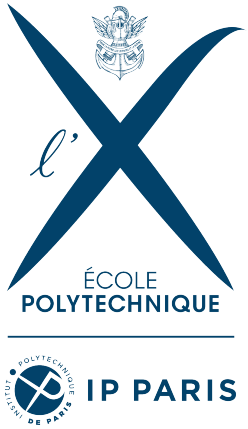
Version du March 14, 2026



[Graphics Part] C++ Programming

Application to Computer Graphics

[CSC-43043-EP]

2026

Damien ROHMER - damien.rohmer@polytechnique.edu

Contents

1 Introduction to Computer Graphics

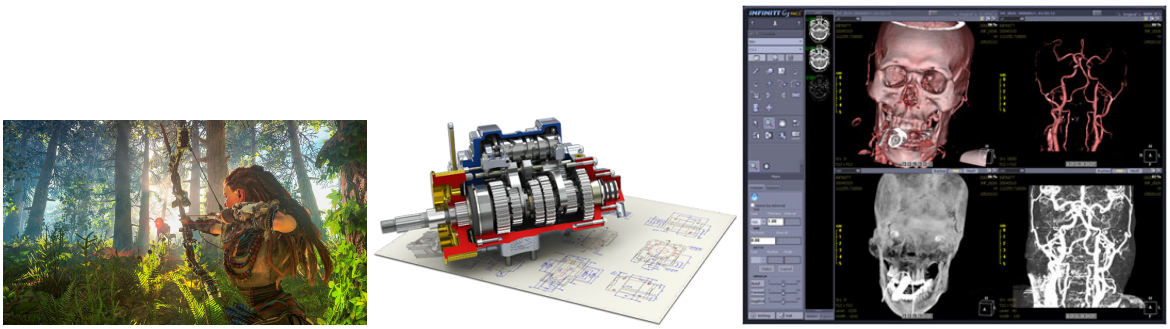
1.1 General Overview

Context

3D models are ubiquitous in many fields:

- **Entertainment applications**: cinema and VFX, video games, virtual reality.
- **Other fields**: CAD (Computer-Aided Design), medical imaging, physical simulation.

Some examples of these applications are presented below.



Examples of 3D applications: video game (left), CAD (center), medical imaging (right).

However, creating and manipulating 3D content remains a complex and costly task:

- 3D modeling tools (Maya, Blender, 3DS Max, etc.) have improved but remain very technical (approximately 3 years of training for a CG artist). The Blender interface, shown in the following figure, gives an overview.

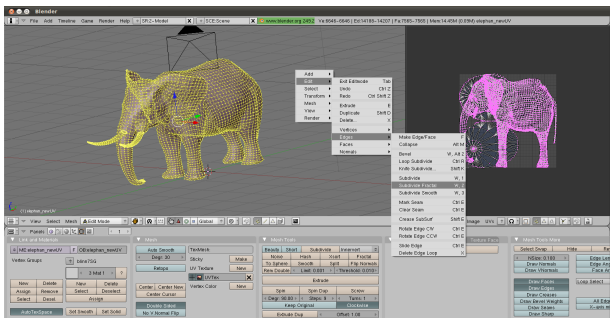


Figure 1: Interface of the 3D modeling software Blender.

- The quantity and quality of content demanded have increased faster than the capabilities of the tools.
- Production costs have never been higher: an animated film or a AAA video game can cost over 100 million dollars and mobilize hundreds of developers and thousands of artists over several years.

Automatic tools based on generative AI are starting to emerge, but remain very difficult to control and have a significant environmental cost. They are still largely experimental for real productions. Controllability, quality, and efficiency remain major challenges.

Definition of Computer Graphics

Computer graphics refers to all the sciences and techniques for generating and manipulating visual data:

- Efficiently in terms of time, energy, and memory.
- In a controllable and interactive manner.
- While enabling the generation of high-quality images.

The applications are numerous: entertainment, design, simulation of natural, medical, and biological phenomena, etc.

Computer graphics is structured around three main areas:

1. **Modeling**: creation and representation of 3D shapes. This covers the design of object geometry (their shape, their structure) as well as their visual attributes (textures, materials). Modeling can be manual (an artist using software like Blender or Maya) or procedural (algorithmic generation of shapes).
2. **Animation**: setting objects and characters in motion over time. Animation encompasses both kinematic techniques (explicit displacement of objects frame by frame or by interpolation of key poses) and physical techniques (simulation of forces, gravity, collisions, fluids, cloth, etc., which generate motion automatically from physical laws).
3. **Rendering**: generation of 2D images from a 3D scene. Rendering takes as input the description of geometry, materials, lights, and the camera, and then computes the resulting image. A distinction is made between **real-time** rendering (interactive, typically $\geq$ 25-60 frames per second, used in video games and visualization) and **offline** rendering (photo-realistic, which can take from a few seconds to several hours per image, used in cinema and architecture).

These three areas are illustrated below.

Modeling (left), animation (center), and rendering (right).

Related Topics and Vocabulary

Sub-topics

- **Virtual Reality (VR)**: interaction with a 3D scene using a virtual reality headset and sensors.
- **Augmented Reality (AR)**: overlaying virtual elements on a view of the real world.
- **Mixed / Extended Reality (XR)**: combination of VR and AR.
- **(Scientific) Visualization**: visual representation of data for explanation or communication purposes.
- **3D Simulation**: animation method based on a physics-inspired model.
- **CAD (Computer-Aided Design)**: design creation using 3D modelers.

Other Vocabulary

- **Computer graphics (applied)**: use of software for creating visual data (Maya, Blender, 3DS Max, etc.).
- **Image analysis / processing**: extraction of information from images. It is in some sense the opposite of image synthesis.
- **Computer Vision**: analysis of images or videos to understand a scene as a human would (shape recognition, motion recognition, etc.).

1.2 Concepts of a 3D Scene

Description of a 3D Scene

A 3D scene is described by three fundamental components:

- **3D Model**: a surface or a volume representing the objects in the scene. Each object has a geometry (shape) and visual attributes (color, material, texture). The scene can contain one or more objects, each positioned and oriented in a common 3D space called **world space**.
- **Light source**: one or more sources illuminating the scene. A light source is characterized by its position (or direction), its intensity, and its color. Common types include point lights (emission from a point), directional lights (parallel rays, simulating the sun), and area lights (emission from a surface, for softer shadows). The interaction between light and object materials determines the visual appearance of the scene.
- **Camera**: the viewpoint from which the scene is observed. The camera is defined by its position in space, its orientation (viewing direction), and its optical parameters (field of view, focal length, aspect ratio). It determines which objects are visible and how they are projected onto the final image.

The **output** of the rendering process is a **2D image** corresponding to what the camera ”sees” of the 3D scene. The following diagram summarizes the arrangement of these components.

