triangles composing the mesh. Typically, vertex coordinates are stored as Cartesian coordinates to optimize memory usage. Before applying transformations using the world view projection matrix, the rendering engine adds a fourth component set to one to ensure homogeneous coordinates.

It's evident that many triangles within a mesh share a significant number of vertices. Leveraging this characteristic is crucial for minimizing the memory footprint required to encode an object.

Various types of mesh encoding have been proposed in the literature, but only two are standard in Vulkan:

- *Triangle lists*: each triangle is encoded as a trio of distinct coordinates, without reusing any vertices. They are suitable for encoding disconnected triangles. Encoding $N$ triangles necessitates $3N$ vertices.
- *Triangle strips*: A series of contiguous triangles forming a band-like surface. The encoding starts with the first two vertices, with each subsequent vertex connected to the preceding two. To encode $N$ triangles, $N + 2$ vertices are required.

**Properties** In some cases, triangle strips may not be applicable even when the topology appears suitable for this type of encoding. This limitation arises because a vertex is typically defined by more parameters than just its local coordinates. For triangle strip encoding to be viable, shared vertices must be identical across all parameters.

## 4.3 Index primitives

While triangle strips can offer significant savings, often up to two-thirds compared to triangle lists, many scenarios still entail repeated vertices. Due to the inability to encode certain

primitives with a single triangle strip, vertices may be shared among different strips. Indexed primitives mitigate this issue by reducing the overhead of duplicating vertices across lists or strips.

Consider a sphere, typically composed of multiple strips where each vertex is shared by at least four triangles. Although triangle strips can decrease the space needed for each band, the same vertex is reiterated across various bands.

Indexed primitives employ two arrays: the vertex array containing vertex definitions (positions), and the index array specifying triangles indirectly. Triangles are drawn based on their indices, with vertex coordinates retrieved from the vertex list according to the index's position.

Indexing achieves substantial byte savings without the complexity of determining correct ordering. Consequently, many file interchange formats exclusively support meshes encoded with indexed triangle lists to streamline their architecture.

**Restart** In Vulkan, the restart feature enables additional space reduction in band-like structures. Specifically, a negative index restarts the strip, optimizing the encoding process.

**Lines** When creating 2D plots and charts, lines are typically employed instead of triangles.

**Wireframe** Lines are also utilized to generate Wireframe views, showcasing only the outlines of objects by connecting their vertices with lines. These views are valuable in numerous scenarios, particularly for debugging 3D applications.

The two primary types of wireframe mesh encoding are:

- *Line lists*: each segment is encoded as a pair of separate vertices. Encoding $N$ segments necessitates $2N$ vertices.
- *Line strips*: a path of connected vertices is encoded. To represent $N$ segments, $N + 1$ vertices are required.

Wireframe primitives can also be indexed. Additionally, Vulkan supports drawing standard objects solely using the contour of their triangles. However, this approach is simply a means to simplify creating a wireframe object corresponding to the mesh's wireframe.

## 4.4 Smooth shading

Meshes, typically polygonal objects with sharp edges, can be made to appear smooth through specialized rendering techniques. This smoothing effect not only enhances visual quality but also addresses performance considerations, particularly regarding how frequently the rendering equation is solved.

The solution to achieve smooth shading can be computed either per-vertex or per-pixel.

In modern applications, 3D models often consist of approximately 100,000 vertices and occupy a substantial number of pixels on the screen, typically over 2,000,000. To achieve smoother images, the rendering equation may need to be solved multiple times per pixel to mitigate the aliasing effect. While solving the rendering equation per pixel yields visually appealing results, it comes at the cost of a significant performance reduction, typically around one order of magnitude. The two primary techniques for smooth shading are:

- *Gouraud shading* (per-vertex): smoothing is achieved by blending the colors generated at each vertex.

- *Phong shading* (per-pixel): smoothing is accomplished by interpolating the parameters passed to the shaders involved in solving the rendering equations.

### 4.4.1 Vertex normal vectors

Let's envision a curved surface. To represent this surface, a mesh approximates it by sampling a finite set of points. The finer the sampling, the closer the approximation to the actual surface.

To enhance the rendering of curved surfaces, each vertex is extended to include six values. These values define both the vertex's position and the direction of the normal vector to the surface at that 3D point:

$$\mathbf{v} = (x, y, z, n_x, n_y, n_z)$$

As we've discussed in previous lessons, the rendering equation leverages the normal vector's direction to compute the color of a surface. In the context of a mesh, the rendering equation determines the colors of the pixels at the three vertices of each triangle based on the associated normal vectors. Subsequently, the colors of the interior pixels of the triangles are computed using interpolation techniques.

**Vertex normal vectors directions** The normal vector associated with a vertex may differ from the geometric normal determined by the triangle to which it belongs. In general, it can be an arbitrary vector that might not necessarily align with the associated surfaces. However, normal vectors that are completely uncorrelated with the geometry are rarely useful. Under this definition, the direction of the normal vector is a property of a (vertex, triangle) pair, rather than solely of a vertex. Therefore, two triangles may share a vertex in the same spatial position but characterized by different normal vector directions. Consequently, each triangle is defined by 18 values: three vertices, each comprising three vertex coordinates and three normal vector direction components. Modelers can leverage these properties to encode both smooth and sharp surfaces. For instance, vertices in the same position across adjacent triangles may possess the same normal vector direction to produce a smooth surface. Conversely, vertices in different triangles but with the same position yet different normal vector directions can be used to encode sharp surfaces.

**Transforming vertex normal** When normal vectors are stored with an object that undergoes a transformation encoded in a World Matrix, they need to be transformed accordingly. However, transforming normal vectors is not as straightforward as transforming vertex positions. The primary distinction lies in the fact that normal vectors represent 3D directions (three-component vectors), whereas vertex positions are represented as homogeneous coordinates (four-component vectors). To transform normal vectors correctly, we derive the transformation matrix from the 4x4 matrix responsible for transforming homogeneous coordinates. This involves computing the inverse-transpose of the 3x3 upper-left sub-matrix.

$$M = \begin{bmatrix} & M_l & & d_x \\ & & & d_y \\ & & & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} > M_n = \begin{bmatrix} & M_l^{T-1} & \\ & & \end{bmatrix}$$

In a special scenario where the transpose of the 3x3 rotation matrix is identical to its inverse, the proposed procedure has no effect. This scenario occurs when there are no scaling or shear

transformations applied. In such cases, we can directly transform the normal vector with the world matrix $M$ using a simple trick:

$$n' = \left(M \cdot \begin{bmatrix} n_x & n_y & n_z & 0 \end{bmatrix}\right)^T .xyz$$

### 4.4.2 Gouraud shading

Gouraud shading, a per-vertex shading technique, determines the colors of pixels inside a triangle by interpolating the colors of its vertices. This interpolation is based on geometric principles: the position of a point within a triangle can be calculated using a convex linear combination of its vertices. Similarly, the coefficients used for interpolating the positions can be employed for interpolating colors:

- For any internal point $p$ within a triangle, its coefficients are established via a linear system of equations.
- These coefficients are then utilized for color interpolation.

For objects that remain static in the scene, illuminated by fixed lights, and whose material BRDF does not vary with the observer's direction (for instance, solely the diffuse component following the Lambert model), pre-computed vertex colors can be stored along with the geometry. This enables the real-time rendering process to disable computation of the light model, replacing vertex normal vectors with vertex colors.

The pre-computation of colors is typically carried out within 3D authoring software like Blender. Additionally, vertex colors tend to occupy less memory compared to normal vector directions (i.e., 4 bytes versus 12 bytes).

### 4.4.3 Phong shading

The Phong shading algorithm computes the color of each internal pixel individually, making it a per-pixel shading technique. Here, vertex normal vectors are interpolated to approximate the normal vector direction of the actual surface at the internal points of the triangle.

Interpolation is conducted by treating the $x, y, z$ components of the normal vectors separately. However, this approach may result in interpolated vectors that are no longer unitary, even if the normal vectors associated with the triangle's vertices are unitary. Consequently, the interpolated normal vectors must be normalized at each step to maintain the required size.

For every pixel, the illumination model is then computed using the interpolated normal vectors along with other constants necessary for the light model and the BRDF in the rendering equation.

Compared to the Gouraud method, Phong shading is significantly more computationally expensive because it necessitates solving the rendering equation for each pixel. This can greatly reduce performance, especially when multiple light sources are considered, as the illumination model exhibits linear complexity concerning the number of lights. However, Phong shading can produce high-quality rendering even for models with few vertices.

While the Gouraud technique may generate artifacts in the image and fail to capture specific lighting conditions due to interpolation potentially overlooking details between vertices, both methods tend to yield similar results when applied to geometries composed of many vertices. This is because, in such cases, the area of each triangle is only a few pixels wide.

# CHAPTER 5

# Optimization and visualization problems

## 5.1 Introduction

Four techniques employed in real-time computer graphics to tackle specific optimization and visualization challenges include:

- Back-face culling.
- Depth testing (and stencil buffer).
- Clipping.
- Screen update synchronization.

Each of these techniques can be fine-tuned and configured within Vulkan, influencing various aspects of the visualization process.

## 5.2 Back-face culling

Back-face culling effectively removes faces on the rear side of a mesh by evaluating the vertex order of triangles concerning the viewer, determining if they're arranged clockwise or counter-clockwise.

This evaluation can take place before projection or using normalized screen coordinates. Vulkan's implementation of back-face culling operates within normalized screen coordinates, hence our focus on this approach.

Suppose all triangles of a mesh are consistently encoded with a specific orientation, such as clockwise. When projected onto the screen, front faces maintain their clockwise order, while back faces are ordered oppositely.

To compute the orientation of triangle vertices in normalized screen coordinates, a straightforward cross-product can be employed. If the resultant vector points toward the viewer (with a positive z component), the vertices are ordered clockwise. Because only the sign of the $z$ component matters, this test can be executed efficiently:

$$\begin{bmatrix} u_x & u_y & u_z \end{bmatrix} \times \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y & u_z v_x - u_x v_z & u_x v_y - u_y v_x \end{bmatrix}$$

**Remarks** There are transformations that can alter the vertex ordering, thus affecting the effectiveness of back-face culling. Additionally, not all 3D asset creation software follows the same convention for defining front faces. Therefore, users must have the ability to specify whether faces with vertices ordered clockwise or counterclockwise should be displayed.

While back-face culling can enhance application performance, there are potential issues that need consideration. Firstly, if a world matrix involves scaling with an odd number of negative coefficients (either one or all three), the acceptance test must be inverted, indicating that back-face culling may not always be applicable. Moreover, in non-2-manifold objects, those with holes or thin faces, certain polygons belonging to the back faces that should be visible might inadvertently be deleted.

If a camera is positioned inside an object where back-face culling is enabled, its boundaries become invisible since their vertices are oriented in the opposite direction. To address this, an object delineating the borders of the 3D world area (e.g., a room or a skybox) should be created with vertices ordered in the opposite direction.

## 5.3 Depth testing

In complex scenes, objects often overlap, necessitating that polygons closer to the observer obscure those behind them. Failure to draw faces in the correct order, as briefly discussed in the context of 3D normalized screen coordinates, can result in unrealistic visuals. Maintaining the appropriate visualization order within a set of primitives is known as hidden surfaces elimination.

**Painter algorithm** When dealing with non-transparent objects, the painter algorithm is commonly employed. This technique involves drawing primitives in reverse order based on their distance from the projection plane. Consequently, objects nearer to the viewer overlay those further away. However, there are scenarios where determining the correct order becomes challenging, rendering the straightforward application of the painter algorithm impractical through mere object sorting. In such cases, the solution lies in splitting the primitives to facilitate the determination of an appropriate order for the individual pieces.

### 5.3.1 Z-buffer

The $z$-buffer technique operates at the pixel level, often referred to as depth-testing due to its association with the distance from the projection plane. This method relies on a dedicated memory area known as the $z$-buffer or depth-buffer, storing additional information for each pixel on the screen. Here's how the algorithm works:

- All primitives in the scene are drawn, and pixels on the screen are tested for rendering.
- For every pixel, both color and distance from the observer are computed.
- The $z$-buffer stores the normalized screen $z$-coordinate (distance from the observer) for each pixel.
- When a new pixel is written to the same position, its distance from the observer is compared with the value stored in the $z$-buffer.
- If the distance of the new pixel is less than the stored value in the $z$-buffer, the new pixel is drawn on screen, and the $z$-buffer value is updated.

- If the distance of the new pixel is greater than the stored value in the $z$-buffer, the new pixel is discarded, as it corresponds to an object behind the one already displayed, and no update is performed.

While the $z$-buffering technique is straightforward, it requires an additional memory area to store distance information for all pixels. In Vulkan, programmers must create this memory area, making z-buffer usage more complex compared to other environments. Additionally, it necessitates generating pixels for all primitives, even if they are fully obscured by other objects.

The main challenge with $z$-buffering lies in numerical precision. Since a significant portion of the [0,1] range of normalized screen $z$-coordinates is used for points near the projection plane, sufficient precision is required to store distances for objects further away. Failure to maintain precision may lead to z-fighting, where the final color is determined by the round-off of distances between two nearly co-planar figures.

As normalized screen $z$-coordinates are relative to the near and far planes, these parameters cannot be arbitrarily set small or large. They must be chosen appropriately for the specific scene to prevent issues.

### 5.3.2 Stencil buffer

The stencil buffer technique, akin to the $z$-buffer, serves to restrict drawing within specified regions of the screen. Similar to $z$-buffering, it involves storing additional information for each pixel on the screen in a designated memory area known as the stencil buffer. Here's an overview of stencil buffer functionality:

- Stencil buffer usage enables applications to limit drawing to particular areas of the screen.
- This technique employs a special memory area, the stencil buffer, to store information for each pixel on the screen.
- A common application of the stencil buffer is to define rendering areas of arbitrary shapes, such as reserving space to depict the cabin of a ship or a Head-Up Display (HUD).
- More intricate uses of stencil buffers include rendering shadows, reflections, or contours in multi-pass rendering techniques.
- The stencil buffer assigns integer information, typically encoded at the bit level, to each pixel on the screen.
- During rendering, data from the stencil buffer is utilized to execute specific tasks on corresponding pixels.

Typically, the stencil buffer stores a single bit of information for each pixel (1 indicating that the pixel should be drawn, 0 indicating it can be skipped), although more complex functions can be achieved using this technique.

## 5.4 Clipping

When triangles from a mesh intersect screen boundaries, they may only be partially visible. Clipping addresses this by truncating and removing portions of graphics primitives to ensure they are fully contained within the screen.

Clipping occurs after the projection transform but before normalization (division by $w$). Consequently, it operates within a space known as Clipping Coordinates. In 3D, clipping is conducted against the viewing frustum, the space that defines what is visible to the viewer within the scene.

### 5.4.1 Half spaces

Let's revisit the equation of a plane:

$$n_x x + n_y y + n_z z + d = 0$$

Here, $n_x, n_y, n_z$ represent the components of the normal vector to the plane and the constant term $d$ defines the distance from the origin (zero if the plane passes through it). When expressed as an inequality, the equation divides space into two regions known as half-spaces:

$$n_x x + n_y y + n_z z + d > 0$$

Using homogeneous coordinates, a plane can be identified with a four-component vector $n$. Thus, the plane equation becomes a scalar product between the homogeneous coordinates of the point and the vector representing the plane:

$$\begin{cases} n = (n_x, x_y, n_z, d) \\ p = (x, y, z, 1) \\ n \cdot p = 0 \end{cases}$$

A point $p$ lies in either of the two half-spaces depending on the sign of the scalar product between $p$ and $n$. Additionally, the result of the product is the signed distance of the point from the plane dividing the two half-spaces.

As a frustum is a convex solid, it can be defined as the intersection of the six half-spaces corresponding to its six faces.

Clipping is performed in clipping space, where coordinates are considered to be inside the frustum if they fall within certain ranges: between $-1$ and 1 for $x$ and $y$, and between 0 and 1 for $z$. The six vectors representing the six faces of the frustum are as follows:

$$\begin{cases} n_l = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \\ n_r = \begin{bmatrix} -1 & 0 & 0 & 1 \end{bmatrix} \\ n_b = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \\ n_t = \begin{bmatrix} 0 & -1 & 0 & 1 \end{bmatrix} \\ n_n = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \\ n_f = \begin{bmatrix} 0 & 0 & -1 & 1 \end{bmatrix} \end{cases}$$

### 5.4.2 Points

To ascertain whether a point lies within the frustum, we conduct scalar products between its homogeneous coordinates and the six normal vectors. Specifically, the point resides inside the frustum if all these products yield positive results. A clipping algorithm can then eliminate a point $p$ if the product with at least one of the normal vectors to the frustum is negative:

$$p \cdot n_v > 0$$

### 5.4.3 Triangles

Dealing with triangles adds complexity to the clipping process. Let's focus on a side of the frustum with a normal vector $n_v$. We compute the distances from the plane ($d_1, d_2$, and $d_3$) by performing scalar products of the three coordinates of vertices $p_1, p_2$, and $p_3$ with the normal vector $n_v$.

We classify a side as either trivial reject or trivial accept based on the signs of the three distances. If all distances have the same sign, we either reject or accept the side. In case of rejection (all negative distances), the algorithm stops as the triangle lies entirely outside the frustum. For acceptance (all positive distances), the iteration continues with the next plane:

$$\begin{cases} d_1 = p_1 \cdot n_v > 0 \\ d_2 = p_2 \cdot n_v > 0 \\ d_3 = p_3 \cdot n_v > 0 \end{cases}$$

If two points are outside the frustum, e.g., $p_2$ and $p_3$, we compute two intersections $p_2'$ and $p_3'$ of the segments connecting them to $p_1$ with the plane using interpolation. The distances from the plane $d_2$ and $d_3$ are used as interpolation coefficients, and the two vertices $p_2$ and $p_3$ are replaced by $p_2'$ and $p_3'$:

$$\begin{cases} d_1 = p_1 \cdot n_v > 0 \\ d_2 = p_2 \cdot n_v < 0 \\ d_3 = p_3 \cdot n_v < 0 \end{cases}$$

If just one point is outside, e.g., $p_3$, we compute two intersections $p_3'$ and $p_3''$ on the segments that connect it to $p_1$ and $p_2$. In this case, two triangles are produced, and usually, the choice is arbitrary.

$$\begin{cases} d_1 = p_1 \cdot n_v > 0 \\ d_2 = p_2 \cdot n_v > 0 \\ d_3 = p_3 \cdot n_v < 0 \end{cases}$$

If the triangle is not rejected, it is clipped against the next plane. If two triangles have been produced, they are considered separately.

The algorithm continues until all triangles have been checked with all sides of the frustum, or when all triangles have been rejected. While the algorithm is simple, it can generate numerous triangles since they can potentially double at every check. This has implications on the required data structure to store the triangles, as it must accommodate a variable number of figures. Additionally, computing intersections is complex as it must consider all parameters assigned to vertices, such as normal vectors, colors, and UV coordinates.

### 5.4.4 Remarks

Understanding the process of clipping, although typically handled transparently by drivers, provides insight into an important step in rendering graphics on screen. Furthermore, the techniques discussed can be adapted to address various intersection problems that may arise in an application.

While clipping is usually automated, advanced users can manipulate the clipping distance of triangles to achieve intriguing effects. However, delving into such advanced capabilities is beyond the scope of this course.

## 5.5 Screen synchronization update

The CPU, GPU, and screen update process operate at varying and independent speeds. If a program is not properly written, this discrepancy can lead to flickering in animations.

Monitors and display devices construct images by sequentially updating their pixels in a predetermined order, typically scanning horizontally from left to right and top to bottom. Flickering manifests as breaks in the animation, known as tearing, occurring because the graphics adapter reads video memory while the program is still in the process of composing the image.

Each graphics adapter issues an interrupt signal, often referred to as Vsync, to the processor upon completing the screen tracing. Applications can intercept this interrupt and initiate background updates only upon receiving the Vsync signal. This synchronization ensures smoother animation rendering and reduces flickering issues.

### 5.5.1 Double buffering

In double buffering, the video memory consists of two frame buffers: the front buffer and the back buffer. While the video adapter displays the content of one buffer, the application can simultaneously compose the image in the other.

Initially, the application operates on one buffer (the back buffer), while the video adapter displays the other one (the front buffer).

Once the new frame is composed, the application awaits the Vsync signal from the monitor. Upon receiving the signal, it swaps the front buffer with the back buffer. Subsequently, the application begins composing the new frame while the monitor displays the just-finished one. This process ensures smoother animation rendering by synchronizing the buffer swapping with the monitor's refresh cycle.

### 5.5.2 Triple buffering

While double buffering effectively addresses certain issues, it also presents drawbacks. For instance, the application must halt image composition upon completing a screen and wait for the Vsync interrupt to resume. This synchronization with the monitor's refresh rate limits the frame rate and can lead to application locks, reducing CPU and GPU utilization.

Triple Buffering resolves these limitations by introducing complete independence between the application and presentation processes. Initially, the application operates on one frame buffer while the system displays another. Upon finishing composing a screen, the application immediately begins working on the next one in the buffer currently not used by the presentation.

Upon receiving the next Vsync signal, the presentation displays the last fully composed frame (the one where the application is currently not working). The application can switch between the two back buffers not currently shown as many times as necessary while awaiting the Vsync. If a frame is completed before the Vsync is received, the previously generated screen is skipped and never displayed. This process, known as frame skipping, enables achieving frame rates higher than the display's rate by automatically discarding unused frames.

Triple buffering ensures smooth animations and prevents the tearing effect by swapping buffers at the Vsync. However, it has drawbacks, including increased memory requirements (at least three frame buffers are needed) and potentially wasteful resource utilization by both the CPU and GPU on frames that will be discarded.

# CHAPTER 6

# Rendering

## 6.1 Introduction

Rendering is the process of creating realistic 3D images with accurate colors. To achieve realistic images with filled surfaces rather than just wireframes, it's essential to accurately emulate the reflection of light. Rendering involves replicating the effects of illumination by defining the light sources within the virtual environment and specifying the surface properties of the objects within the 3D world.

Light sources are components of the scene from which illumination originates. These light sources emit various frequencies of the spectrum, and objects in the scene reflect a portion of this emitted light. Photons emitted by the light sources interact with objects, bouncing off them, with some eventually reaching the viewpoint (i.e., the camera). During rendering, the intensity and color of these photons are computed. The amount of light reflected depends on both the input (incoming light) and output (reflected light) directions, as photons can bounce off objects in numerous ways before reaching the viewer.

**Radiance** Radiance refers to the energy emitted by a surface in all directions during a specific time interval, measured in Joules ($J$). Power, measured in Watts ($W$), represents the instantaneous emission of light energy by a surface at a given moment.

**Definition** (*Irradiance*)**.** Irradiance, denoted by $E$, quantifies the fraction of power emitted by a point on a surface within a specific time interval and is measured in $W/m^2$.

**Definition** (*Radiance*)**.** Radiance, denoted by $L$, measures the energy emitted from a point on a surface in a particular direction at a specific time instant, expressed in $W/(m^2 \cdot sr)$.

Here, $sr$ represents steradians, the unit of solid-angle measure. Using solid angles allows radiance to remain independent of the distance at which an object is observed.

Most light sensors, including human eyes and cameras, produce readings proportional to radiance. In rendering, the radiance received at each point on the projection plane (i.e., each pixel on the screen) is determined based on the direction of the corresponding projection ray.

## 6.2 Rendering equation

Light emitted by sources is characterized by its radiance. The material comprising an object's surface determines the intensity of light reflected in a particular direction.

In practice, the surface's microscopic structure determines the direction of light bounces, which can vary significantly between materials. The surface properties of an object can be described by a function known as the bidirectional reflectance distribution function (BRDF).

### 6.2.1 Bidirectional reflectance distribution function

The BRDF function takes as inputs the incoming ($\omega_i$) and outgoing ($\omega_r$) light directions, both represented as unit vectors. It quantifies the amount of irradiance from the incoming angle that is reflected towards the outgoing angle and is measured in steradians per unit solid angle ($sr^{-1}$):

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\omega_i, \omega_r)$$

This allows for the consideration of various reflection angles for incoming radiance, depending on the surface material.

A good BRDF should satisfy two main properties: reciprocity and energy conservation.

**Reciprocity** Reciprocity implies that swapping the incoming and outgoing directions doesn't change the function's value (hence the term bidirectional):

$$f_r(\omega_i, \omega_r) = f_r(\omega_r, \omega_i)$$

**Energy conservation** Energy conservation dictates that the BRDF cannot increase the total irradiance leaving a point on a surface:

$$\int f_r(\omega_i, \omega_r) \cos\theta_r \, d\omega_r \leq 1$$

The integral may be less than one if the point absorbs energy.

**BRDF functions** Various approximations to common BRDF functions have been proposed in the literature. Some of these approximations provide satisfactory results during rendering, even if they don't fully adhere to the aforementioned properties.

### 6.2.2 The equation

The BRDF facilitates the connection between the irradiance in all directions for every point on the objects within a scene. This connection is formalized through the rendering equation:

$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y) \, dy$$

In this equation, the following components are present:

- $L(x, \omega_r)$ determines the radiance of each point $x$ of any object in the 3D world, for any output direction $\omega$.

- $L_e(x, \omega_r)$ accounts for the light that the object at $x$ emits in direction $\omega$. This parameter mainly characterizes light sources (e.g., bulbs or neon) and serves as the initial injection of photons into the scene.
- The integral accounts for the light that hits the considered point $x$ from all the points $y$ of the surfaces (including the light sources) of all the objects and lights in the scene. It also includes other points of the same object to allow for the computation of effects such as self-shadowing or self-reflection. It is composed of the following terms:
  - $L(y, \overrightarrow{yx})$: for each object, the equation considers the radiance emitted toward point $x$. Here $\overrightarrow{yx}$ represents the direction of the line that connects point $y$ to point $x$.
  - $f_r(x, \overrightarrow{yx}, \omega_r)$ is the BRDF of the material of the object at point $x$. Since the materials of objects might change over the surface, the BRDF depends also on the position of the considered point $x$. The input angle corresponds to the direction of the incoming light, oriented along the segment that connects $y$ to $x$. The output angle corresponds to the direction from which the output radiance is being computed, represented by $\omega_r$ on the left-hand side of the equation.
  - $G(x, y)$ encodes the geometric relation between points $x$ and $y$. This is necessary because the angle between the surfaces has an impact on both the light emitted and reflected. It considers both the relative orientation and the distance of the two points and is defined as follows:

    $$G(x, y) = \frac{\cos\theta_x \cos\theta_y}{r_{xy}^2}$$

    The two cosine terms account for the angle relative to the respective normal vectors, and $r_{xy}^2$ represents the squared distance of the two points.
  - $V(x, y)$: finally, term $V(x, y)$ considers the visibility between two points $x$ and $y$: $V(x, y) = 1$ if the two points can see each other, and $V(x, y) = 0$ if point $y$ is hidden by some other objects in between. Term $V(x, y)$ allows for the computation of shadows and ensures that in each input direction, at most a single object is considered.

The term $L(x, \omega)$ represents the unknown quantity in the equation. As it appears on both sides of the expression, the rendering equation is mathematically classified as an integral equation of the second kind. This classification is denoted using mathematical notation as follows:

$$\varphi(x) = f(x) + \lambda \int_a^b K(x, t)\varphi(t)\, dt$$

**Colors** The rendering equation is applied for every wavelength $\lambda$ of light, typically involving repetition for the three different RGB channels. Geometric and visibility terms remain constant across wavelengths and are unique for each pair of points.

Light sources are associated with RGB values representing the photons emitted for each of the three main frequencies. Objects are characterized by a BRDF with distinct parameters for each of the three primary colors. Three images are generated independently, each considering either the red, green, or blue components individually. Subsequently, the three colors are combined to produce the final output image.

Due to the separation of color components, the interaction between light and material colors results in outcomes that are not immediately intuitive.

**Extension** The proposed rendering equation accurately computes various effects such as reflections, shadows, matte, and glossy objects. However, it falls short in simulating other effects like participating media (e.g., rendering of gases and fumes) or transparent objects like glass or water. To account for these materials, extensions to the BRDF and rendering equations have been made.

One extension involves considering transmitted light, i.e., transparency, which is achieved by defining the bidirectional transmittance distribution function (BTDF). The BTDF has a similar definition to the BRDF but is used in the opposite direction:

$$f_t(\theta_i, \phi_i, \theta_r, \phi_r) = f_t(\omega_i, \omega_r)$$

Since the angles for the BRDF and BTDF typically do not overlap, they are often combined into a single function called the bidirectional scattering distribution function (BSDF). The rendering equations need to be updated to incorporate both the BRDF and the BTDF. Here, $x'$ denotes the point on the other side of the surface (assuming it is unique) from which light is directed toward point $x$.

$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y) + \\ + L(y, \overrightarrow{yx}) f_t(x', \overrightarrow{yx}, \omega_r) G(x', y) V(x', y)\, dy$$

In certain materials, light can bounce inside the object and exit from another point. This phenomenon is known as subsurface scattering, observed in materials like human skin and marble. For such cases, a more complex function called the bidirectional surface reflectance distribution function (BSSRDF) is utilized. The BSSRDF incorporates an extra parameter $x'$ to account for the point on the surface from which light enters at angle $\omega_i$. The rendering equation now integrates over all points of an object to consider the quantity of light that can enter from a given position and direction and exit from a specific position and direction:

$$L(x, \omega_r) = L_e(x, \omega_r) + \iint L(y, \overrightarrow{yx}) f_{ss}(x, \overrightarrow{yx}, x', \omega_r) G(x', y) V(x', y)\, dx' dy$$

**Solution** Solving the rendering equations is inherently challenging and often necessitates complex discretization techniques. In the subsequent sections, we will explore straightforward approximations to the rendering equation that yield satisfactory results with manageable complexity. Some of these techniques are supported in Vulkan through its specific features.

### 6.2.3 Lights basics

The simplest approximations to the rendering equation categorize light sources into direct and indirect sources.

Direct sources emanate from specific positions and directions, such as lamps, studio spotlights, or the sun in outdoor scenes.

Indirect sources encompass all other forms of illumination, primarily caused by light bounces and reflections among surfaces. Photographers often utilize these sources to introduce soft lighting in their images.

Images illuminated solely by direct sources may appear overly dark and lack realism. Points not struck by any light source would appear pitch black. While this setup is unrealistic, it is often the simplest assumption made to simplify real-time rendering equation solutions in computer graphics.

Projected shadows are formed by the obstruction of direct light sources and can be approximated using specific techniques.

Indirect lighting enhances realism by accounting for light that bounces off other surfaces. However, achieving accurate computations for indirect lighting demands considerable effort and is challenging to implement in real-time scenarios. Nonetheless, various approximations exist to recreate indirect lighting effects.

In many cases, the light contribution for a single point and direction is computed offline and stored in an image-based structure. This precomputed data is later used to perform approximations for indirect lighting during real-time rendering.

## 6.3 Pipelines

To convert a dataset depicting a mesh into a visible image on the screen, a series of operations must be carried out. This series, akin to the classical pattern designed for leveraging parallel execution of various steps on data, is termed a pipeline.

A pipeline is a framework in which a continuous flow of data undergoes processing through multiple sequential steps. As one element progresses to the second step, a new element can begin processing in parallel with the preceding one. In Vulkan, as well as in computer graphics in general, generating an image on screen from a primitive description involves a series of steps that can be structured into a pipeline.