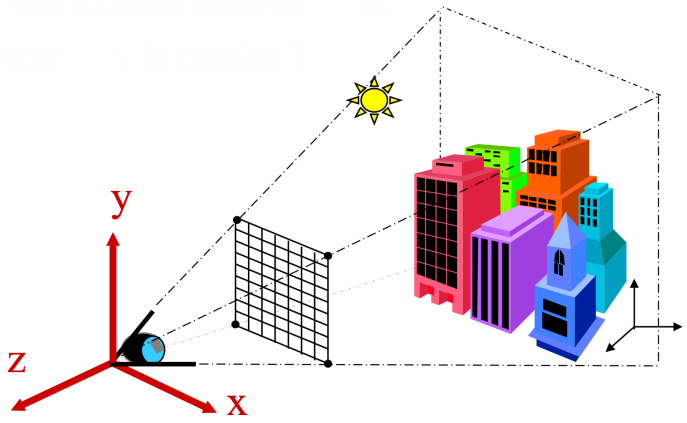


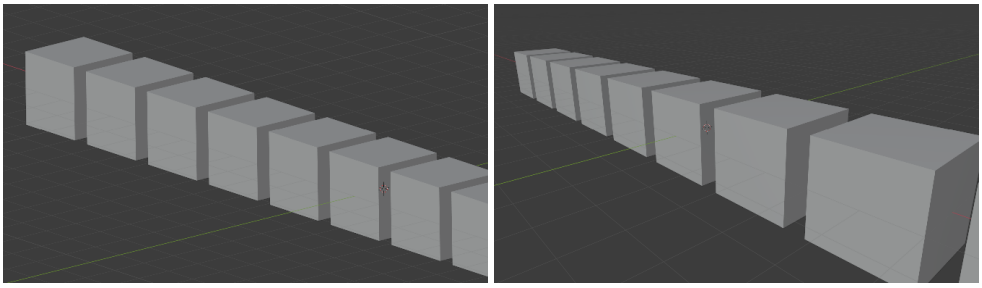
Figure 2: Diagram of a 3D scene: objects, light, and camera in world space.

Perception of a 3D Scene on an Image

A fundamental challenge of computer graphics is to create the illusion of depth and volume on a screen that is inherently a 2D medium. Several visual phenomena contribute to this perception.

Perspective Effect

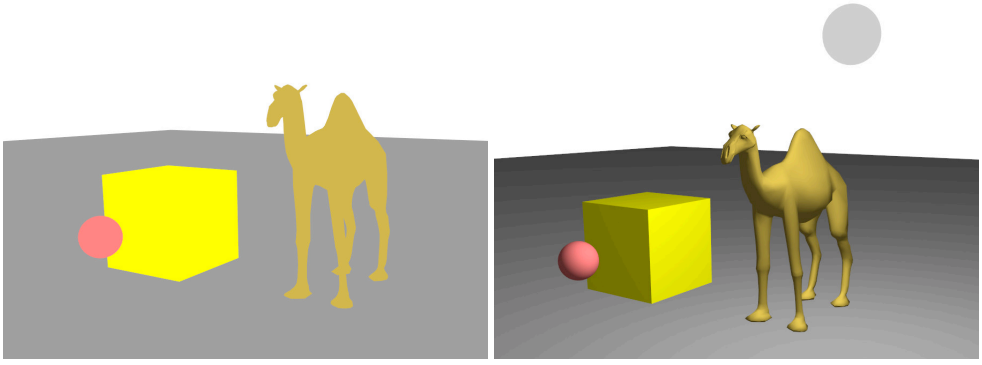
Distant objects appear smaller than nearby objects. Two parallel lines in the scene converge toward a vanishing point on the image. This effect is directly related to the perspective projection performed by the camera (see the section on generalized coordinates).



Orthographic projection (left) vs perspective projection (right).

Illumination and Shading

The color of a point on a surface varies depending on the orientation of that surface relative to the light source. A surface facing the light appears bright, while a surface turned away appears dark. This gradient of brightness, called **shading**, provides essential cues about the curvature and volume of objects. In contrast, an object displayed with a uniform color (without shading) appears flat, like a disk rather than a sphere.



Uniform color (left) vs diffuse illumination (right): shading reveals the geometry.

Occlusion

Nearby objects hide objects located behind them. This simple but powerful phenomenon provides a direct cue about the depth ordering of objects in the scene.

Cast Shadows

The shadow that an object casts on another object or on the ground visually anchors the object in the scene and provides information about its relative position and the direction of the light.

Definition of an Image

An image is a **2D grid of pixels** of dimension $N_x \times N_y$.

- Each pixel corresponds to a color.

- The standard format is **RGB** (Red, Green, Blue), sometimes extended to **RGBA** (with an alpha channel for transparency).
- In floating-point: $c = (r, g, b)$ with $r, g, b \in [0, 1]$.
  - Examples: $(0, 0, 0)$ = black, $(1, 0, 0)$ = red, $(1, 1, 0)$ = yellow, $(1, 1, 1)$ = white.
  - This is an **additive** color format.

Other common formats:

- **RGB in integers**: $(r, g, b) \in [\![0, 255]\!]^3$.
- **CMYK**: Cyan, Magenta, Yellow, Black (used for printing).
- **HSV**: Hue, Saturation, Value.
- **YPbPr**: Luminance, Blue Chrominance, Red Chrominance.
- **CIELAB**: perceptually uniform color space.

Note: the RGB space is not perceptually uniform (two colors that are close in RGB distance are not necessarily visually close). The RGB cube is shown in the figure below.

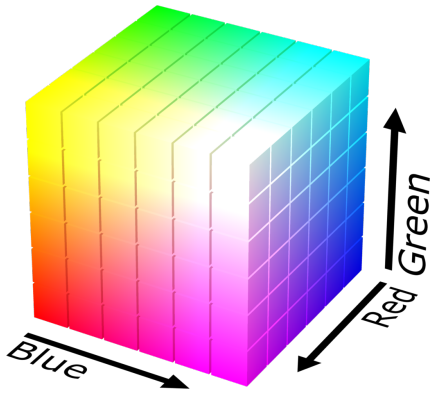


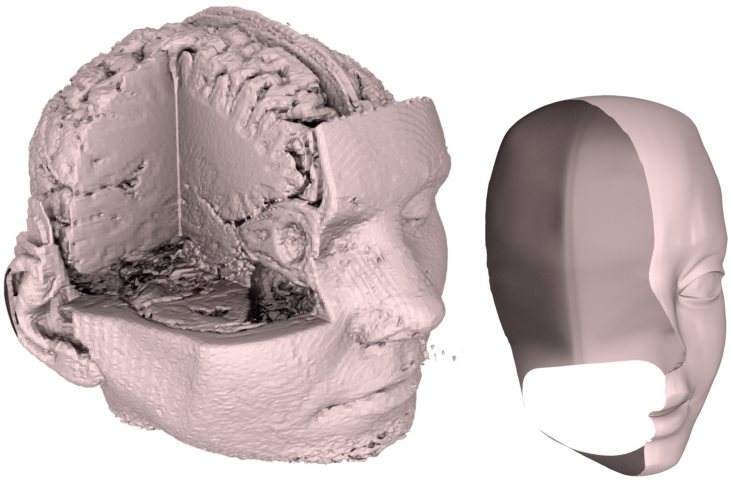
Figure 3: Visualization of the RGB color cube.

Representation of 3D Models

A 3D object can be represented in two fundamental ways:

- **Volume**: complete description of the object, including its interior. One can, for example, associate a density, a temperature, or any other physical attribute at each point of the volume. This representation is essential in medical imaging (MRI, CT scan), fluid simulation (smoke, clouds), and materials physics. However, it is very memory-intensive: a volume of resolution 256^3 already contains over 16 million voxels (volumetric pixels).
- **Surface**: only the outer, visible part of the object. Only the "skin" of the object is described, without information about the interior. This representation is much more memory-efficient and, more importantly, much more efficient for GPU rendering, since only the surface contributes to the final image.

In real-time computer graphics, **surface** representations are predominantly used. Volumetric representations are reserved for specific cases (atmospheric effects, medical data, physical simulation). The following figure compares these two approaches.



Volumetric representation (left) vs surface representation (right).