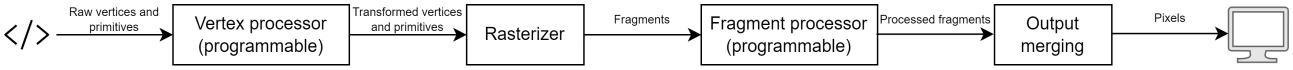



Figure 6.1: Rendering pipeline

The actions performed at each stage of the pipeline can either be predefined by the system or customized by the user. For historical reasons, algorithms running in the programmable stages of the pipeline are referred to as shaders.

### 6.3.1 Taxonomy

Various types of pipelines have been defined to handle the generation of 3D images, each serving specific purposes and equipped with distinct sets of fixed functions, input and output descriptions, and programmable stages. Creating a pipeline involves configuring all the parameters required by its fixed functions and connecting it with shaders responsible for the user-defined aspects. In the latest versions of Vulkan, up to four types of pipelines are supported:

- Graphic pipelines.
- Ray-tracing pipelines.
- Mesh Shading pipelines.
- Compute pipelines.

The first three are primarily designed for rendering 3D meshes, while the compute pipeline is utilized for general computation purposes, such as general-purpose GPU (GPGPU) tasks. Ray tracing and Mesh Shading pipelines may still have limited support as they are relatively recent developments. Several techniques have been introduced to approximate the rendering equation, including:

- Scan-line rendering.
- Ray casting.
- Ray tracing.
- Radiosity.
- Monte Carlo techniques.

These techniques are closely associated with the types of pipelines that support them.

**Scan-line rendering** Scan-line rendering is the simplest approximation of the rendering equations. It treats light sources and objects separately within a scene, where objects reflect light but do not illuminate other objects. Consequently, no projected shadows or indirect lighting effects are produced.

In this technique, only the points currently visible to the camera are considered, with lights characterized by having only the emission term in the rendering equation, which can vary in position and direction. The vertices of triangles belonging to a mesh are projected onto the screen to determine their corresponding hardware coordinates. Subsequently, all pixels belonging to a triangle are enumerated, and for each pixel, the rendering equation is solved.

Inter-reflection between objects is not accounted for, simplifying the integral to a summation over all light sources. The geometric term is typically incorporated into the Bidirectional Reflectance Distribution Function (BRDF):

$$L(x, \omega_r) = L_e(x, \omega_r) + \sum_l L_e(l, \overrightarrow{lx}) f_r, (x, \overrightarrow{lx}, \omega_r)$$

Visibility in scan-line rendering is determined solely from the perspective of the observer, typically through the z-buffer algorithm. Because the visibility term ($V(\cdot)$) of the rendering equation is not considered for lights, scan-line rendering does not generate projected shadows. Additionally, it does not account for light emitted by other objects in the scene, nor does it produce effects such as reflection, refraction, or indirect illumination.

However, despite these limitations, scan-line rendering does allow for the incorporation of various types of Bidirectional Reflectance Distribution Function (BRDF) functions. These BRDF functions can provide detailed descriptions of the materials composing the objects in the scene, enhancing the visual realism despite the simplifications in other aspects of the rendering process.

## 6.4 Vulkan pipeline

Vulkan facilitates scan-line rendering through a designated pipeline known as the graphics pipeline. Let's delve into its stages and their respective functions. As outlined in the Vulkan documentation, the graphics pipeline comprises the following structure:

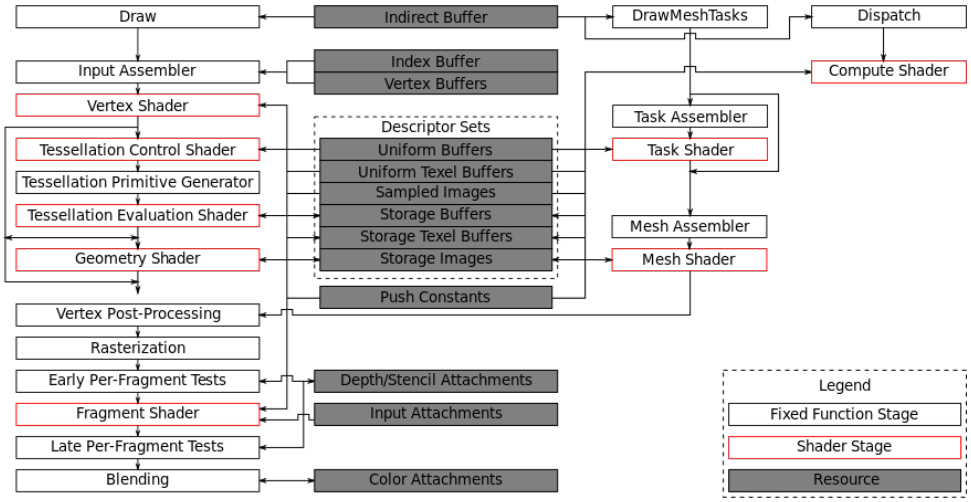



Figure 6.2: Vulkan pipeline

The graphics pipeline in Vulkan accommodates up to five distinct types of shaders, which define the functionalities of the programmable stages within the pipeline. Typically, only the initial and final stages are mandatory. In the majority of scenarios, solely the Vertex and fragment shaders are essential to produce an image. The tessellation and geometry stages are discretionary; their absence leads the pipeline to disregard these functions and proceed to the subsequent stages.

**Stages** To draw a triangle using the Vulkan pipeline, you would typically follow these steps:

1. *Input assembler*: whenever a draw command is issued, Vulkan creates the vertices by combining all parameters that describe them. If several instances of the same object are used, vertices are replicated as many times as required. This stage also decides if we are drawing points, lines, or triangles, using lists or other strip-based approaches.

2. *Vertex shader*: vertex shaders are executed to perform operations on each vertex. These operations include transforming local coordinates to clipping coordinates by multiplying vertex positions with the corresponding World-View-Projection (WVP) matrix, or computing colors and other values associated with vertices, which will be used in later stages of the process.

3. *Tessellation*: tessellation is used to increase the resolution of an object.

4. *Geometry shader*: geometry shaders can remove or add primitives to the stream, starting from the previously generated elements. In principle, they could perform the same tasks as the tessellation stages. However, due to their generality, implementing these functions in geometry shaders would require more complex code and result in slower performance.

5. *Rasterization*: rasterization determines the pixels in the frame-buffer occupied by each primitive. These are called fragments, not pixels, since a single pixel on the screen can be computed by merging several fragments to increase the quality of the final image (the so-called anti-aliasing). During these stages, the division by $w$ to transform clipping coordinates into normalized screen coordinates is also performed.

   For example, if the considered basic primitive corresponds to a triangle, the rasterization stage will generate at least a fragment for all the pixels connecting the screen projections

of its three vertices. Fragments are usually generated per line, left to right, with respect to the corresponding triangle. This feature is what motivates the scan-line rendering name of this technique.

6. *Fragment shader*: the final color of each fragment is determined by a user-defined function contained in the fragment shader. This section will use either physically based models or other artistic techniques to produce either realistic or effective images. In other words, it will compute the approximate solution of the rendering equation for the considered pixel.

7. *Color blending*: finally, the computed colors might either replace the ones already present in the same position, or be combined with them. The latter can be used to implement transparency or other blending effects.

Several important actions occur in the final fixed sections of the pipeline. In particular, this is where a few functionalities previously introduced take place: primitives clipping and back-face culling in vertex post-processing. Depth testing (z-buffer) and stencil operations in early per-fragment tests.

The pseudocode of a scan-line rendering algorithm is as follows:

**Algorithm 1** Scan-line rendering algorithm

```
1: for each mesh object A in the scene do
2:     Determine the screen coordinates of each vertex of t
3:     for each visible (passes the back-face culling and clipping) triangle t of A do
4:         for each pixel x of t on screen do
5:             if pixel x is visible (passes Z-buffer test) then
6:                 Set the pixel color C = L_e(x, ω_r)
7:                 for each light l in the scene do
8:                     Set C = C + L(l, lx) * fr(x, lx, ω_r)
9:                 end for
10:            end if
11:        end for
12:    end for
13: end for
```

### 6.4.1 Shaders

In Vulkan, shaders are defined by SPIR-V code blocks. SPIR stands for Standard Portable Intermediate Representation, and it is a binary format for specifying instructions that a GPU can run in a device-independent way.


Figure 6.3: Shaders

Every Vulkan driver converts the SPIR-V code into the binary instructions of their corresponding GPU. SPIR-V has been created with the goal of being efficiently converted into instructions for the most popular GPUs, so this process is usually not very expensive from a computational point of view.

Shaders are written in high-level languages, such as:

- GLSL: OpenGL Shading Language.
- HLSL: High Level Shading Language (Microsoft DirectX).

At development time, the shaders are compiled from their original language to SPIR-V. In general, depending on the shader type, the file containing its source code has a different extension. Compiled shaders into SPIR-V files have instead the `spv` extension.

Shaders can be compiled using the glslc tool, which is included in the Vulkan SDK.

## 6.5 Ray casting

Ray casting expands upon scan-line rendering by calculating the visibility function for all point-triangle/light source pairs in the scene. It facilitates the inclusion of projected shadows. The visibility function is determined by tracing a ray from each considered point to every light source. If the ray intersects an object, the light is obstructed, and its contribution to the rendering equation is disregarded. Below is the pseudocode for a ray casting rendering algorithm:

**Algorithm 2** Ray casting rendering algorithm

```
1: for each object A in the scene do
2:     for each visible (pass back-face culling and clipping) triangle t of A do
3:         for each pixel x of t on screen do
4:             if pixel x is visible (pass Z-buffer test) then
5:                 Set the pixel color C = 0
6:                 for each light l in the scene do
7:                     if light l is not occluded (ray-casting) then
8:                         Set C = C + L(l, lx) · fr(x, l_x, ω_r)
9:                     end if
10:                end for
11:            end if
12:        end for
13:    end for
14: end for
```

One common real-time technique for ray casting is the utilization of a Shadow Map. This map is essentially an image generated from the perspective of the light source, where each pixel's color represents the distance from the point to the light source. This information is then employed to ascertain whether a pixel is illuminated by the light or obscured.

The shader responsible for rendering computes the distance from the light source to the pixel being rendered and contrasts it with the corresponding distance stored in the shadow map. If the computed distance is greater, indicating that the pixel is farther from the light source than the corresponding point in the shadow map, the light contribution is not considered.

## 6.6 Ray tracing

Ray tracing involves considering the light emitted by other objects in two distinct directions: mirror reflection and refraction for transparent objects. This is mathematically represented as

follows:

$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L(l,\overrightarrow{lx}) f_r(x,\overrightarrow{lx},\omega_r)V(x,l) + L(r,\overrightarrow{rx}) f_{r'}(x,\overrightarrow{rx},\omega_r)V(x,r) + \\ + L(t,\overrightarrow{tx'}) f_{t'}(x',\overrightarrow{tx'},\omega_r)V(x,t)$$

In this equation, $L(x,\omega_r)$ represents the radiance at point $x$ in direction $\omega_r$. The first term $L_e(x,\omega_r)$ accounts for emitted radiance. The summation term involves contributions from other objects: $L(l,\overrightarrow{lx})$ denotes radiance from objects in the scene, $f_r(x,\overrightarrow{lx},\omega_r)$ represents the reflection function for mirror reflection, $f_{r'}(x,\overrightarrow{rx},\omega_r)$ accounts for mirror reflection, and $f_{t'}(x',\overrightarrow{tx'},\omega_r)$ represents the refraction function. $V(x,l)$, $V(x,r)$, and $V(x,t)$ are visibility functions indicating whether the light from a given direction is visible from the viewpoint. In this equation, the second term of the summation handles mirror reflection, while the third term considers refraction.

**Reflection** In the context of reflection, this direction corresponds to the mirror direction: the angle of reflection equals the angle of incidence but on the opposite side relative to the surface normal at the point of impact. This facilitates the recreation of realistic perfect (mirror-like) reflections. Specifically, at each point $x$, the algorithm searches for points $r$ on all objects within the scene along the mirror direction $\overrightarrow{rx}$ and chooses the one closest to $x$.

**Refraction** In the context of refraction, the algorithm emulates the physical properties of objects by incorporating the material's index of refraction to determine the angle of the refracted ray. Here, for each point $x$, the algorithm initially identifies the exit point $x'$ from which the ray will on the opposing side by considering the varying refractive indices of the materials separated by the surface. Subsequently, it searches for points $t$ on all objects along the direction $\overrightarrow{tx'}$.

**Ray casting** The algorithm relies on a ray-casting procedure to determine the colors visible from a given point in space $x$ and direction $\omega$. This procedure identifies the closest object to point $y$ in the specified direction $\omega$ and utilizes the approximated rendering equation to compute $L(y,\omega)$. Initially, the algorithm iterates over each point on the projection plane (each pixel of the generated image) in the direction of the projection ray, applying the ray-casting procedure to each.

To incorporate reflection and refraction into each pixel, the procedure is invoked recursively with the computed points and directions. This recursion continues up to a defined number of bounces, known as the ray depth.

Light sources remain distinct from objects, and visibility for light sources is also accounted for. In this case, ray tracing determines whether a light source is visible or not. This determination is made by the first term of the summation. If the reflected ray does not intersect with any other object in the direction of the light, then the light source is considered visible. However, if the ray intersects with another object, the light is deemed obstructed and excluded from the rendering equation.

### 6.6.1 Ray-tracing pipeline

Frameworks such as Vulkan and DirectX offer dedicated pipelines to facilitate real-time ray tracing, provided that compatible GPUs are accessible. In Vulkan, this pipeline is referred to as the Ray Tracing pipeline.

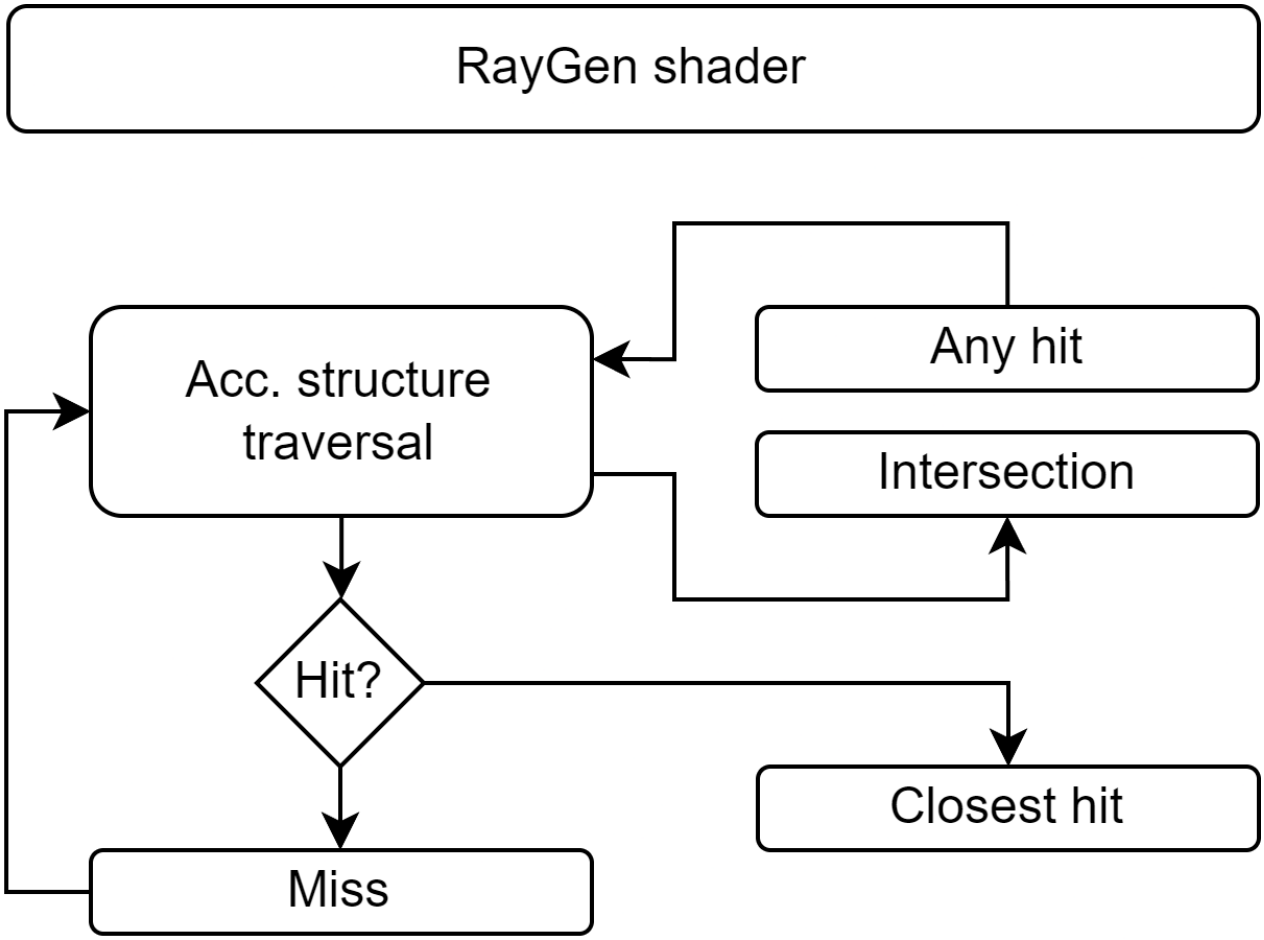



Figure 6.4: Shaders

The ray-tracing pipeline generates images by casting rays from the pixels on the screen, unlike the traditional graphics pipeline driven by triangles and vertices. For each fragment on the screen, a ray is projected into the scene and intersects with all the triangles of all the meshes in the 3D environment. Only the closest intersection to the viewer is considered.

To compute the color of each fragment accurately, additional rays are traced to reproduce reflections and refractions (transparencies). In the presented ray-tracing rendering technique, these rays typically include the perfect reflection ray, the refraction ray, and the rays connecting the point to the light sources.

Determining intersections with all triangles in the scene is a computationally intensive task. Without proper optimization, it can have a complexity of $O(n)$ in the number of triangles. Special acceleration structures provided by the user are necessary to address this complexity. Due to time constraints, a detailed discussion of these structures cannot be provided here.

The ray-tracing pipeline requires five shaders to compute and handle all triangle-ray intersections.

**RayGen shader** The RayGen shader is executed for each output fragment of the images, responsible for determining the starting point and direction of the corresponding ray in the scene. Typically, it considers each pixel on the screen and casts a ray from the camera's viewpoint in the corresponding direction.

However, users are not limited to considering a planar projection plane with equally-spaced pixels. Cylindrical and spherical projections can be easily implemented with minor adjustments

to the RayGen procedure.

**Intersection and closest hit** The intersection shader enables the implementation of custom ray-triangle intersection procedures. On the other hand, the closest hit shader is invoked on the point closest to the viewer. Its primary function is to compute the color of the point by approximating the rendering equation, akin to a fragment shader in the graphics pipeline. Additionally, it can recursively cast other rays to handle effects like reflections and refractions.

**Any hit and miss** The any hit shader serves to filter out intersections that should not be considered, such as those involving partially transparent objects. It provides a mechanism to handle such cases effectively.

On the other hand, the miss shader is invoked when the ray does not hit any object in the scene. Typically, it is utilized to render a background that appears behind all other objects in the scene.

**Structure traversal** The fixed part of the ray-tracing pipeline governs the traversal of acceleration structures and the determination of the closest hit. While this approach is powerful, it demands significant computational resources. Only the most advanced GPUs available today can provide satisfactory performance for real-time ray tracing.

### 6.6.2 Algorithm

The pseudo-code of a ray tracing rendering algorithm is the following:

**Algorithm 3** Ray tracing rendering algorithm

```
for each pixel x on screen do
    Compute color casting a ray from x according the projection
end for
```

The ray-casting procedure is the heart of the technique:

**Algorithm 4** Ray casting procedure

```
Determine the point q of the closest object with respect to the ray
Set the pixel color C = 0
for each light l in the scene do
    if light l is not occluded (ray-casting) then
        Set C = C + contribution of light l to q
    end if
end for
Set C = C + reflection contribution (recursion)
Set C = C + refraction contribution (recursion)
```

Indeed, the number of traced rays can potentially double at each step in ray tracing, leading to significantly increased rendering times. Additionally, it necessitates the computation of the closest intersection using acceleration structures instead of relying on the Z-buffer, as in traditional rasterization techniques.

Ray tracing offers the capability to incorporate mirror reflection and transparency with refraction. This enables realistic rendering of materials such as glass, fluids, and shiny metals, among others.

However, it's important to note that ray tracing has limitations. It is not inherently capable of simulating indirect lighting or handling glossy reflections, which restricts the level of realism achievable with this technique.

## 6.7 Radiosity

Radiosity offers a distinct simplification of the rendering equation, focusing on materials with constant Bidirectional Reflectance Distribution Function (BRDF). This simplification reduces the unknowns in the rendering equation to one variable per point on the objects, as reflection is independent of the direction from which it is observed. This variable is termed the radiosity of the object, which corresponds to the output counterparts of irradiance:

$$L(x) = L_e(x) + \rho_x \int L_(y) G(x,y) V(x,y)\, dy$$

In this equation, the surfaces of the objects are divided into patches, each with its own unknown radiosity value. Light sources are implemented as patches that emit radiosity ($L_e(\cdot)$ term). The rendering equation transforms into a system of linear equations, which can be solved iteratively:

$$L(x_i) = L_e(x_i) + \rho_{x_i} \sum_{y_j} L(y_j) G(x_i, y_j) V(x_i, y_j) = L_e + \sum_{y_j} L(y_j) R(x_i, y_j)$$

In matrix notation, the vector $L$ contains one element (per color frequency) per patch, representing its radiosity. Matrix $R$ includes visibility, constant BRDF, and geometric relations between any two patches in the scene. The solutions of these equations are then interpolated to generate a view of the scene.

### 6.7.1 Algorithm

The pseudo-code of a radiosity rendering algorithm is the following:

**Algorithm 5** Radiosity rendering algorithm

1: Discretize the scene, and compute matrix $R$
2: Compute the solution of equation $L = L_e + R \cdot L$
3: Render the scene using either scan-line or ray tracing
4: Interpolate $L$ to obtain the color for each pixel

Indeed, the most time-consuming steps are the first two. However, once they have been computed, the scene can be rendered very quickly from any point of view. In most cases, the solution of the equation can be computed with a fixed-point iteration, starting from $L = 0$, and refining its value at every iteration.

**Algorithm 6** Rendering equation algorithm

1: $L_{old} = L_{new} = 0$
2: **repeat**
3: $\quad L_{old} = L_{new}$
4: $\quad L_{new} = L_e + R \cdot L_{old}$
5: **until** $|L_{new} - L_{old}| >$ threshold

## 6.8 Montecarlo techniques

Photorealistic results in rendering can only be achieved by approximating the solution of the complete rendering equation. Due to its complexity, Monte Carlo techniques are commonly employed: the integral is computed by averaging several random points and directions chosen from the equations. Many alternative approaches are possible, and each advanced rendering engine typically exploits one of them.

Numerous techniques extend ray tracing: instead of sending a single ray in the mirror direction, a sampling of the most probable output directions is considered (importance sampling). A ray is then traced for each selected direction, and the radiance is computed using the Bidirectional Reflectance Distribution Function (BRDF) of the considered material.

Photon mapping, on the other hand, emulates the movements of photons in the scene, taking into account bounces, focalizations, and other advanced phenomena such as caustics.

Due to the inherent randomness in these techniques, Monte Carlo-based rendering algorithms tend to produce noisy images. This noise can only be reduced by increasing the number of rays (and consequently, the rendering time).

## 6.9 Mesh shader pipeline

The mesh shader pipeline represents a generalized version of the graphics pipeline where there's no initial fixed part, and mesh generation can be entirely handled by shaders.

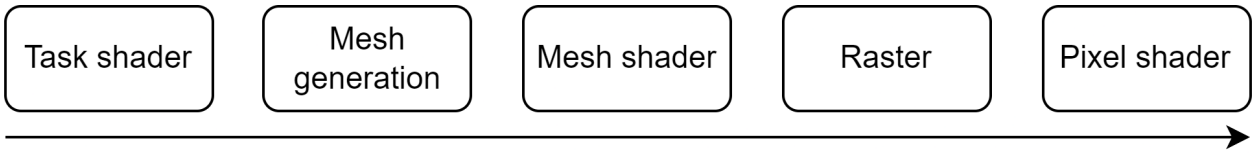



Figure 6.5: Mesh Shader pipeline

Mesh shaders compute indexed triangle lists, providing a set of vertices and groups of three indices for each triangle. Vertices are typically computed in normalized screen coordinates to simplify rasterization. However, there are limitations on the number of vertices and triangles that a Mesh shader can generate. To overcome this limitation, each object is divided into many patches called meshlets.

The optional Task shader plays a crucial role in this pipeline. Its purpose is to subdivide a larger mesh into smaller meshlets and control the corresponding mesh shader for generating all the required patches. This subdivision helps manage the computational load efficiently.

## 6.10 Compute pipeline

The compute pipeline serves a distinct purpose from rendering images, focusing on performing General-Purpose Graphics Processing Unit (GPGPU) computations. In this scenario, the application provides a single shader, known as the Compute Shader, which executes the desired operations.

Data required for computation is copied into buffers in GPU memory, if available. Compute shader executions are identified by a (up to) tridimensional index. This index, often referred to as a vector index, allows the shader to access data and determine the partition on which it can work. By utilizing this index, the shader can efficiently perform computations on the provided data.

## 6.11 OpenGL Shading Language

Most of the Bidirectional Reflectance Distribution Function (BRDF) functions, indirect light approximation, and light emission models are typically implemented using shaders written in the GLSL language.

To illustrate its features, let's consider an example that computes the Mandelbrot set, the most famous fractal, using vertex and fragment shaders.

### 6.11.1 Vertex shader

The vertex shader is defined as:

```
#version 450
layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;
layout(location = 0) in vec3 inPosition;
layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main () {
    gl_Position = ubo.vpMat * ubo.worldMat *
    vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img = inPosition.y * 2.5;
}
```

Shaders in GLSL follow classical conventions, where global variables and functions are declared in the main scope of the file. The entry point of the shader can be user-defined within the code, but it is typically the function `main ()`.

Vulkan requires support for at least a certain GLSL version. Therefore, a shader source code should begin with the `#version` directive, specifying at least version 4.5.

Comments in GLSL code follow the classical C notations. Blocks are denoted using curly brackets.

Variables in GLSL are typed, and their names follow the same conventions as in C (case-sensitive, allowing letters, numbers, and underscores, but cannot start with a number). Like in C, variables are local to the block they are defined in.

In addition to type and name, variables can be preceded by various qualifiers. GLSL supports a wide range of types, including scalar types (such as `int` and `float`), vector types (such as `vec2`, `vec3`, and `vec4`), and matrix types (such as `mat2`, `mat3`, and `mat4`).

**Vectors** In GLSL, types are available for containing vectors of two, three, or four components. Floating-point vectors are commonly used to represent colors and coordinates. Vector elements can be accessed individually using the dot syntax.

**Matrices** GLSL also defines matrix types, with the most commonly used being the $2\times2$, $3\times3$, or $4\times4$ matrices composed of single precision elements. GLSL uses column-major encoding for matrices, which means that elements are stored column-wise in memory. Elements of a matrix can be accessed using indices starting from zero in `[][]` brackets.

**GLM and GLSL** The types defined in the GLM library used in our C++ source code have been named to follow the equivalent GLSL conventions. This simplifies the interactions between the shaders and the application code.

By aligning the naming conventions between GLM types and GLSL types, it becomes easier to pass data between the application and shaders. This consistency enhances the overall development process and promotes code readability.