Explicit and Implicit Surfaces

There are two fundamental families of mathematical surface descriptions:

- **Explicit (parametric) representation**: the surface is described by a function that maps each pair of parameters (u, v) to a 3D point on the surface: $S(u, v) = (x(u, v), y(u, v), z(u, v))$.
  - Advantage: the notion of neighborhood is natural (two points close in (u, v) are close on the surface), differential calculus is straightforward (tangents and normals can be computed by differentiating S).
  - Disadvantage: it is difficult to represent surfaces with complex topology (with holes, branches) using a single parametric function.
  - Example (sphere of radius R):

$$S(u, v) = (R \sin u \cos v,\ R \sin u \sin v,\ R \cos u), \quad u \in [0, \pi],\ v \in [0, 2\pi]$$

- **Implicit representation**: the surface is defined as the zero set of a scalar field:

$$S = \{(x, y, z) \in \mathbb{R}^3 \mid f(x, y, z) = 0\}$$

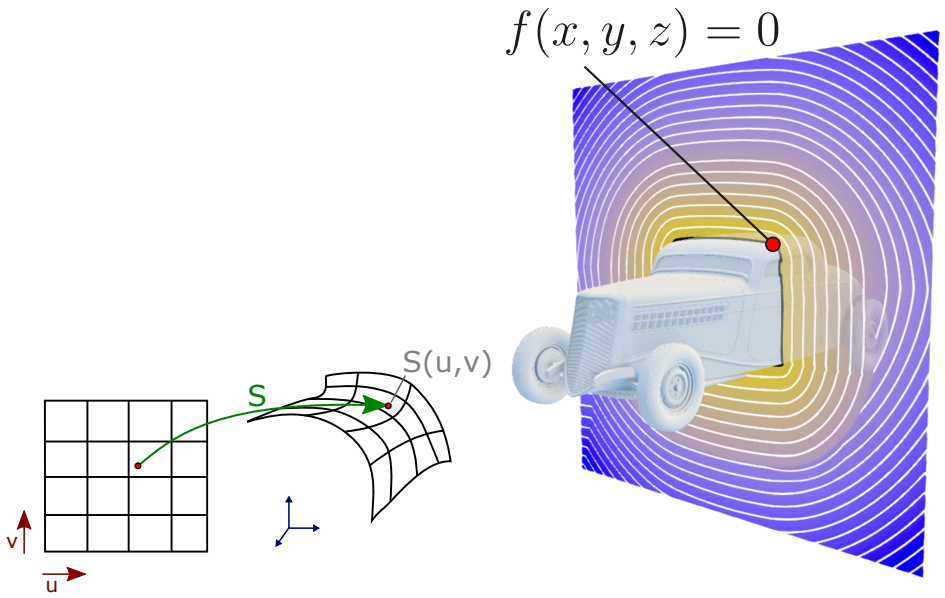
  The function f assigns a value to each point in space; the surface is the locus where this value is zero.

  - Advantage: natural topological adaptation. When two implicit objects come close together, their isosurfaces can automatically merge or separate without manual intervention. This is very useful for simulating fluids, meta-matter, or Boolean operations (union, intersection, difference of shapes).
  - Disadvantage: it is more difficult to traverse the surface (no natural parameterization) and direct rendering is more costly.
  - Example (sphere of radius R centered at the origin):

$$f(x, y, z) = x^2 + y^2 + z^2 - R^2$$

    Points where $f < 0$ are inside the sphere, those where $f > 0$ are outside.

The two types of representation are compared in the diagram below.



Explicit parametric surface (left) vs implicit surface (right).

In practice, for complex shapes like a character or a vehicle, no simple analytical formula can describe the surface. One then resorts to discrete approximations.

Discretization and Primitives

In practice, arbitrary surfaces are rarely described by a single analytical function. **Piecewise approximations** using primitives and discrete elements are used.

The ideal properties of a description are:

- Accurate approximation of arbitrary surfaces.
- Small number of primitives.
- Efficient GPU rendering.
- Easy manipulation for modeling.

Examples of models:

- **Meshes**: triangular or polygonal meshes.
- **Polynomials**: Bezier curves and surfaces, Splines, NURBS.
- **Implicit**: voxels, skeletons, RBF, MLS.
- **Point sets**, **Gaussian Splats**.

Graphics cards are specialized for rendering **triangle meshes**.

1.3 3D Rendering

Two Rendering Approaches

Ray Tracing

Ray tracing simulates the physical path of light in the scene. The basic principle is as follows: for each pixel of the image, a **ray** is cast from the camera through that pixel to find the first object in the scene that the ray intersects.

Once the intersection point is found, its color is computed based on the lighting, the material, and potentially reflections and refractions (by recursively casting new rays).

In practice, rays are often traced in the reverse direction of light (from the camera toward the scene), because the vast majority of rays emitted by a light source never reach the camera. This reverse tracing is much more efficient.

Advantages:

- **Photo-realistic** rendering: soft shadows, reflections (mirrors), refractions (glass, water), caustics, and global illumination (indirect light bouncing between surfaces) are handled naturally by the model.
- Compatible with **arbitrary surfaces**: the only prerequisite is being able to compute the intersection of a ray with the surface, which is possible for many representations (triangles, spheres, implicit surfaces, etc.).

Disadvantages:

- High **computational cost**: each pixel potentially requires many intersection calculations and the number of secondary rays (reflections, shadows) can grow exponentially. For a high-quality image (cinema), rendering a single image can take from a few minutes to several hours.
- Not well suited for real-time rendering, although recent GPU advances (hardware ray tracing) are beginning to make it possible in some cases.

The principle is summarized in the following diagram.

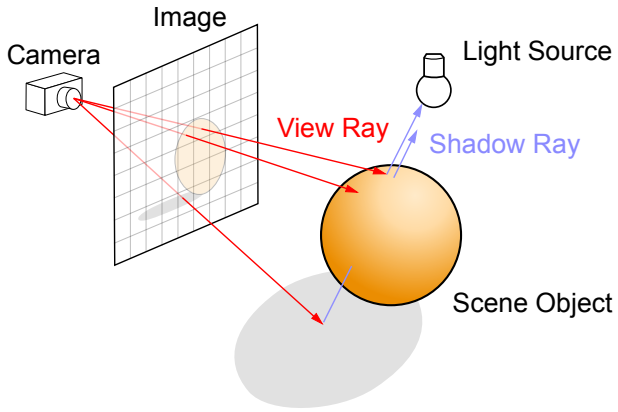


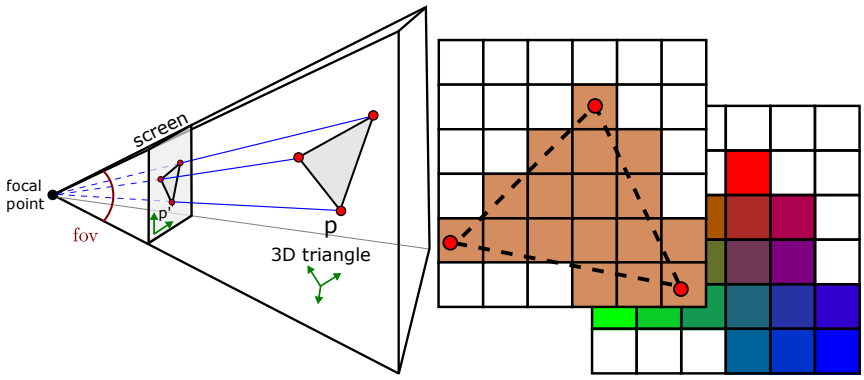
Figure 4: Principle of ray tracing.