### 6.11.2 Fragment shader

The fragment shader is defined as:

```
#version 450
layout(location = 0) in float real;
layout(location = 1) in float img;
layout(location = 0) out vec4 outColor;
layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
    float time;
} gubo;

// The main procedure
void main() {
    float m_real = 0.0f, m_img = 0.0f, temp;
    int i;
    for(i = 0; i < 16; i++) {
        if(m_real * m_real + m_img * m_img > 4.0) {
            break;
        }
        temp = m_real * m_real - m_img * m_img + real;
        m_img = 2.0 * m_real * m_img + img;
        m_real = temp;
    }
```

```
    outColor =
    vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
    float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Please be aware that flow control statements behave differently on the GPU compared to the CPU. Due to the SIMD architecture of the GPU, which operates on multiple elements simultaneously, there are some important considerations:

- Both the if and else branches are always executed.
- In variable-length loops, all iterations are conditioned by the longest one in the batch being run concurrently.

Therefore, it's generally advisable to minimize the use of loops with variable iterations and conditional statements to optimize performance on the GPU.

### 6.11.3 Shader-pipeline communication

Communication between shaders and the pipeline takes place via global variables. `in` and `out`variables serve as interfaces with the programmable or configurable part of the pipeline. Additionally, communication with the fixed part of the pipeline utilizes predefined global variables, with `gl_Position` being a crucial one required in all graphic pipeline applications. However, numerous other global variables exist primarily to interface with different shader types.

For communication between shaders and the application, special types of external variables known as uniforms are employed. The most common among these are Uniform Variable Blocks. Each block resembles a C `typedef struct` element, possessing a tag, a name, and a collection of components. Elements within these blocks are accessed within a shader program akin to fields of a structure in C code.

## 6.12 Textures

Defining the appearance of realistic 3D objects requires meticulous detail, especially considering the varying material specifications across the surface. However, assigning distinct materials to numerous small triangles is impractical due to memory constraints. A common approach involves using tables to adjust shader parameters based on the surface's internal points. These tables, often referred to as textures or maps, contain not only the diffuse color obtained from images but also other properties to parameterize the Bidirectional Reflectance Distribution Function (BRDF) of a surface.

Textures come in various forms:

- *2D textures*: these define surface parameters for objects.
- *3D textures*: they extend to defining parameters within volumes.
- *1D textures*: these hold pre-computed function values or serve optimization purposes.
- *Cube maps textures*: used for defining parameters associated with directions in a 3D coordinate system, using a slightly different approach.

While images defining object surfaces are planar, the objects they are applied to exhibit complex non-planar 3D topologies with multiple sides. When applying 2D textures to 3D objects, a mapping relation is established, associating each surface point with a point on the texture. This mapping creates a correspondence between object points and texture pixels (texels). However, achieving a mapping that utilizes the entire texture space is challenging, often leaving some areas unused.

For meshes, the mapping procedure establishes a correspondence between surface triangles and corresponding triangles on the texture.

### 6.12.1 UV coordinates

Points on 2D textures are addressed using a Cartesian coordinate system, typically denoted by axes labeled $u$ and $v$. This coordinate system is commonly referred to as UV, mapping, or texture coordinates. In this convention, the $u$ and $v$ values range between 0 and 1 along the horizontal and vertical axes of the texture, respectively.

UV coordinates are assigned exclusively to the vertices of triangles, adding two additional parameters to their positions (and often to the direction of the normal vector). For internal points, UV coordinates are computed through interpolation, using techniques similar to those employed for normal vector directions.

Properly assigning UV coordinates to the model is a crucial but complex and often tedious task performed by 3D artists and modelers. It's worth noting that UV mapping does not necessarily need to be a bijection; the same part of the texture can be shared by several triangles on the 3D object. Skilled 3D artists can leverage this feature to enhance the quality of their models.

It's important to recognize that, similar to normal vector directions, two vertices belonging to different triangles may occupy the same position in space but have different UV coordinates.

**UV intervals** In many scenarios, the $u$ or $v$ values may extend beyond the $[0, 1]$ range. This occurrence can be interpreted in various ways, each of which can affect the final texture sampling differently. The four most common approaches for managing values that are negative or greater than one are:

- *Clamp*: extends the colors of the border to the texels that lie outside the $[0, 1]$ range.
- *Repeat*: utilizes only the fractional part of the UV coordinates, discarding their integer part, thus repeating the pattern.
- *Mirror*: repeats the texture pattern similar to the repeat strategy but flips it at each repetition.
- *Constant*: replaces the texture samples falling outside the $[0, 1]$ range with a default color $c_d$.

By employing various combinations of strategies for the $u$ and $v$ directions, numerous effects can be achieved.

### 6.12.2 Rasterization with textures

The initial step in supporting textures involves interpolating the UV coordinates alongside other vertex parameters, such as normal vectors. Subsequently, a lookup procedure is employed to

retrieve the corresponding texel from the texture. Finally, a per-pixel operation is executed to calculate the rendering equation for each pixel on the screen.

**Perspective interpolation** When perspective is utilized, conventional interpolation methods are inadequate due to the non-linearity of the projection. Applying traditional interpolation techniques to compute internal parameter values associated with triangles under perspective can result in apparent wobbling of straight lines, as depicted in the image below.

To mitigate such wobbling artifacts, applications adopt perspective-correct interpolation. This technique involves non-linear interpolation, where the interpolation behavior depends on the distance of the interpolated points from the projection plane.

### 6.12.3 Cube map textures

Cube-map textures consist of collections of six images, each corresponding to one of the six sides of a cube. These textures are indexed by a direction vector, specified with its $x$, $y$, and $z$ components. The components of the normal vector determine both the side of the cube (the base image) and the corresponding texel on that image.

In modern applications, cube map textures play a crucial role as they serve as building blocks for creating realistic rendering effects that emulate mirror reflection and transparency. Additionally, they are utilized to store image-based ambient lighting parameters.

### 6.12.4 Texture filtering

UV coordinates are represented as floating-point numbers, whereas texels are indexed by integer values. Additionally, the shape of a pixel on the screen may correspond to a complex polygon composed of several texels on the texture. To address these challenges, a set of techniques collectively known as filtering is employed.

When a texel is larger than the corresponding pixel on the screen, a magnification filtering problem arises. Conversely, when the pixel on the screen corresponds to multiple texels, a minification filtering problem occurs. The latter case is notably more challenging than the former.

**Magnification filters** Two magnification filters are typically defined:

- *Nearest pixel*: in this approach, the lookup procedure transforms the UV coordinates with respect to the texture size and returns the texel $p[i][j]$, considering only the integer part of the scaled coordinates. This method is fast, requiring just one texel read per pixel. However, it tends to produce blocky images.

$$x = u \cdot w \qquad i = \lfloor x \rfloor$$
$$y = v \cdot h \qquad j = \lfloor y \rfloor$$

- *Bilinear interpolation*: linear interpolation calculates the pixel color by interpolating from the values of its closest neighbors. While this technique produces smoother results, it necessitates four texture accesses and three interpolations. Although the term bilinear interpolation is commonly used, it's more appropriately described as linear interpolation. For 1D textures, interpolation occurs over one axis (involving 2 texels and 1 interpolation), while for 3D textures, it spans three dimensions (involving 8 texels and 7 interpolations). This interpolation becomes even more intricate in cube maps, especially for pixels along the borders or at the corners.

**Minification filters** The most commonly used minification filters include:

- *Nearest pixel.*
- *Bilinear interpolation.*
- *Mip-mapping*: this approach pre-computes a series of scaled versions of the texture (called levels), each with halved size. Each pixel in the inner level of a Mip-map corresponds to the average of a set of pixels in the original image. Depending on the texture-to-pixel ratio, the algorithm selects the closest image in the Mip-map, providing smoother results compared to basic interpolation.
- *Trilinear interpolation*: when images are angled with respect to the viewer, there may be a change of level within the mapping of a single figure. Trilinear filtering computes the required pixel twice, using the two closest levels of the Mip-map, and then interpolates between them for a smooth transition.
- *Rip-mapping*: a Rip-map encodes rectangular scaling of the original texture, addressing the blurring issue that occurs with traditional Mip-maps when surfaces are angled with respect to the viewer. However, this technique is rarely implemented due to its large memory requirements and inability to deal with angled triangles effectively.
- *Anisotropic*: when surfaces are angled with respect to the screen border, the pixel on the screen corresponds to a non-square and non-rectangular trapezoid over the texture. To address this issue, the Anisotropic Unconstrained Filter offers a solution. This algorithm approximates the precise area of texels corresponding to a pixel on the screen. It leverages Mip-maps and samples one of its levels following the direction of the pixel over the texels. Initially, the algorithm identifies the texture position of the four borders of the pixel. Subsequently, based on the size of the smallest side of the trapezoid, it determines the most suitable Mip-map level (one with a similar pixel-to-texel ratio). Then, it samples the selected Mip-map level along a line in the direction of the on-screen pixel. Typically, the number of samples is capped at a maximum value, often set to 8 or 16. While highly effective, this method incurs a significant overhead that may slow down rendering. Vulkan supports enabling Anisotropic filtering if it's supported by the graphics adapter.

Nearest pixel and bilinear interpolation behave similarly to magnification filters, but they yield poor results when the reduction is substantial. Minification should ideally average the texels falling inside a pixel rather than merely guessing an intermediate value.

# CHAPTER 7

# Lighting

## 7.1 Introduction

As previously described, both in scan-line rendering and ray-casting, the scene is constructed using a finite set of light sources. The contributions of all these light sources, denoted as $l$, are combined to determine the final color of each pixel.

Initially, we'll overlook the scenario where objects emit small amounts of light, which allows us to simplify the equation as follows:

$$L(x, \omega_r) = \underbrace{L_e(x, \omega_r)}_{\text{ignored}} + \sum_l L(l, \overrightarrow{lx}) f_{r'}(x, \overrightarrow{lx}, \omega_r)$$

Here, each term in the summation represents the product of two components: the light model, responsible for determining the quantity and direction of the relevant light source, and the Bidirectional Reflectance Distribution Function (BRDF), which governs how the surface reflects incoming light.

The light model defines how light is emitted in various directions within space. Given the position of a point $x$ on an object as input, it provides two outputs: a vector representing the direction of the light, and a color value representing the intensity of light received by point $x$ across different wavelengths.

The light direction is typically denoted by a vector $\overrightarrow{lx}$, with a convention that the sign of this vector points towards the light source. Additionally, it's important to note that the direction of the light vector is normalized, ensuring it remains a unit vector.

### 7.1.1 Light color

A vector $L(l, \overrightarrow{lx})$ with RGB components specifies the intensity of light for each wavelength, thereby defining its color. These components are not restricted to the 0 to 1 range; larger values can represent stronger light sources. However, it's imperative that the components remain non-negative.

Given that the light color $L(l, \overrightarrow{lx})$ is encoded in a vector, the Bidirectional Reflectance Distribution Function $f_r(x, \overrightarrow{lx}, \omega_r)$ also returns a color vector.

We will explore three fundamental direct light models commonly used in real-time graphics: direct point, and spot lights.

## 7.2 Direct light models

Directional lights are commonly employed to simulate distant sources such as sunlight. These sources are positioned far away from objects, thereby uniformly influencing the entire scene.

Because of their distant location, rays from directional lights are parallel across all positions in space, maintaining a constant color and intensity. The direction of such lights is represented by a constant vector $\mathbf{d} = (d_x, d_y, d_z)$ , independent of the position $x$ on the object. Similarly, the light color is specified by another constant vector $\mathbf{l} = (l_R, l_G, l_B)$.

For each point on an object, the direction of the light and its color are expressed using these constant values $\mathbf{l}$ and $\mathbf{d}$:

- Light intensity: $L(l, \overrightarrow{lx}) = \mathbf{l}_l$.
- Light direction: $\overrightarrow{lx} = \mathbf{d}_l$.

In the case of a single directional light, the rendering equation simplifies to:

$$L(x, \omega_r) = \mathbf{l} \times f_r(x, \mathbf{d}, \omega_r)$$

## 7.3 Point light model

Point lights are light sources that emit light from fixed positions in space and do not have a specific direction. Unlike directional lights, which emit light uniformly in a particular direction from a distant source, point lights radiate light equally in all directions from a single point in space. They are commonly used to simulate light sources such as lamps or bulbs, where light emanates spherically in all directions from a specific location.

The attributes of a point light, namely its position $\mathbf{p} = (p_x, p_y, p_z)$ and color $\mathbf{l} = (l_R, l_G, l_B)$ define its characteristics. The direction of light extends from point $\mathbf{x}$ to the light center, varying depending on the object's surface being illuminated. It's essential to normalize the light direction to ensure it's a unit vector:

$$\overrightarrow{lx} = \frac{\mathbf{p} - \mathbf{x}}{|\mathbf{p} - \mathbf{x}|}$$

The subtraction $\mathbf{p} - \mathbf{x}$ signifies the ray's orientation from the object toward the light source.

**Realistic light** To simulate realistic light behavior, point lights incorporate a decay factor. In physical terms, light intensity diminishes at a rate proportional to the inverse square of the distance. However, this may result in excessively dark images. Hence, light models typically allow users to specify a decay factor $\beta$, which can be constant, inverse-linear, or inverse-squared:

$$L(l, \overrightarrow{lx}) = \left( \frac{g}{|\mathbf{p} - \mathbf{x}|} \right)^{\beta} \mathbf{l}$$

The parameter $g$ that represents the distance at which the light reduction is exactly one; intensity surpasses $\mathbf{l}$ for distances shorter than $g$, and dims for longer distances.

**Rendering Equation for a single point light:** For a single point light, the rendering equation for a pixel can be stated as follows:

$$L(x, \omega_r) = \left( \frac{g}{|\mathbf{p} - \mathbf{x}|} \right)^{\beta} \mathbf{l} \cdot f_r \left( x, \frac{\mathbf{p} - \mathbf{x}}{|\mathbf{p} - \mathbf{x}|} . \omega_r \right)$$

This equation describes the light intensity $L$ at pixel $x$ in a scene, considering the illumination from a single point light source. The term $\left( \frac{g}{|\mathbf{p}-\mathbf{x}|} \right)^{\beta}$ represent the attenuation of light with distance, $\mathbf{l}$ represents the color of the light source, and $l_r$ is the Bidirectional Reflectance Distribution Function (BRDF), describing how light is reflected from the surface at point $x$ in the scene.

## 7.4 Spot light model

Spotlights are specialized projectors designed to illuminate specific objects or areas within a scene. They emit light in a conical pattern defined by a direction $\mathbf{d}$ and a position $\mathbf{p}$, with the light starting from point $\mathbf{p}$. The spotlight's illumination is divided into three zones by two angles, $\alpha_{IN}$ and $\alpha_{OUT}$: constant (inside $\alpha_{IN}$), decay (between $\alpha_{IN}$ and $\alpha_{OUT}$) and absent (outside $\alpha_{OUT}$). Within the decay zone, light intensity decreases linearly from the inner to the outer angle. These angles, $\alpha_{IN}$ and $\alpha_{OUT}$, allow for sizing the light to focus its effect on a specific subject. Typically, the cosine of half-angles of the inner and outer cones, $c_{in}$ and $c_{out}$, are used for implementation.

The cosine of the angle between the light direction vector $\overrightarrow{lx}$ and the spot direction $\mathbf{d}$ can be calculated by the dot product between the two:

$$\cos \alpha = \overrightarrow{lx} \cdot \mathbf{d}$$

The cone dimming effect is determined as:

$$\text{clamp} \left( \frac{\cos \alpha - c_{OUT}}{c_{IN} - c_{OUT}} \right)$$

Where,

$$\text{clamp}(y) = \begin{cases} 0 & y < 0 \\ y & y \in [0, 1] \\ 1 & y > 1 \end{cases}$$

Spotlights modulate other light sources with the introduced dimming term. They inherit the light direction $\overrightarrow{lx}_0$ from their parent model and adjust their color $L_0(l, \overrightarrow{lx})$ with the dimming term:

$$L(l, \overrightarrow{lx}) = L_0(l, \overrightarrow{lx}) \cdot \text{clamp} \left( \frac{\frac{\mathbf{p}-\mathbf{x}}{|\mathbf{p}-\mathbf{x}|} \cdot \mathbf{d} - c_{OUT}}{c_{IN} - c_{OUT}} \right)$$

The most common implementation of spotlights integrates the dimming factor with point lights. Here, spotlights are also characterized by a decay factor $\beta$, a target distance $g$, and a color vector $\mathbf{l}$. The light direction is computed similarly to point lights.

## 7.5 Area lights

Many real-world light sources do not emit light from a single point. Instead, they have a finite area from which light emanates. Area lights are designed to capture the shape and extent of these light sources within a scene. However, incorporating area lights introduces complexity, as individual sources cannot be treated independently anymore. In scan-line rendering, this necessitates the use of a full integral rather than simple point calculations.

Current methods for simulating area lights rely on approximations to this integral, as accounting for the complete light shape is computationally intensive. Moreover, these approximations are tightly coupled with the Bidirectional Reflectance Distribution Function of surfaces, making it challenging to separate the effects of area lights from surface properties.

## 7.6 Bidirectional reflectance distribution function

In scan-line rendering, the BRDF often lacks energy conservation in many cases. Typically, the BRDF comprises two main components:

- Diffuse Reflection
- Specular Reflection

$$f_r(x, \overrightarrow{lx}, \omega_r) = f_{diffuse}(x, \overrightarrow{lx}, \omega_r) + f_{specular}(x, \overrightarrow{lx}, \omega_r)$$

The diffuse component of the BRDF represents the primary color of an object. Shiny surfaces tend to reflect incoming light at specific angles, known as specular directions, which depend on the viewpoint $\omega_r$. This effect is captured in the specular component of the BRDF. Typically, the number and shape of highlights correspond to those of the direct light sources in the scene.

In scan-line rendering, BRDF values for each color frequency and both diffuse and specular components generally fall within the range of $[0, 1]$. However, due to the influence of individual lights, the resultant pixel color can sometimes exceed one. To address this, a common solution is to clamp the values within the range $[0, 1]$ at the end of computation. Although not physically accurate, this technique creates effects similar to overexposure in photography, where areas receiving excessive light appear as white spots.

### 7.6.1 High Dynamic Range

Modern rendering techniques employ more sophisticated computations, allowing values beyond the traditional $[0, 1]$ range. Instead of clamping, the final color is mapped into the $[0, 1]$ range using suitable non-linear functions:

$$L(x, \omega_r) = g(L'(x, \omega_r))$$

This approach enables the consideration of larger color dynamics, resulting in images with details visible in both very dark and extremely bright areas. However, HDR requires significantly more memory (four times as much) and greater computational power to apply the final non-linear scaling function.

## 7.7 Diffuse reflection models

This section will introduce the following diffuse reflection models:

- Lambert.
- Oren-Nayar.
- Toon.

### 7.7.1 Lambert reflection

The simplest form of BRDF consists solely of the diffuse part, which entails a constant term. This BRDF, known as Lambert reflection, is foundational in radiosity techniques.

According to Lambert's law of reflection, each point on an object hit by a ray of light reflects it with a uniform probability distribution in all directions above the surface. This reflection is independent of the viewing angle and corresponds to a constant BRDF:

$$f_r(x, \omega_i, \omega_r) = \rho_x$$

However, the amount of light received by an object varies with the angle between the light ray and the reflecting surface (the geometric term $G(x, y)$ of the rendering equation).

Let $n_x$ be the unit normal vector to the surface, $\mathbf{d} = \overrightarrow{lx}$ be the direction of the light ray, and $\alpha$ be the angle between the two vectors. Lambert demonstrated that the reflected light is proportional to $\cos\alpha$:

$$R_l = S_l \cos\alpha$$

Here, $S_l$ is the sent light, $R_l$ is the received light. Utilizing the geometric properties of the dot product, $\cos\alpha$ can be calculated as the dot product between the unit vectors corresponding to the normal vector $n_x$ and the light direction $\mathbf{d}$.

Let $m_D$ be a vector expressing the material's capability to perform Lambert reflection for each RGB color frequency (i.e., $\rho_x = m_D$):

$$m_D = (m_R, m_G, m_B)$$

Remarkably, when an object with a reflection factor $m_D$ is illuminated by a perfectly white source ($l = (1, 1, 1)$), it will display colors with RGB components equal to $m_D$. Hence, $m_D$ is commonly treated as a color parameter.

The BRDF of Lambert reflection for scan-line rendering can be expressed as:

$$f_r(x, \overrightarrow{lx}, \omega_r) = f_{diffuse}(x, \overrightarrow{lx}) = \mathbf{m}_D \max(\overrightarrow{lx}\mathbf{n}_x, 0)$$