Projection / Rasterization

The rasterization approach adopts a radically different strategy from ray tracing. Instead of starting from each pixel and finding which object is visible, one starts from each object (triangle) and determines which pixels it covers. This approach assumes that the scene is composed of **triangles**. The process takes place in two steps:

1. **Projection**: each triangle vertex is projected from 3D space onto the camera's 2D plane. This operation is performed by a matrix multiplication in projective space: $p' = M\,p$, where M is the matrix combining the scene transformation, the camera placement, and the perspective projection. This operation is extremely fast because it reduces to a matrix-vector product per vertex.
2. **Rasterization**: once the triangles are projected in 2D, each triangle is ”filled” pixel by pixel. For each pixel covered by the triangle, its barycentric coordinates are computed and the vertex attributes (color, normal, texture coordinates, etc.) are interpolated to obtain the value at the pixel. Each pixel thus processed is called a **fragment**.

These two steps are illustrated below.



Projection of vertices onto the camera plane (left) and rasterization of triangles into pixels (right).

A **z-buffer** (depth buffer) mechanism handles occlusion: for each pixel, only the fragment closest to the camera is kept, ensuring that objects in front occlude those behind.

Advantages:

- A dedicated approach massively optimized on **GPU**. Graphics cards are specifically designed to accelerate this pipeline.
- Rendering at interactive speeds ($\geq$ 25-60 fps), and even well beyond on modern GPUs.

Disadvantages:

- Limited to **triangles** (or to primitives that decompose into triangles).
- No natively physically correct effects: shadows, transparency, reflections, and indirect lighting require additional techniques (shadow maps, environment maps, screen-space effects, etc.) which are approximations.

Rasterization is the **standard for real-time rendering** on GPU. It is the approach used in all video games, interactive visualization applications, and CAD software.

1.4 Triangle Meshes

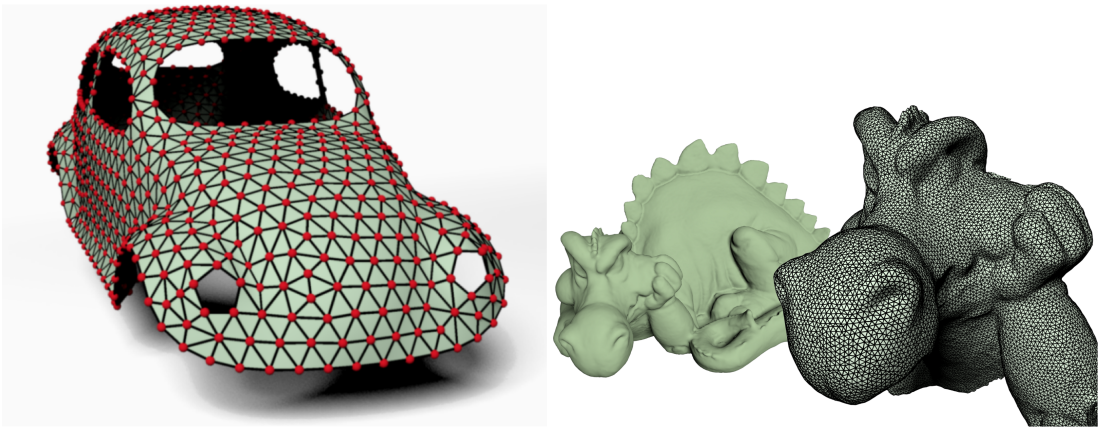
Structure

A triangle mesh is the simplest and most widespread representation for approximating a continuous surface with a set of triangles. The higher the number of triangles, the more faithful the approximation to the original surface, but the greater the storage and rendering cost. In practice, a typical 3D model in a video game contains from a few thousand to several hundred thousand triangles. A high-resolution model for cinema can contain several million triangles.

A mesh is described by three types of elements:

- **Vertices**: the 3D points that form the corners of the triangles. The set of vertices is denoted $\mathcal{V} = (p_1, \ldots, p_{N_v})$ with $p_i = (x_i, y_i, z_i) \in \mathbb{R}^3$. Each vertex can carry additional attributes: a normal (direction perpendicular to the surface at that point), a color, texture coordinates, etc.
- **Faces**: each face is a triangle defined by a triplet of vertex indices $\mathcal{F} = (f_1, \ldots, f_{N_f})$ with $f_i = (p_{i_1}, p_{i_2}, p_{i_3})$. The faces define the **connectivity** (or topology) of the mesh, that is, how the vertices are connected to each other.
- **Edges** (optional): segments connecting two adjacent vertices $\mathcal{E} = (e_1, \ldots, e_{N_e})$ with $e_i = (p_{i_1}, p_{i_2})$. Edges are not always explicitly stored because they can be derived from the faces, but they are useful for certain algorithms (subdivision, simplification, etc.).

Concrete examples of meshes are shown below.



Examples of triangle meshes: car model (left) and dragon model (right).

Properties of Triangles

A triangle T is defined by three vertices (p_1, p_2, p_3). The triangle is the basic element of rasterization and has very useful mathematical properties that we detail below.

Parameterization

Any point p inside the triangle can be expressed as a linear combination of the two edges originating from p_1:

$$p \in T \Leftrightarrow S(u,v) = p_1 + u\,(p_2 - p_1) + v\,(p_3 - p_1), \quad u \geq 0,\ v \geq 0,\ u + v \leq 1$$

The parameters (u, v) form a local coordinate system on the triangle. The vertex p_1 corresponds to $(0, 0)$, p_2 to $(1, 0)$, and p_3 to $(0, 1)$.

Barycentric Coordinates

An equivalent and often more practical way to parameterize a triangle is to use **barycentric coordinates** (α, β, γ):

$$p \in T \Leftrightarrow p = \alpha\, p_1 + \beta\, p_2 + \gamma\, p_3, \quad (\alpha, \beta, \gamma) \in [0,1],\ \alpha + \beta + \gamma = 1$$

Barycentric coordinates have an intuitive geometric interpretation: α represents the "weight" or influence of vertex p_1 on the point p. The closer p is to p_1, the larger α is (close to 1) and the smaller β and γ are. At the vertices, we have respectively $(\alpha, \beta, \gamma) = (1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. At the centroid of the triangle, all three coordinates equal $1/3$.

The computation of barycentric coordinates for a point p is done via the ratio of sub-triangle areas:

$$\alpha = \frac{A_{p23}}{A_{123}}, \quad \beta = \frac{A_{p31}}{A_{123}}, \quad \gamma = \frac{A_{p12}}{A_{123}}$$

with $A_{123} = \frac{1}{2}\|(p_2 - p_1) \times (p_3 - p_1)\|$ the total area of the triangle, and A_{p23}, A_{p31}, A_{p12} the areas of the sub-triangles formed by p with the opposite edges:

$$A_{p23} = \tfrac{1}{2}\|(p_2 - p) \times (p_3 - p)\|$$
$$A_{p31} = \tfrac{1}{2}\|(p_3 - p) \times (p_1 - p)\|$$
$$A_{p12} = \tfrac{1}{2}\|(p_1 - p) \times (p_2 - p)\|$$

This geometric construction is illustrated in the figure below.

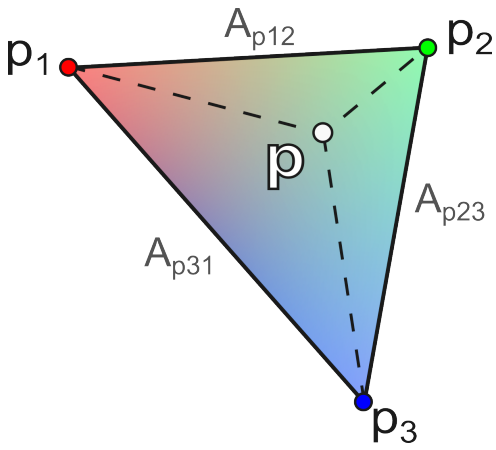


Figure 5: Barycentric coordinates: the point p is located by the areas of the sub-triangles.

Applications

Barycentric (Linear) Interpolation

Let a triangle T with colors (c_1, c_2, c_3) associated with the vertices (p_1, p_2, p_3). The interpolated color at a point p inside the triangle is:

$$c(p) = \alpha\, c_1 + \beta\, c_2 + \gamma\, c_3$$

where (α, β, γ) are the barycentric coordinates of p.

This interpolation is fundamental in rendering: it is used to interpolate colors, normals, texture coordinates, etc., inside each triangle. An example of color interpolation is shown below.

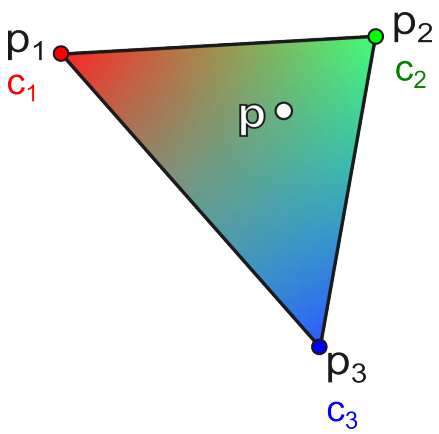


Figure 6: Barycentric interpolation of colors inside a triangle.

Intersection Test

To test whether a point p is inside a triangle $T = (p_1, p_2, p_3)$:

1. **Necessary condition**: p must be in the plane of the triangle. We verify that $(p - p_1) \cdot n = 0$ with $n = (p_2 - p_1) \times (p_3 - p_1)$ (triangle normal).
2. **Sufficient condition**: computation of barycentric coordinates (α, β, γ). We verify that $0 \leq \alpha, \beta, \gamma \leq 1$ and $\alpha + \beta + \gamma = 1$.

The principle of this test is shown in the following diagram.

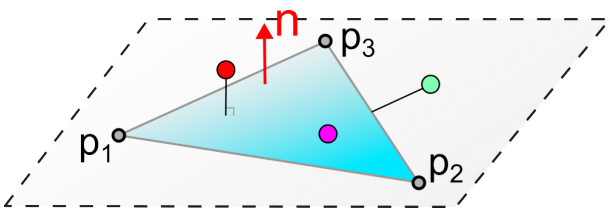


Figure 7: Ray-triangle intersection test.

Connectivity Description

Let us consider a tetrahedron (p_0, p_1, p_2, p_3) as an example.

1st solution: triangle soup

Each triangle is described by its three coordinates, without vertex sharing:

```
triangles = [(0.0,0.0,0.0), (1.0,0.0,0.0), (0.0,0.0,1.0),
             (0.0,0.0,0.0), (0.0,0.0,1.0), (0.0,1.0,0.0),
             (0.0,0.0,0.0), (0.0,1.0,0.0), (1.0,0.0,0.0),
             (1.0,0.0,0.0), (0.0,1.0,0.0), (0.0,0.0,1.0)]
```

2nd solution: geometry + connectivity (topology)

The vertex positions and face indices are separated:

```
geometry = [(0.0,0.0,0.0), (1.0,0.0,0.0), (0.0,1.0,0.0), (0.0,0.0,1.0)]
connectivity = [(0,1,3), (0,3,2), (0,2,1), (1,2,3)]
```

This second approach is much more memory-efficient because shared vertices are stored only once. In the tetrahedron example, the first solution stores $4 \times 3 = 12$ coordinate triplets (i.e., 36 floats), while the second stores only 4 vertices (12 floats) plus 4 index triplets (12 integers). For large meshes, the savings are considerable. This is the standard representation used in practice and by graphics APIs (OpenGL, Vulkan, etc.).

Note: the order of indices in each face defines the **orientation** of the face. By convention (known as the **right-hand rule** or **counterclockwise**), when looking at the face from outside the object, the vertices appear in counterclockwise order. This orientation allows computing the **outward-facing normal** of the face via a cross product: $n = (p_{i_2} - p_{i_1}) \times (p_{i_3} - p_{i_1})$. The GPU uses this information for **back-face culling**: triangles seen from behind (whose normal points away from the camera) are not displayed, which reduces the rendering cost by roughly half. This convention is illustrated below.

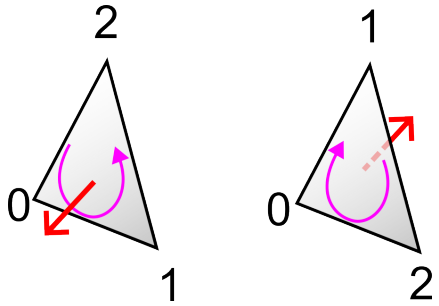


Figure 8: Face orientation convention (counterclockwise).

File Format: OBJ

The **OBJ** (Wavefront) format is one of the most common text formats for storing 3D meshes. Its simplicity makes it easy to read and write, both by humans and programs. An OBJ file is a text file where each line begins with a keyword followed by values:

- `v x y z`: defines a vertex with its 3D coordinates.
- `vt u v`: defines texture coordinates (for mapping an image onto the surface).
- `vn x y z`: defines a vertex normal (direction perpendicular to the surface, used for shading).
- `f v1 v2 v3`: defines a triangular face by its vertex indices. Indices start at **1** (not 0). One can also reference normals and textures with the syntax `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3`.

Minimal example (tetrahedron):

```
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 0.0 0.0 1.0
f 1 2 4
f 1 4 3
f 1 3 2
f 2 3 4
```

1.5 Generalized Coordinates

Affine Transformations

Affine transformations are the fundamental operations for positioning, orienting, and sizing objects in 3D space. In computer graphics, they are used constantly: to place an object in the scene, to animate a character (rotation of joints), to position the camera, etc.

Translation

Translation moves a point by a constant vector (t_x, t_y, t_z):

$$t(p) = (x + t_x,\ y + t_y,\ z + t_z)$$

Translation is **not** a linear transformation (it cannot be represented by a 3×3 matrix multiplication, since $t(0) \neq 0$ in general). This property is the reason for the need for homogeneous coordinates described later.

Scaling

Scaling multiplies each coordinate by a scale factor:

$$s(p) = (s_x\, x,\ s_y\, y,\ s_z\, z)$$

In matrix notation:

$$S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$$

If $s_x = s_y = s_z$, the scaling is **uniform** (the object keeps its proportions). Otherwise, it is **non-uniform** (the object is deformed, for example stretched along one axis).

Rotation

A rotation preserves distances and angles (it is an isometry). It is described by a 3×3 orthogonal matrix:

$$R = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}, \quad R\,R^T = I \text{ and } \det(R) = 1$$

The columns (and rows) of R form an orthonormal basis: they are mutually orthogonal and of unit norm. The condition $\det(R) = 1$ distinguishes rotations from reflections (which have a determinant of -1).

There are several ways to parameterize a rotation in 3D: Euler angles (three successive angles around the axes), axis-angle (an axis and a rotation angle around that axis), or quaternions (a 4-component representation, widely used in animation for its efficiency and absence of singularities).

Shearing

Shearing shifts one coordinate proportionally to another. For example:

$$sh_{xy}(p) = (x + \lambda\, y,\ y,\ z)$$

In matrix notation:

$$Sh_{xy} = \begin{pmatrix} 1 & \lambda & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Shearing preserves volumes ($\det(Sh) = 1$) but does not preserve angles (it is not an isometry). It is rarely used intentionally in computer graphics, but can appear as a byproduct of certain matrix decompositions. Its geometric effect is shown in the following figure.

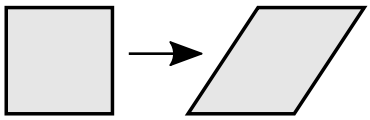


Figure 9: Effect of shearing on a square.