When a face is opposite to a light source, it cannot be illuminated, as the cosine becomes negative. Clamping the value at zero ensures that the faces are not illuminated. Since the Lambert reflection model solely comprises the diffuse component of lighting, the vector $m_D$ is often referred to as the diffuse color of the object, representing its primary color.

It's worth noting that the pixel color remains independent of $\omega_r$; when Lambert diffuse reflection is applied, the viewing angle has no effect, and the final image depends solely on the objects' positions and the directions of the lights.

### 7.7.2 Toon shading

Toon shading simplifies the color output range by using discrete values based on predefined thresholds, resulting in a cartoon-like rendering style. This technique can be applied to both the diffuse and specular components of the BRDF.

To start, a standard Lambert BRDF is used for the diffuse component, and either a Phong or Blinn BRDF with $\gamma = 1$ is employed for the specular component. Two colors ($\mathbf{m}_{D1}, \mathbf{m}_{D2}$) or ($\mathbf{m}_{S1}, \mathbf{m}_{S2}$) and a threshold ($t_D$ or $t_S$) are utilized to determine which color to choose. For the diffuse component:

$$f_{diffuse}(x, \overrightarrow{lx}) = \begin{cases} \mathbf{m}_{D1} & \overrightarrow{lx}\mathbf{n}_x \geq t_d \\ \mathbf{m}_{D2} & \overrightarrow{lx}\mathbf{n}_x < t_d \end{cases}$$

For the specular component:

$$\mathbf{r}_{l,x} = 2\mathbf{n}_x \left(\overrightarrow{lx}\mathbf{n}_x\right) - \overrightarrow{lx}$$

$$f_{specular}(x, \overrightarrow{lx}, \omega_r) = \begin{cases} \mathbf{m}_{S1} & \omega_r \mathbf{r}_{l,x} \geq t_s \\ \mathbf{m}_{S2} & \omega_r \mathbf{r}_{l,x} < t_s \end{cases}$$

To achieve better visual results, more than two colors are often used for both the specular and diffuse parts. Additionally, small gradients are added to smooth transitions between different colors. This smoothing is usually implemented using a color that is a function of the cosine of the angles between the considered rays. These functions are implemented as 1D textures, which offers performance benefits in modern GPU hardware, as texture look-up is generally more efficient than program branching.

### 7.7.3 Oren-Nayar

Certain real-world materials like clay, dirt, and certain types of cloth exhibit a unique optical phenomenon known as retro-reflection. These materials tend to reflect light back in the direction of the light source due to their rough surfaces, which cannot be accurately simulated using the Lambert diffuse reflection model. To address this, the Oren-Nayar diffuse reflection model has been developed to accurately represent such materials.

In most cases, materials exhibiting retro-reflection do not display specular reflections and are characterized solely by a diffuse component. The model requires three vectors: the direction of the light $\mathbf{d}$, the surface normal vector $\mathbf{n}$, and the viewing direction $\omega_r$.

These vectors define three angles:

- $\theta_i$ between $\mathbf{d}$ and $\mathbf{n}$.
- $\theta_r$ between $\omega_r$ and $\mathbf{n}$.
- $\gamma$ between the projections of $\omega_r$ and $\mathbf{d}$ on the plane perpendicular to $\mathbf{n}$, denoted as $v_i$ and $v_r$, respectively.

The model is characterized by two parameters:

- $m_D = (m_R, m_G, m_B)$, representing the main color of the material.
- $\sigma \in \left[0, \frac{\pi}{2}\right]$, which denotes the roughness of the material. Higher values of $\sigma$ correspond to rougher surfaces.

The model converges to the Lambert diffusion model when $\sigma = 0$

$$\begin{aligned}
\theta_i &= \cos^{-1}(d \cdot n_x) \\
\theta_r &= \cos^{-1}(\omega_r \cdot n_x) \\
\alpha &= \max(\theta_i, \theta_r) \\
\beta &= \min(\theta_i, \theta_r) \\
A &= 1 - 0.5 \cdot \frac{\sigma^2}{\sigma^2 + 0.33} \\
B &= 0.45 \cdot \frac{\sigma^2}{\sigma^2 + 0.09} \\
v_i &= \text{normalize}(d - (d \cdot n_x)n_x) \\
v_r &= \text{normalize}(\omega_r - (\omega_r \cdot n_x)n_x) \\
G &= \max(0, v_i \cdot v_r) \\
L &= m_d \cdot \text{clamp}(d \cdot n_x) \\
\text{diffuse}(x, L_x, \omega_r) &= L \cdot (A + B \cdot G \sin\alpha \tan\beta)
\end{aligned}$$

Parameter $A$ controls how closely the surface adheres to the Lambert principle, while parameter $B$ controls the extent of the retro-reflection effect. In practice, materials often directly specify parameters $A$ instead of $\sigma$, and $B \sin\alpha \tan\beta$ is pre-computed and interpolated from a texture function $f(\cdot)$, dependent on $d\mathbf{n}_x$ and $\omega_r\mathbf{n}_x$.

## 7.8 Specular reflection

We'll explore the following specular reflection models:

- Phong specular reflection.
- Blinn specular reflection.
- Ward.
- Cook-Torrance.

A perfect mirror surface reflects light in only one direction, lying on the same plane as both the incident light and the surface's normal, but with an opposite angle. Consequently, light reflected from such a surface would be visible solely along this specific angle and invisible in any other direction. However, if a surface is rough, incoming light will also be reflected at angles close to the mirror reflection angle. Hence, even though at reduced intensities, the reflected ray could be visible in an area near the mirror direction.

Specular reflection can be accounted for by adding a specular term to the diffuse component. This term calculates the probability of mirror reflection occurring in the given viewing direction $\omega_r$. Similar to the diffuse case, the specular component is characterized by a color $m_S$ that dictates how the RGB components of the incoming light are reflected. In most cases, objects exhibit white specular color, i.e., $m_S = (1, 1, 1)$. However, metallic objects like gold or copper have specular colors identical to their diffuse counterparts.

### 7.8.1 Phong reflection model

In the Phong model, the mirror reflection direction $\mathbf{r}$ is first calculated. For parallel projection, $\omega_r$ remains constant at$\mathbf{p}$ In the case of perspective projection, $\omega_r$ can be computed as the normalized difference between the surface point $\mathbf{x}$ and the center of projection $\mathbf{c}$.

The Phong specular reflection model takes into account the angular distance $\alpha$ between the specular direction and the observer. It computes the intensity of specular reflection based on $\cos\alpha$: the term reaches its maximum when the specular direction aligns with the observer and diminishes to zero as the angle exceeds 90°. To confine highlight regions more effectively, $\cos\alpha$ is raised to the power of $\gamma$. A higher $\gamma$ results in smaller highlight areas, making the object appear shinier as its surface behaves more like a mirror.

To calculate the direction of the reflected ray, $n'$, the projection of the light vector onto the normal vector is first computed. This is achieved by finding its length with the dot product between the normal and light direction ($d \cdot n_x$), and then multiplying the result with the normal vector $\mathbf{n}_x$ to obtain a vector perpendicular to it. Subsequently, subtracting $n'$ from the light vector yields $d'$, the perpendicular from $\mathbf{d}$ to $\mathbf{n}$. By adding $d'$ twice to $d$, the reflected vector $\mathbf{r}$ is obtained:

$$r = d + 2d'$$

In summary:

$$\mathbf{r}_{l,x} = 2\left(\overrightarrow{xl}\mathbf{n}_x\right)\mathbf{n}_x - \overrightarrow{xl}$$

Note that many shading languages provide built-in functions to directly compute the reflected vector.

The intensity of the specular reflection term can then be computed as:

$$\cos^\gamma\alpha = \text{clamp}(\omega_r \cdot \mathbf{r})^\gamma$$

Similar to the Lambert diffuse term, it is necessary to exclude cases where the cosine is negative using the `clamp()` function.

**Simplified parametrization** The Blinn reflection model is an alternative to the Phong shading model that employs the half vector $\mathbf{h}$: a vector positioned midway between the viewer direction $\omega_r$ and the light $\mathbf{d}$.

### 7.8.2 Blinn reflection model

The angle $\alpha$ between the observer and the reflected ray is approximated by the angle $\alpha'$ between the normal vector $\mathbf{n}_x$ and the half vector $\mathbf{h}$. The half vector $\mathbf{h}$ can be computed as the normalized average of vectors $\mathbf{d}$ and $\omega_r$. Using the notation from the rendering equations:

$$\mathbf{h}_{l,x} = \frac{d + \omega_r}{|d + \omega_r|} = \text{normalize}(d + \omega_r)$$

The specular highlight is then computed by raising to the power of $\gamma$ the cosine of $\alpha'$, expressed as the dot product of $\mathbf{n}_x$ and $\mathbf{h}_{l,x}$. The formula for Blinn specular reflection is:

$$f_{specular}(x, \overrightarrow{lx}, \omega_r) = \mathbf{m}_S \cdot \text{clamp}(\mathbf{n}_x \cdot \mathbf{h}_{l,x})^\gamma$$

The Blinn specular model is typically slightly more computationally expensive than the Phong model (as it requires normalization, which is more complex than simple reflection). However, it is still easily achievable in real-time with current hardware.

Since the two techniques yield slightly different results, they are often both implemented and chosen by artists to achieve various effects. For instance, in the examples provided, the left image employs the Phong model, while the right image uses the Blinn specular computation. Blinn reflections generally exhibit a larger decay area compared to Phong reflections with similar parameters.

### 7.8.3 Ward anisotropic specular model

Some objects, such as hairs, CDs, or brushed metals, are characterized by grooves on their surfaces, leading to specular highlights oriented along these grooves. Such surfaces are termed anisotropic materials. The Ward specular model is significant for two key reasons:

1. It is derived from physically inspired principles.
2. It supports anisotropic reflections.

To accommodate anisotropy, it's necessary to specify an orientation for the grooves on the surface by assigning two additional vectors alongside the normal. These vectors, termed tangent and bi-tangent, are denoted by **b** and **t** respectively. We will delve into methods for encoding or deriving these vectors in a future lesson.

Similar to the Blinn specular model, the Ward technique relies on the half-vector **h** between the light and viewer directions. Specifically, it depends on the angle of this vector **h** with the normal (denoted with $\delta$), and the angle between its projection onto the **bt**-plane and the groove direction (denoted with $\phi$).

The formula for the specular component of the Ward model incorporates two roughness parameters for the **h** and **bh** directions, denoted $\alpha_t$ and $\alpha_b$ respectively:

$$f_{\text{specular}}(\mathbf{x}, \mathbf{L}_x, \omega_r) = \frac{e^{-\frac{\left(\frac{\mathbf{ht}_x}{\alpha_t}\right)^2+\left(\frac{\mathbf{hb}_x}{\alpha_b}\right)^2}{\mathbf{hn}_x^2}}}{4\pi\alpha_t\alpha_b\sqrt{\frac{\omega_r\mathbf{n}_x}{\overrightarrow{lx}\mathbf{n}_x}}}$$

### 7.8.4 The Cook-Torrance reflection model

Realistic specular highlights on objects often display a soft falloff area. However, conventional models like Blinn and Phong yield a sharp falloff. Additionally, according to the Fresnel principle, objects tend to exhibit stronger specular reflection when the incident light is nearly parallel to the surface. These observations, among others, prompted the development of more sophisticated specular illumination models capable of accurately capturing these physical characteristics.

The Cook-Torrance Bidirectional Reflectance Distribution Function (BRDF) model aims to compute both the specular and diffuse components in a physically accurate manner. The diffuse component adheres to the Lambert diffusion model. However, for achieving a physically accurate behavior, it is blended with the specular part through linear interpolation, governed by a coefficient $k$:

$$f_r(x, \overrightarrow{lx}, \omega_r) = k \cdot f_{diffuse}(x, \overrightarrow{lx}, \omega_r) + (1-k) f_{specular}(x, \overrightarrow{lx}, \omega_r)$$

Here, $f_{diffuse}(x, \overrightarrow{lx}, \omega_r) = \mathbf{m}_D \cdot \text{clamp}(\overrightarrow{lx}\mathbf{n}_x)$. The specular term is computed as the product of three factors:

$$f_{specular}(x, \overrightarrow{lx}, \omega_r) = \mathbf{m}_S \frac{D \cdot F \cdot G}{4 \cdot \text{clamp}(\omega_r \mathbf{n}_x)}$$

Similar to other specular models, it is characterized by a specular color $\mathbf{m}_S$. The $\text{clamp}(\omega_r \cdot \mathbf{n}_x)$ term is a geometric factor that normalizes the values computed by components $D$, $F$, and $G$.

The model also defines a constant $\rho$, commonly referred to as roughness, which determines the smoothness of the surface:

- $\rho = 0$ denotes a perfectly smooth object.
- $\rho = 1$ represents a very rough surface.

For each of the three terms $D$, $F$, and $G$, various definitions exist, each characterized by its features and complexities. Many formulations will rely on the half-vector, defined for the Blinn specular model as:

$$\mathbf{h}_{l,x} = \frac{\overrightarrow{lx} + \omega_r}{\left|\overrightarrow{lx} + \omega_r\right|} = \text{normalize}(\overrightarrow{lx} + \omega_r)$$

**Distribution term** The distribution term $D$ encapsulates the surface roughness. The Blinn version adapts the corresponding specular model to the Cook-Torrance framework. Specifically, it replaces the specular power $\gamma$ with the roughness parameter $\rho$ and includes a normalization factor.

$$D = \frac{(\mathbf{h}_{l,x} \cdot \mathbf{n}_x)^{\frac{2}{\rho^2}-2}}{\pi\rho^2}$$

The Beckmann version relies on parameter $rho$ to define the average slope of the surface at a microscopic level.

$$D = \frac{e^{-\left(\frac{\tan\alpha}{\rho}\right)^2}}{\pi\rho^2\cos^2\alpha}$$

Here, $\alpha = \arccos(\mathbf{h}_{l,x} \cdot \mathbf{n}_x)$ The GGX version of the distribution term $D$ employs the following definition. In certain studies, it has been shown to yield the most realistic outcomes while maintaining a complexity akin to the Blinn version.

$$D = \frac{\rho^2}{\pi(\text{clamp}(\mathbf{h}_{l,x} \cdot \mathbf{n}_x)^2(\rho^2 - 1) + 1)^2}$$

**Fresnel term** The Fresnel term $F$ is dependent on a parameter $F_0 \in [0, 1]$. It dictates how the light response alters concerning the angle of incidence relative to the viewer and can be approximated with the following expression:

$$F = F_0 + (1 - F_0)\left(1 - \text{clamp}(\omega_r \cdot \mathbf{h}_{l,x})\right)^5$$

**Geometric term** The microfacet version of the geometric term $G$ isn't characterized by any parameters and solely relies on the angles.

$$G = \min\left(1, \frac{2(\mathbf{h}_{l,x} \cdot \mathbf{n}_x)(\omega_r \cdot \mathbf{n}_x)}{(\omega_r \cdot \mathbf{h}_{l,x})}, , \frac{2(\mathbf{h}_{l,x} \cdot \mathbf{n}_x)(\overrightarrow{lx} \cdot \mathbf{n}_x)}{(\omega_r \cdot \mathbf{h}_{l,x})}\right)$$

The GGX version for the geometric term $G$ depends on the surface roughness $\rho$ and utilizes a helper function that is first invoked with the light direction and then with the viewer direction.

$$g_{GGX}(\mathbf{n}, \mathbf{a}) = \frac{2}{1 + \sqrt{1 + \rho^2 \frac{1 - (\mathbf{na}^2)}{\mathbf{na}^2}}}$$

$$G = g_{GGX}(\mathbf{n}_x, \omega_r) g_{GGX}(\mathbf{n}_x, \overrightarrow{lx})$$

**Summary** Despite its complexity, the Cook-Torrance reflection model offers realistic reflections that consider numerous physical behaviors. Typically, it serves as a fundamental building block for various advanced rendering techniques. These techniques generate specialized textures, utilized as look-up tables, enabling the rendering of this reflection model in real-time.

## 7.9 Emission and indirect lighting approximation

### 7.9.1 Material emission

The emission term of a material represents the small amount of light directly emitted by an object. It corresponds to the emissive part of the rendering equation:

$$L_e(x, \omega_r)$$

This material emission term is combined with other parts of the rendering equation. It is independent of the environment but solely depends on the considered object.

For instance, considering a direct light source, Phong specular, Lambert diffuse reflection, and emission, we have:

$$\mathbf{r}_x = 2\mathbf{n}_x \cdot (\mathbf{d} \cdot \mathbf{n}_x) - \mathbf{d}$$

$$L(x, \omega_r) = \text{clamp}(\mathbf{I}_D \cdot (\mathbf{m}_D \cdot \text{clamp}(\mathbf{d} \cdot \mathbf{n}_x) + \mathbf{m}_S \cdot \text{clamp}(\omega_r \cdot \mathbf{r}_x)^\gamma) + \mathbf{m}_E)$$

### 7.9.2 Ambient lighting

When considering only direct light sources (such as directional, point, or spotlight), images can appear very dark. Realistic rendering techniques aim to incorporate indirect lighting as well—illumination caused by light bouncing off other objects.

Ambient lighting serves as the simplest approximation for indirect illumination. It represents a constant factor for the entire scene, encompassing light reflected by all objects in all directions. The ambient light emission is defined by a constant RGB color value $\mathbf{l}_A$.

To extend the Bidirectional Reflectance Distribution Function (BRDF) of the object, another component $f_A(x, \omega_r)$ specifically accounts for ambient lighting. This component is independent of the light direction:

$$L(x, \omega_r) = \sum_l L(l, \overrightarrow{lx}) f_r(x, \overrightarrow{lx}, \omega_r) + \mathbf{l}_A f_A(x, \omega_r)$$

In most cases, the BRDF for the ambient term $f_A(x, \omega_r)$ is a constant known as the ambient light reflection color $\mathbf{m}_A$. Typically, $\mathbf{m}_A$ corresponds to the main color of the object (i.e., $\mathbf{m}_A = \mathbf{m}_D$), but it can be adjusted to achieve specific lighting effects for particular objects.

In the case of a single direct light source, plus the ambient term (assumed to be constant), the rendering equation simplifies to:

$$L(x, \omega_r) = \mathbf{l} \cdot f_r(x, \mathbf{d}, \omega_r) + \mathbf{l}_A \cdot \mathbf{m}_A$$

**Hemispheric lighting** A slight extension of ambient lighting is hemispheric lighting. In this case, there are two ambient light colors—the upper or sky color and the lower or ground color—along with a direction vector. This model simulates the impact of both the sky and ground colors on the indirect light component for the object under consideration.

This technique creates an ambient light color factor by blending the two colors based on the orientation of the object. The two colors, $\mathbf{l}_U$ and $\mathbf{l}_D$, represent the ambient light values at the two extremes, while the direction vector $\mathbf{d}$ governs the blending of the two colors. The orientation of the object is characterized by $\mathbf{n}_x$, the normal vector to the surface at point $x$.

If the normal vector is aligned and in the same direction as $\mathbf{d}$, the ambient color $\mathbf{l}_U$ is used. Conversely, if the normal vector is aligned but in the opposite direction of $\mathbf{d}$, the ambient color $\mathbf{l}_D$ is used. For normal vectors oriented in other directions, the two colors are blended proportionally to the cosine of their angle with the vector $\mathbf{d}$. In particular, $\mathbf{l}_A(x)$ is defined as follows:

$$\mathbf{l}_A(x) = \frac{n_x \cdot d + 1}{2}\mathbf{l}_U + \frac{1 - n_x \cdot d}{2}\mathbf{l}_D$$

In the case of a single direct light source plus the hemispheric ambient term, the rendering equation simplifies to:

$$L(x, \omega_r) = \mathbf{l} \cdot f_r(x, \mathbf{d}, \omega_r) + \left(\frac{n_x \cdot d + 1}{2}\mathbf{l}_U + \frac{1 - n_x \cdot d}{2}\mathbf{l}_D\right) \cdot \mathbf{m}_A$$

### 7.9.3 Image based lighting

MAchieving more accurate reproduction of light sources and advanced approximations to rendering equations are crucial for the photorealistic effects seen in high-end 3D applications. While a comprehensive discussion of these techniques is beyond this course's scope, we can briefly outline some fundamental concepts.

Hemispheric lighting, as introduced earlier, computes the light received by an object based on the orientation of its surface points, determined by their corresponding normal vectors. This computation is relatively straightforward, interpolating between two colors based on relative orientation to a given direction.

Image-based lighting extends this idea by defining generic functions $\mathbf{l}_A(\mathbf{x}_i)$ that return the color received from the environment by a point $\mathbf{x}_i$ on a surface oriented in the direction described by its normal vector $\mathbf{n}_i$. Each point is illuminated according to $\mathbf{l}_A(\mathbf{x})$.

These functions $\mathbf{l}_A(\mathbf{x})$ can be computed from specially taken photographs or high-quality offline rendering of the environment, encoded in a format suitable for real-time use.

Starting with an image (hence the name image-based lighting) and applying filtering, the actual light received from each point in any direction can be determined. The approximation of the rendering equation remains similar to hemispheric lighting, with the light function adjusted to return values computed in the filtering step. This approximation includes both ambient and diffuse components of light:

$$L(x, \omega_r) = \sum_l L(l, \overrightarrow{lx}) f_r(x, \overrightarrow{lx}, \omega_r) + \mathbf{l}_A(x) f_A(x, \omega_r)$$

Efficiently encoding the function $\mathbf{l}_A(\mathbf{x})$ is challenging. Common approaches include interpolating values stored in a table (Cubic Mapping), spectral expansion (Spherical Harmonics), and other approximations.

**Spherical Harmonics** Spherical Harmonics expansion expresses the color received from a given direction $\mathbf{n}_x$ as a sum of contributions multiplied by a set of basic functions indexed by two numbers $l$ and $m$, with $-l \leq m \leq l$ and $l \geq 0$. For low values of $l$ (e.g., $l \leq 1$), the

expansion simplifies significantly:

$$\mathbf{l}_A(x) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \mathbf{c}_{l,m} \cdot y_l^m(\mathbf{n}_x)$$

If we denote the components of the (unitary) normal vector direction $\mathbf{n}_x$ as $(\mathbf{n}_x).x$, $(\mathbf{n}_x).y$, and $(\mathbf{n}_x).z$, and we restrict ourselves to $l \leq 1$, the expansion yields a particularly simple expression:

$$\mathbf{l}_A(x) = \rfloor_{0,0} + (\mathbf{n}_x).x \cdot \rfloor_{1,1} + (\mathbf{n}_x).y \cdot \rfloor_{1,-1} + (\mathbf{n}_x).z \cdot \rfloor_{1,0}$$

For $l \leq 2$, the expression becomes slightly more complex, but it enables the capture of a broader range of illumination conditions using only nine coefficients.

# CHAPTER 8

# Vulkan

## 8.1 Introduction

Modern computer architectures exhibit a multitude of components, including:

- Multiple CPU cores.
- One or more distinct GPUs.
- Diverse memory configurations, encompassing both CPU and GPU memory.
- Concurrent utilization by various applications or virtual machines, necessitating simultaneous access to CPUs and GPUs.

Vulkan emerges as a solution to optimize resource utilization, allowing users to harness available capabilities to their fullest potential. However, this advancement comes with a significant caveat: the complexity of setup is immense. Vulkan is designed to operate across a wide spectrum of systems, encompassing:

- Desktop computers (PCs, Macs, Linux-based systems, etc.).
- Mobile devices (Smartphones, Tablets, VR Headsets, etc.).
- Gaming consoles (e.g., Nintendo Switch).
- Embedded systems (such as map displays in automobiles).

Each system boasts its unique characteristics, and Vulkan strives to provide comprehensive support across this diversity.

### 8.1.1 Starter library

Exploiting all of Vulkan's features in an application typically involves numerous steps. However, in many cases, users rely on the same foundational startup sequence. The `Starter.hpp` file, utilized across all assignments, serves the purpose of defining a standardized initialization procedure. This approach alleviates the need for users to explicitly repeat all the "normal startup steps" within their projects. A typical Vulkan application follows this structure:

```
void run() {
    initWindow();   // Create the operating system window
    initVulkan();   // Set up Vulkan resources
    initApp();      // Load and set up application elements
    mainLoop();     // Execute the update/render cycle of the application
    cleanup();      // Release all resources
}
```

### 8.1.2 Presentation surface

The designated screen area where the host operating system permits Vulkan to render images is referred to as the presentation surface.

For a Vulkan application to function correctly, it must acquire an appropriate presentation surface from the operating system. This particular step is contingent upon the system in use, and we will delve deeper into it later.

**Application window creation** In desktop environments like MS Windows, MacOS, or Linux, the presentation surface is typically encapsulated within a window. For the purposes of this course, we'll focus solely on desktop applications.

GLFW provides a platform-independent method to create windows. Before opening a window with GLFW, initialization of GLFW is necessary. Various parameters can be utilized to specify the characteristics of the window created. In GLFW, this is achieved using the `glfwWindowHint(prop, val)` command, assigning the value `val` to the property `prop`. Since GLFW's default mode is geared towards OpenGL, setting the `GLFW_CLIENT_API` property to `GLFW_NO_API` is imperative for Vulkan usage. Numerous other options can also be configured. The `glfwCreateWindow()` function is employed to generate the operating system window and returns its identifier. This procedure accepts the window's width and height (specified in pixels as `WIDTH` and `HEIGHT`) along with a string to display in the title bar. In applications utilizing `Starter.hpp`, this process is encapsulated within a specific procedure callback named `setWindowParameters()`:

```
void initWindow() {
    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    window = glfwCreateWindow(WIDTH, HEIGHT, 'Vulkan', nullptr, nullptr);
}
```

## 8.2 Vulkan initialization

Initializing Vulkan support is quite complex due to the numerous configuration options available. To understand this process, we need to start with a high-level overview of a Vulkan application.

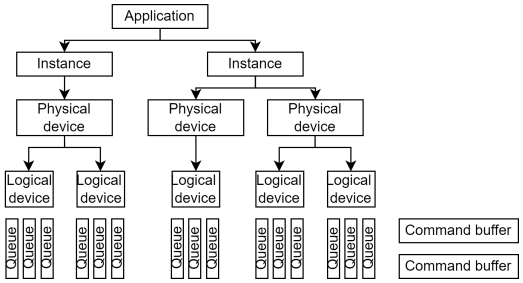



Figure 8.1: Vulkan architecture

At the top level, we have the Vulkan application, which can simultaneously utilize multiple software libraries that leverage Vulkan to exploit GPU capabilities. Each library operates independently through Vulkan instances.

- *Instances*: enable each software library to function independently within the application.
- *Physical devices*: correspond to GPUs. If a system has multiple GPUs, the application can use all of them in parallel to enhance performance or select the most suitable one based on power or efficiency needs.
- *Logical devices*: allow for different configurations of the same GPU to coexist (e.g., one window with antialiasing and another without). This enables the use of the same GPU for varied purposes.

To maximize parallelization, Vulkan operations are managed through queues. This setup allows the application to record the steps for composing the next image while the previous image's processing is still ongoing. Users can request multiple queues as needed and manage their synchronization to fit their application's requirements.

- *Queues*: facilitate parallel processing by enabling multiple tasks to be recorded and executed simultaneously.
- *Command buffers*: store Vulkan operations and transfer them into GPU memory. Each queue can handle several command buffers, allowing different threads running on separate CPU cores to record commands in parallel. This approach minimizes the need to transfer data from the CPU to GPU memory, as only the necessary updates are transferred.

### 8.2.1 Typical Vulkan application

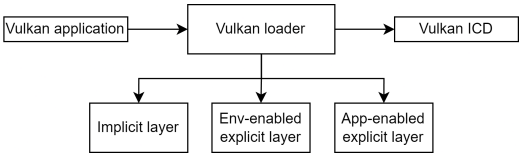



Figure 8.2: Vulkan application

Most Vulkan applications, however, will use only a single library instance, operate on a single GPU, utilize just one configuration, and perform all commands in a single queue.

**Vulkan structure** Vulkan's structure is inherently complex, consisting of a fixed component known as the Vulkan Loader, and a set of GPU drivers called Installable Client Devices (ICDs). A specific Vulkan deployment can also include a set of Extension Layers, which expose OS-specific or device-specific functions. These functions enable Vulkan to operate within a particular environment and access special hardware features.

**Instance creation** Creating a Vulkan instance requires specifying the following information:

- List of requested extensions.
- Name and other features of the application.

To maintain platform independence, the GLFW interface library provides the `glfwGetRequiredInstanceExtensions()` command, which retrieves the necessary extensions for the architecture on which the program is running. When the instance is no longer needed, it should be released using `vkDestroyInstance()`. Similarly, the operating system window should be closed using `glfwDestroyWindow()`. To free all remaining resources, the GLFW library also requires a call to `glfwTerminate()`.

### 8.2.2 Presentation surface

Creating a presentation surface in Vulkan requires both a window and a Vulkan instance to be established first. Therefore, the presentation surface can only be created after these initial steps are completed.

The GLFW library facilitates the creation of the presentation surface through the `glfwCreateWindowSurface()` function. This function returns a handle to the surface in the VkSurfaceKHR object, which is passed as the last argument of the function.

To release the presentation surface at the end of the application, it must be destroyed before the Vulkan instance and the window are destroyed. This is done using the `vkDestroySurfaceKHR()` command.

### 8.2.3 Physical devices

Even though this course focuses on applications utilizing a single GPU, the system in use may have multiple GPUs available. Each physical device (GPU) is characterized by various attributes:

- *Properties*: includes details such as the manufacturer, whether the GPU is integrated into the CPU or separate, driver IDs, etc.
- *Features*: indicates support for specific types of shaders, data types, graphics commands, and other capabilities.
- *Memory types*: specifies whether the memory is shared, GPU-specific, etc.
- *Memory heaps*: defines the total available memory.
- *Supported queue families*: specifies the types of operations the device can perform.

Each property contains a field called limits, which indicates the maximum sizes of supported objects. The features structure includes numerous fields, each representing a feature that can be selected during the creation of logical devices.

**Memory types** Memory types describe whether the corresponding memory is CPU-visible, GPU-only, and how it can be interfaced with Vulkan. Memory heaps define the available memory quantity and whether the memory is local to the GPU.

**Logical devices and queues** Logical devices are created from physical devices, each containing one or more queues. Each physical device can support different types of queues, and the selection of a physical device may depend on the queues its logical devices can utilize. Queues are grouped into families, each supporting different types of operations.

The function `vkGetPhysicalDeviceQueueFamilyProperties()` can enumerate the queue families supported by a physical device. Queue types include: graphics, compute, transfer, sparse memory management, and presentation. A computer graphics application typically requires at least a graphics queue and a presentation queue. Depending on the system, these might be separate or supported by one specific queue family.

Logical devices and their queues are created using the `vkCreateDevice()` command, starting from a physical device. During creation, device features, extensions, and debug layers are enabled. Once the device is created, queue handles must be retrieved using the `vkGetDeviceQueue()` command. Since multiple queues per family can be created, this command requires both the family index and the queue ID. At the end of the application, logical devices must be released.

### 8.2.4 Command buffer

Once queues have been retrieved, command buffers can be created to utilize them. Given the frequent use of multiple command buffers, they are allocated from larger groups known as command pools. Each command pool is specifically tied to the queue families it utilizes.

Command pools are created using the `vkCreateCommandPool()` function. The primary parameter to define in the creation structure is the `queueFamilyIndex` field, which specifies the queue family on which its commands will be executed. Command pools should be released when no longer needed using the `vkDestroyCommandPool()` function.

Command buffers are created from these pools using the `vkAllocateCommandBuffers()` function. Their handles are returned in `VkCommandBuffer` objects. The command pool handle is passed in the commandPool field of the creation structure. Multiple command buffers can be created in a single call; their number is specified in the commandBufferCount field. If more than one buffer is required, the return value must be an array of the appropriate size.

Command buffers are automatically destroyed when their corresponding pool is released, eliminating the need for explicit destruction.

### 8.2.5 Vulkan swap chain

In Vulkan, screen synchronization is managed with a generic circular queue called the Swap Chain. The swap chain can handle single, double, triple buffering, and potentially longer presentation queues. The properties of the swap chain depend on the combination of the surface and the physical device.

Each swap chain is characterized by:

- *Capabilities*: basic information such as the number of buffers supported and graphical extents.
- *Supported formats*: different color spaces and resolutions.

- *Presentation modes*: synchronization algorithms used in Vulkan.

Swap chain capabilities provide fundamental details like the number of supported buffers and graphical extents. Even though colors are encoded using the RGB system, several alternative formats exist, each with different color spaces and resolutions. Graphics adapters can support a variety of these formats, such as 8bpc, 10bpc, and 16bpc, each offering different trade-offs between memory usage, performance, and quality. Swap chain formats are defined by the number of bits and components (specified in an enumeration) and the corresponding color profile.

Presentation modes, which are akin to synchronization algorithms in Vulkan terminology, are returned as an array of
`VkPresentModeKHR` enumerations via the
`vkGetPhysicalDeviceSurfacePresentModesKHR` command. Four main presentation modes are supported.

When no longer needed, swap chains can be released using the `vkDestroySwapchainKHR()` command. Each buffer in the swap chain is treated as a generic image in Vulkan and must be retrieved after creation. These images are identified by `VkImage` objects and can be retrieved using the `vkGetSwapchainImagesKHR` command.

**Image views** Images can be of various formats and used for different purposes, requiring extra information about their structure and pixel access methods. Image views provide this necessary information by associating each image with a description of how it can be used and accessed.

After retrieving the swap chain images, their views (contained in `VkImageView` objects) must be created using the `vkCreateImageView()` command. The most critical information is the corresponding image, specified in the image field. Additional details will be discussed in subsequent sections.

Swap chain images are destroyed with the `VkSwapchainKHR` object, but image views must be explicitly destroyed using the `vkDestroyImageView()` command.

## 8.3 Layout

The Vulkan graphics pipeline processes rendering as follows:

- *Mesh initialization*: it begins with a mesh defined by a set of vertices and indices.
- *Vertex processing*: each vertex is processed by a vertex shader, which computes its normalized screen coordinates and other parameters to pass to the fragment shader.
- *Fragment processing*: for every fragment (pixel) on the screen, the fragment shader computes the final color of the pixel, implementing algorithms like the Bidirectional Reflectance Distribution Function (BRDF).

**Pipelines** The graphics pipeline in Vulkan, while based on a set of fixed functions configurable by the user, must support a wide array of use cases, each with unique characteristics. Because there is no one-size-fits-all solution, Vulkan provides users with extensive programming capabilities to tailor the pipeline to their specific needs. Besides allowing custom shader writing, Vulkan enables users to:

- *Vertex data definition*: specify which information is associated with vertices.
- *Shader communication*: define the information passed between vertex and fragment shaders and determine if and how it is interpolated.
- *Shader parameters*: define additional parameters that can be passed to shaders for proper vertex processing and fragment computation.

### 8.3.1 Data structure

In Vulkan, the term Layout encompasses any data structure employed to specify the format and type of information within user-defined encodings. This includes defining pixel color encodings such as the number of bits per pixel, selected color space, and the presence of an alpha-channel.

However, the term "Layout" is sometimes ambiguously used in Vulkan documentation, leading to confusion regarding its context. This ambiguity can be problematic when referencing related documentation and tutorials.

Despite this ambiguity, Vulkan manages layouts in a manner that encourages reusability whenever possible. Moreover, Vulkan emphasizes interoperability among shaders, vertices, and various data blocks, facilitating the mixing and matching of data types that yield identical results.

**Vertex attributes** In GLSL, in and out global variables serve as the interface between shaders and other components of the pipeline, whether fixed or programmable. Within the graphics pipeline, vertex-related values are directly transmitted to the Vertex Shader by the Input Assembler component. Specifically, vertex coordinates are conveyed through in variables of the Vertex Shader.

Each vertex possesses an implicit integer index represented by the global variable `gl_VertexIndex`. Additionally, vertices can feature an arbitrary array of user-defined attributes, each characterized by one of the supported GLSL types. Some vertices may lack user-defined attributes altogether.

For instance, in a 2D game application, a set of vec2 normalized screen coordinates might directly denote element positions. Conversely, a typical 3D scene employs at least a vec3 element to store positions in 3D local space. Other pipelines might necessitate vec3 position attributes (measured in 3D local coordinates) and vec3 color attributes to vary diffuse reflection across object surfaces.

Consistency is crucial: all vertices within a mesh must adhere to the same vertex format, i.e., share the same attributes. The pipeline's fixed functions facilitate the transmission of such values to the Vertex Shaders.

Different meshes may feature distinct vertex formats, albeit requiring the creation of separate pipelines. Vulkan offers exceptional flexibility in configuring pipelines to specify the vertex attributes transmitted to the Vertex Shader. Specifically, Vulkan enables the partitioning of vertex data into separate arrays, each containing specific attributes. These arrays, known as bindings, are distinguished by progressive binding IDs. Although various approaches exist, employing a single binding is most common.

Typically, a C++ structure encapsulating all vertex attributes is created, utilizing GLM types aligned with those defined in the corresponding Vertex Shader. This binding is defined within a VkVertexInputBindingDescription structure, featuring fields specifying the binding ID and object size in bytes. The inputRate field accommodates instanced rendering, a concept elaborated on in subsequent lessons.

Individual attributes are defined within elements of an array of `VkVertexInputAttributeDescription` structures. Each attribute definition includes specifications for both its binding and location IDs, along with a constant denoting its data type (format). Lastly, the byte offset within the data structure for the respective field must be provided, computable using the C++ `offsetof()` macro.

To synchronize global variables with corresponding vertex attributes, the Vertex Shader employs the `layout(location)` directive.

### 8.3.2 Vertex and fragments shaders

In Vulkan, only the vertex shader has access to attributes. These values must be passed to other pipeline components using out variables. Vertex attributes, communicated through in and out variables, are organized into slots, each identified by a location number starting from zero. These location numbers are constrained by a hardware-dependent constant, typically sufficient for standard applications. When defining in or out variables, the user specifies the slot's location ID in a layout directive. Notably, only slot numbers are utilized, allowing the corresponding global variable names to vary between shaders.

The Input Assembler configures the slots utilized by the Vertex Shader's in variables, which are available for communication. Additionally, pipeline configuration dictates the out variables written by the Fragment Shader. Typically, this includes the final pixel color, but advanced applications may compute additional values. Communication between the Vertex and Fragment shaders adheres to their GLSL specifications. The pipeline's fixed functions interpolate out variable values from the Vertex Shader based on pixel positions on the screen before passing them to the Fragment Shader.

By default, interpolation between Vertex and Fragment shaders employs Perspective Correct techniques. However, this behavior can be modified using directives like flat and noperspective preceding in and out variables.

### 8.3.3 Uniform buffers

When we introduced GLSL, we discussed how applications can send scene- and mesh-dependent data to shaders using Uniform Blocks as global variables. This approach is also used for passing textures to shaders. Uniform blocks are accessed using a two-level indexing system.

Some shader parameters are dependent on the scene, and each shader requires its own pipeline along with specific parameters. In certain scenarios, the parameters that configure a Bidirectional Reflectance Distribution Function (BRDF) are referred to as materials. Depending on the shader, each material requires specific settings. These settings may be shared across multiple objects, and to optimize GPU performance, meshes with identical material settings are often grouped and rendered sequentially.

In addition to shared material properties, each mesh also has its own unique attributes, which shaders use to render their triangles.

In Vulkan, uniform variables are organized into Sets, with each Set representing a level of update frequency. Each Set is identified by an ID, starting from 0, with lower IDs assumed to change less frequently. A single Set can contain various resources, such as:

- Uniform blocks serving different purposes (e.g., light definitions, environmental properties).
- Textures.

- Other types of data.

Resources within a Set are identified by a secondary index called the Binding, which also starts from zero. Multiple resource types can be accessed as global Uniform Variables.

In this context, three key concepts are crucial:

- Descriptor Set Layouts.
- Descriptor Sets.
- Pipeline Layouts.

**Descriptor Set Layouts** In object-oriented programming (OOP) terms, Descriptor Set Layouts can be thought of as the "class" of uniform variables. They define the following:

- The type of descriptors (e.g., uniform buffer, texture image).
- The binding ID associated with each descriptor.
- The shader stage(s) in which the descriptors will be used (e.g., Vertex Shader, Fragment Shader, or both).

Descriptor Layouts within the same set, but with different bindings, are specified in an array of `VkDescriptorSetLayoutBinding` structures. Each binding in this array defines an integer ID starting from zero, the descriptor type (e.g., Uniform, Texture sampler), and the Shader Stage(s) that can access it.

Uniform blocks can be defined as arrays containing multiple elements. Additionally, if a texture is consistent across all pipelines in which it appears, certain optimizations may be possible. However, these advanced topics are beyond the scope of this course.

The `VkDescriptorSetLayout` objects are created using the `vkCreateDescriptorSetLayout` function. This function takes a `VkDescriptorSetLayoutCreateInfo` structure as an argument, which includes a pointer to the binding array and the number of elements in the array.

**Descriptor Sets** In object-oriented programming (OOP) terms, Descriptor Sets are instances of uniform data. They define the actual values that will be passed to the uniforms. For example, different meshes using the same material but requiring distinct world matrices will access different Descriptor Sets, each associated with the same Descriptor Layout.

**Pipeline Layout** The Pipeline Layout specifies which Descriptor Layouts will be accessed by the shaders in a given pipeline. It also defines the Set IDs where these descriptors will be found in the shaders. Descriptor sets are grouped into an array and provided in the `pSetLayouts` field of the `VkPipelineLayoutCreateInfo` structure, which is used to create the `VkPipelineLayout` with the `vkCreatePipelineLayout` function. The number of sets in the array is indicated by the `setLayoutCount` field.

The position of each layout in the array corresponds to the Set ID that the shader code will use to access the associated Descriptor Set.

**Descriptor Pools** Descriptor sets must be allocated from a Descriptor Pool, similar to how Command Buffers are allocated. However, this process is somewhat more complex because an accurate estimate of the number of sets is required.

A Descriptor Pool is defined by a set of `VkDescriptorPoolSize` objects, each specifying the type and quantity of descriptors (using the `descriptorCount` field). This array of descriptor requests is used to populate a `VkDescriptorPoolCreateInfo` structure, which also requires specifying the maximum number of descriptor sets that the application will use.

Determining the correct number of descriptors and descriptor sets needed by an application can be challenging, as it depends heavily on the structure of the rendering engine. The number should be equal to the sum of the different Descriptor Sets and the elements of each type used in the application.

Descriptor Pools are necessary for allocating Descriptor Sets using the `vkAllocateDescriptorSets()` function, with the allocation details provided in a `VkDescriptorSetAllocateInfo` structure. The function returns an array of `VkDescriptorSet` handles, each representing a descriptor set instance.

Descriptor Sets instantiate the Descriptor Layouts: typically, at least one Descriptor Set is needed for each unique value assigned to a Uniform. For Uniforms that change with the Scene, one set per scene is required; for those that change with the material, one set per material, and so on.

The method for linking Descriptor Set handles to their corresponding objects varies depending on the type of descriptor. This process shares similarities with defining vertex layouts.

### 8.3.4 Descriptor Buffers

First, a C++ data structure is created to store the variables that need to be sent to the shader. Instances of this structure reside in CPU memory (i.e., RAM). To be accessible within the shader, this data must be transferred to GPU-accessible memory (i.e., VRAM).

GPU memory may have specific alignment requirements that must also be respected in the C++ structure. This alignment can be managed using the `alignas()` C++ keyword.

Memory buffers are used to store and retrieve information from GPU-accessible video memory. They are characterized by two handles: a `VkBuffer` that identifies the buffer as a whole, and a `VkDeviceMemory` object that describes the allocated memory.

Once the descriptors have been set up, the application can update them in three steps:

1. Acquire a pointer to a memory area where the CPU can write the data, using the `vkMapMemory()` command.

2. Fill this memory area with the new values, typically using the `memcpy()` function.

3. Trigger the update of the video memory by calling `vkUnmapMemory()`.

### 8.3.5 Binding and Textures

Shaders must match the data types and order of the corresponding CPU objects. Additionally, they should use the same Set and Binding IDs defined in the application.

Textures are passed to shaders through special Layout Bindings within Set Layouts. When creating the Descriptor Pool, a request for the Combined Image Sampler must be included to support textures.

The actual texture pointer is specified when creating the Descriptor Set object associated with the Descriptor Set Layout. The `VkWriteDescriptorSet` structure, in addition to specifying the correct binding, indicates that this descriptor is a Combined Image Sampler.

Textures are passed to shaders as specific uniform variables of the Combined Texture Sampler type. To sample a texture, the shader uses the `texture()` function. The first parameter of this function specifies the image, while the second parameter defines the texel coordinates (format dependent on the sampler type). The function returns a `vec4` color, where the last component represents the alpha channel (transparency).

### 8.3.6 Texturing

Vulkan allows specifying a texture and its sampler in two distinct uniforms. This option is not covered in this course, so we will not delve into it further.

Each Graphics Adapter has its own internal format for images. Texture sampling, which involves numerous interpolations and floating-point operations, is managed by special hardware blocks on the GPU called Texture Units. The source image provided by the application usually differs from the format used internally by the GPU. Therefore, the input texture data must be converted into a format that the GPU and its Texture Units can process more efficiently.

In OpenGL and other graphics systems, this format conversion was handled automatically by the drivers without requiring special developer intervention. Vulkan, however, requires developers to explicitly manage this process.

The texturing stages are:

- *Loading*: The first step in defining a texture is loading the image into a memory area accessible by the CPU. This can be done using libraries such as `stb_image`. Specifically, the `stbi_load()` function returns an array of RGBA pixels along with the texture's size information. Size measurements can help determine other important details such as memory requirements and the number of Mip-Map levels.

- *Staging Buffer*: The next step is to move the texture into a memory area accessible by the GPU, known as the Staging Buffer. After the texture is placed in the staging buffer, the original memory area can be released. Finally, the texture must be transferred to its designated memory area, and its format converted to one recognized by the GPU.

**Mip-mapping** In older graphics engines like OpenGL, Mip-Map generation was handled by the driver, with the option for the developer to provide the Mip-Map layers directly. In Vulkan, however, the entire process of Mip-Map generation must be managed by the user.