Homogeneous Coordinates

Rotation and scaling are **linear** transformations, representable by 3×3 matrices. Translation, on the other hand, is a **non-linear** transformation that cannot be directly encoded by a 3×3 matrix.

This poses a practical problem: in computer graphics, rotations, scalings, and translations are constantly chained together to position objects in the scene. For example, to place an object in the world, one might apply a scaling (to adjust its size), then a rotation (to orient it), then a translation (to position it). If only rotations and scalings were matrices, it would be necessary to maintain two separate representations (a matrix for the linear part and a vector for the translation), which would considerably complicate the code.

Idea: add an extra coordinate to obtain a **unified** 4D representation, in which **all** affine transformations (including translation) are expressed as matrix products.

- A 3D **point** (x, y, z) is represented by the 4D vector $(x, y, z, 1)$.
- A 3D **vector** (x, y, z) is represented by $(x, y, z, 0)$.

The unified affine transformation is then written as a 4×4 matrix product:

$$\begin{pmatrix} p' \\ 1 \end{pmatrix} = \begin{pmatrix} L & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ 1 \end{pmatrix} = \begin{pmatrix} L\,p + t \\ 1 \end{pmatrix}$$

with L the linear part (3×3) and t the translation vector.

Principle in 2D

For a point $p = (x, y)$, we add a coordinate: $p = (x, y, 1)$.

Translation becomes a linear operation:

$$\begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Similarly for rotations and scaling, which are expressed as 3×3 matrices in 2D homogeneous coordinates.

Extension to 3D

In 3D, 4-component vectors and 4×4 matrices are used.

Any sequence of translations, rotations, and scalings can be factored into a **single matrix product**:

$$M = T_0 \, R_0 \, S_0 \, T_1 \, R_1 \, S_1 \, \ldots$$

The advantage is considerable: regardless of the complexity of the transformation chain, the result is always a single 4×4 matrix. Applying this transformation to a point amounts to a single matrix-vector multiplication. This property is massively exploited by the GPU, which can transform millions of vertices by applying the same matrix M to each of them in parallel.

[Warning]: the order of multiplications matters. Matrix transformations are **not commutative** in general: $T\,R \neq R\,T$. By convention, the rightmost transformation is applied first. Thus, $M = T\,R\,S$ means: first scaling, then rotation, then translation.

Points and Vectors

The interest of generalized coordinates is the natural differentiation between **points** and **vectors**:

Point: $(x, y, z, \mathbf{1})$ -- translation applies.

$$M \begin{pmatrix} p \\ 1 \end{pmatrix} = \begin{pmatrix} L & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ 1 \end{pmatrix} = \begin{pmatrix} L\,p + t \\ 1 \end{pmatrix}$$

Vector: $(x, y, z, \mathbf{0})$ -- translation does not apply.

$$M \begin{pmatrix} v \\ 0 \end{pmatrix} = \begin{pmatrix} L & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v \\ 0 \end{pmatrix} = \begin{pmatrix} L\,v \\ 0 \end{pmatrix}$$

This behavior is consistent with geometric operations:

- vector $\pm$ vector $\rightarrow$ vector
- position $\pm$ vector $\rightarrow$ position
- position $-$ position $\rightarrow$ vector

This distinction is summarized in the diagram below.

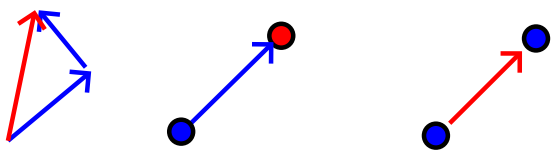


Figure 10: Distinction between points and vectors in generalized coordinates.

Projective Space

Let us consider the case of a generalized point with coordinate $w \neq 1$: $p_{4D} = (x, y, z, w)$.

The "renormalization" (or projection) onto the space of 3D points gives: $p_{3D} = (x/w, y/w, z/w, 1)$.

Example:

The sum of two points gives:

$$(x_1, y_1, z_1, 1) + (x_2, y_2, z_2, 1) = (x_1 + x_2,\ y_1 + y_2,\ z_1 + z_2,\ 2)$$

After homogenization:

$$p_{3D} = \left(\frac{x_1 + x_2}{2},\ \frac{y_1 + y_2}{2},\ \frac{z_1 + z_2}{2},\ 1 \right)$$

Which corresponds to the **centroid** of the two points.

The interest of projective space is to encode **rational** operations (such as perspective) through a simple matrix multiplication.

Application to Perspective

Perspective is modeled by a division by depth.

In 2D (1D projection), for a point (x, y) projected onto a focal plane at distance f:

$$y' = f\,\frac{y}{x}$$

This non-linear model can be written linearly in homogeneous coordinates:

$$\begin{pmatrix} f\,y \\ x \end{pmatrix} = \begin{pmatrix} 0 & f \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

After homogenization (division by the last component x), we indeed obtain $y' = f\,y/x$.

Real points (2D or 3D) are those whose last component is $w = 1$ (obtained after homogenization). **Vectors** correspond to $w = 0$. This projection mechanism is shown in the following figure.

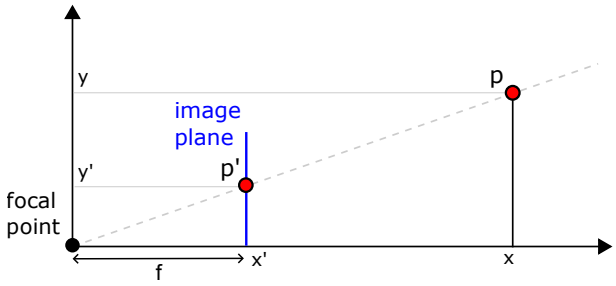


Figure 11: 2D perspective projection: the point is projected onto the focal plane by division by depth.

1.6 OpenGL and Shaders

CPU and GPU

A computer has two main processors for computation:

- **CPU (Central Processing Unit)**: the central processor, designed to execute complex and varied sequential tasks (computation, conditional branching, memory access, input/output, operating system management, etc.). The CPU has a small number of very powerful cores, each capable of executing arbitrary instructions independently. Associated memory: RAM.
  - Number of cores: 2 to 64 (independent, each with its own instruction pipeline).
  - RAM: 4 to 64 GB.
  - Optimized for **latency**: each individual task is executed as fast as possible.
- **GPU (Graphics Processing Unit)**: the graphics processor, designed to execute a very large number of **identical** operations in parallel. The architecture is **SIMD** (Single Instruction, Multiple Data): the same instruction is applied simultaneously to thousands of different data items. For example, applying the same transformation matrix to each of the millions of vertices in a mesh, or computing the color of each of the millions of pixels on the screen. Associated memory: VRAM (Video RAM).
  - Number of cores: 1000 to 16000 (organized in groups sharing resources).
  - VRAM: 4 to 24 GB.
  - Optimized for **throughput**: a very large volume of data is processed per unit of time, but each individual task may be slower than on CPU.

The figures below show the physical appearance and internal architecture of these two types of processors.

CPU (left) and GPU (right).

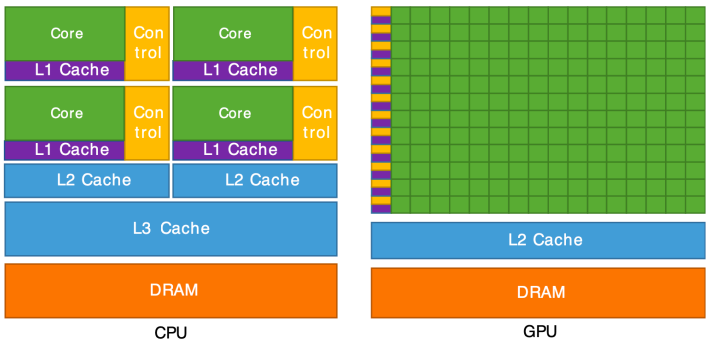
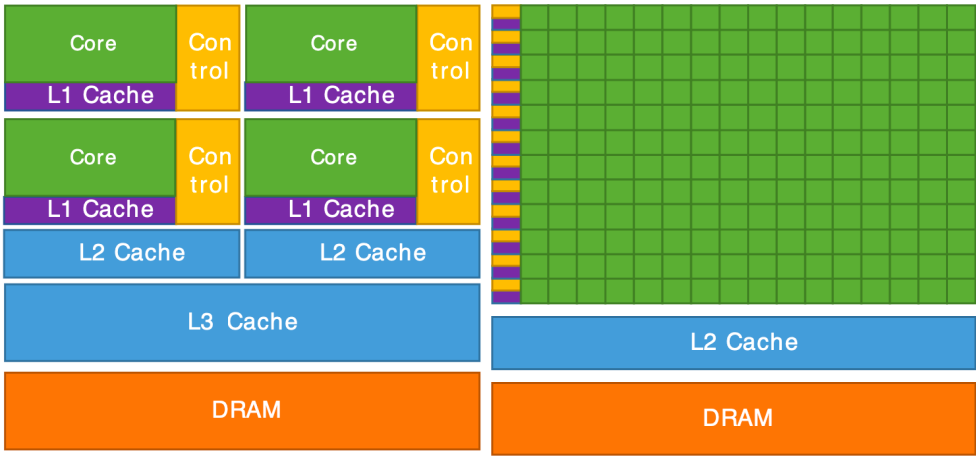


Figure 12: Comparison of CPU and GPU architectures.



Internal architecture of a CPU (left) and a GPU (right).

In summary, the GPU is a supercomputer optimized for performing the **same operation on a large number of data items** in parallel. This is exactly what is needed for graphics rendering: transforming millions of vertices (vertex shader) and then coloring millions of pixels (fragment shader). The CPU, on the other hand, handles the program logic (scene management, data loading, user interactions) and sends rendering commands to the GPU.

OpenGL

OpenGL (Open Graphics Library) is an API (Application Programming Interface) for communicating with the GPU, oriented toward 3D graphics.

Characteristics:

- Low-level, high-performance, cross-platform.
- An API is neither software nor a library: it is a standardized set of variables and functions. Different systems and GPUs have different implementations of OpenGL (installation depends on the graphics driver).
- Developed and maintained by the Khronos Group.

Other graphics APIs: **Vulkan**, **WebGL**, **WebGPU**, **DirectX** (Windows), **Metal** (Mac).

CPU/GPU Communication with OpenGL

The CPU and GPU each have their own memory (RAM and VRAM respectively) and cannot directly access each other's memory. The rendering process therefore requires explicit communication between the two, orchestrated by the OpenGL API. This process follows three main steps:

1. **Data preparation (CPU)**: the C++ program (running on the CPU) loads or computes the geometric data of the scene: vertex positions, colors, normals, texture coordinates, connectivity indices, etc. This data is organized in arrays in RAM.
2. **Data transfer (CPU → GPU)**: the data is transferred in blocks from RAM to VRAM via OpenGL calls. This transfer is relatively costly (the bus between CPU and GPU has limited bandwidth), which is why it should be minimized: ideally, static data (meshes that do not change) is sent only once at startup. Only data that changes every frame (transformation matrices, animation parameters) is transmitted each frame.
3. **Shader execution (GPU)**: once the data is in VRAM, the GPU executes shader programs in parallel on each vertex and then on each pixel. The CPU no longer intervenes during this phase: it simply sends the ”draw” command and the GPU does the rest autonomously. The result is the final image, stored in the VRAM **framebuffer**, which is then displayed on the screen.

The following diagrams detail this communication pipeline.

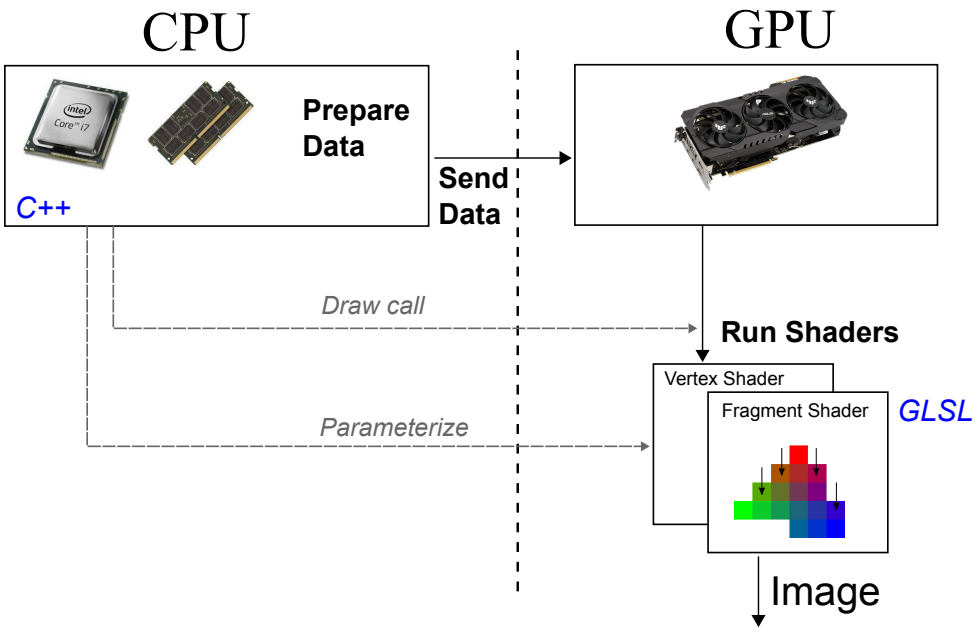


Figure 13: Simplified diagram of CPU/GPU communication via OpenGL.

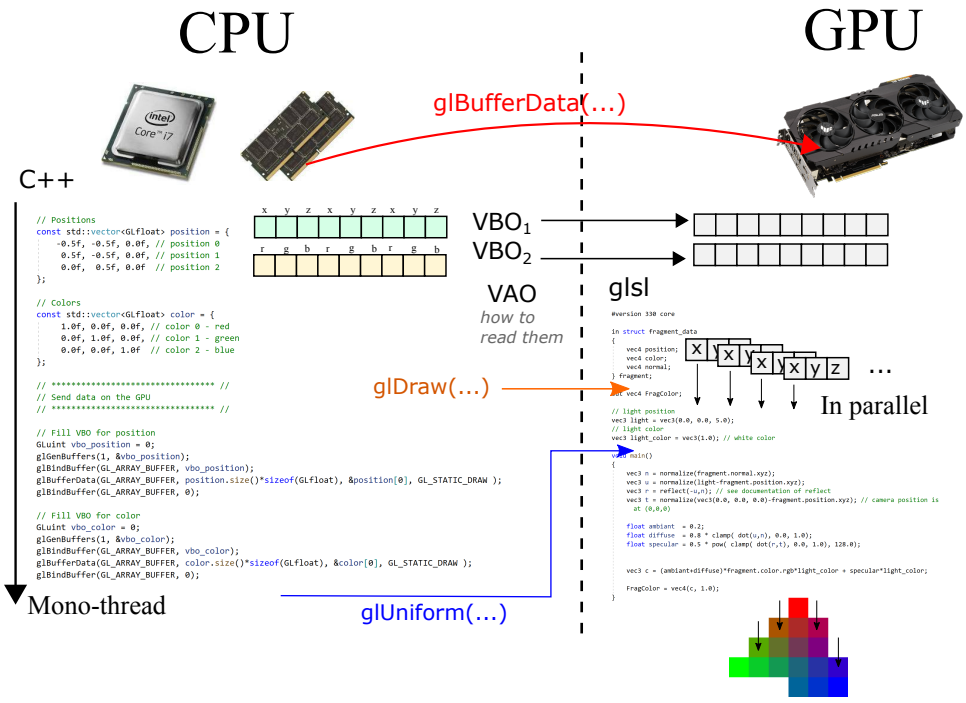


Figure 14: Overview of the OpenGL rendering pipeline.

Shaders

Shaders are small programs that run directly on the GPU. They are written in a dedicated language called **GLSL** (OpenGL Shading Language), whose syntax is close to C. The term ”shader” comes from ”shading”, because their original role was to compute surface shading, but they are now used for all kinds of graphics computations.

The fundamental characteristic of shaders is that they are executed **massively in parallel**: the same program is launched simultaneously on thousands of vertices or pixels. The programmer writes the code for **a single** vertex or pixel, and the GPU takes care of executing it for all of them.

Two main types of shaders are involved in the rendering pipeline:

Vertex Shader

Executed once **for each vertex** of the mesh. Its main role is to transform the vertex position from the 3D scene space to the 2D screen space (by applying the transformation and projection matrices). It can also compute and transmit attributes (colors, transformed normals, texture coordinates, etc.) that will be used by the fragment shader.

```
#version 330 core

layout(location = 0) in vec4 position;
layout(location = 1) in vec4 color;

out vec4 vertexColor;

void main()
{
    gl_Position = position;
    vertexColor = color;
}
```

In this example:

- `#version 330 core`: indicates the GLSL version used (corresponding to OpenGL 3.3).
- `layout(location = 0) in vec4 position`: declares an input attribute (the vertex position), received from the data sent by the CPU. The `location = 0` indicates the attribute buffer index.
- `out vec4 vertexColor`: declares an output variable that will be transmitted to the fragment shader. Between the vertex shader and the fragment shader, this variable will be automatically interpolated by rasterization.
- `gl_Position`: a special variable predefined by OpenGL, in which the vertex shader must write the final vertex position (in projected coordinates).

Fragment Shader

Executed once **for each pixel (fragment)** covered by a triangle after rasterization. Its role is to determine the final pixel color based on the interpolated attributes (color, normal, texture coordinates), lighting, textures, etc. It is in the fragment shader that illumination models and visual effects are implemented.

```
#version 330 core

in vec4 vertexColor;
out vec4 fragColor;

void main()
{
    fragColor = vertexColor;
}
```

In this example:

- `in vec4 vertexColor`: input variable coming from the vertex shader. Its value has been automatically interpolated by rasterization between the three vertices of the current triangle, using the barycentric coordinates of the fragment.
- `out vec4 fragColor`: the output color of the fragment, which will be written to the final image (framebuffer).
- Here, the fragment shader is minimal: it simply passes through the interpolated color. In practice, it is in the fragment shader that lighting is computed, textures are sampled, and visual effects are applied.

Uniform Variables

In addition to attributes (specific to each vertex) and interpolated variables (specific to each fragment), shaders can receive **uniform variables**: values that are constant for all vertices or fragments within a single draw call. They are defined on the CPU side and sent to the GPU. Typical examples are transformation matrices, the camera position, the light direction, the current time, etc.

```
uniform mat4 modelViewProjection; // transformation matrix (4x4)
uniform vec3 lightDirection;      // light direction
```

The complete pipeline works as follows:

1. The **vertex shader** is executed in parallel on each vertex, applying geometric transformations.
2. The projected triangles are **rasterized**: split into fragments (pixels).
3. The **fragment shader** is executed in parallel on each fragment, computing the final color.
4. The result is the **final image** displayed on the screen.

The attributes transmitted from the vertex shader to the fragment shader (such as `vertexColor`) are automatically **interpolated** in a barycentric manner between the triangle vertices, which corresponds exactly to the barycentric interpolation described previously. The entire pipeline is summarized in the diagram below.

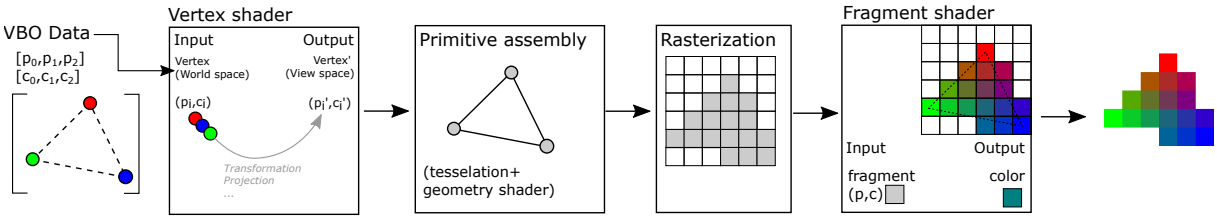


Figure 15: Rendering pipeline: vertex shader, rasterization, fragment shader.

2 Rendering Pipeline and Illumination

2.1 Pipeline overview

From the 3D scene to the image

As seen in the previous chapter, a 3D scene is described by models (surfaces), light sources, and a camera. Rasterization-based rendering produces a 2D image of this scene.

However, at the GPU and OpenGL level, there is **no explicit notion** of a camera or a light. The GPU only handles **vertices** with their attributes (position, color, normal, etc.) as input to the **vertex shader**, and colored **fragments** (pixels) as output of the **fragment shader**. The programmer's entire job consists of translating high-level concepts (camera, light, materials) into operations on vertices and fragments through shaders.

The rendering pipeline is broken down into three major stages. The first is **vertex transformation**, performed by the vertex shader: each vertex is projected from the 3D scene space to the 2D screen space by applying transformation and projection matrices. The second stage is **rasterization**: the projected 2D triangles are converted into fragments (pixels) through an automatic discretization process. Finally, the third stage is **color computation**, performed by the fragment shader: for each fragment, its final color is determined based on lighting, material, and textures.

2.2 Vertex transformation — Projection and perspective

Projection principle

OpenGL only displays triangles whose vertices lie within the cube $[-1, 1]^3$. This normalized space is called **Normalized Device Coordinates (NDC)**. The first two components $(x_{\text{ndc}}, y_{\text{ndc}})$ correspond to screen coordinates, while z_{ndc} encodes the **depth**, that is, the distance to the camera in image space. This normalized cube is shown in the following figure.

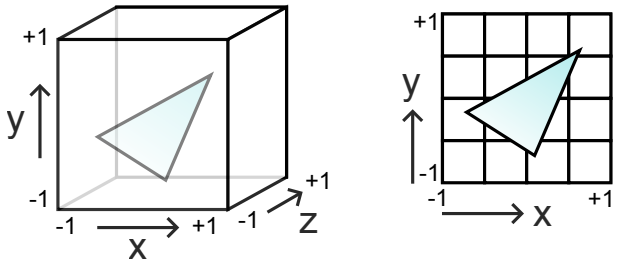


Figure 16: NDC space: the normalized cube in which OpenGL displays triangles.

The goal of projection is to **convert world coordinates** (world space) into NDC coordinates. This conversion must on one hand define a **camera model** that specifies which part of 3D space is visible, and on the other hand apply **perspective** so that distant objects appear smaller than nearby objects.

The most common perspective model uses a visibility volume shaped like a **truncated pyramid** (frustum), bounded by a near plane (z_{near}) and a far plane (z_{far}), as illustrated below.

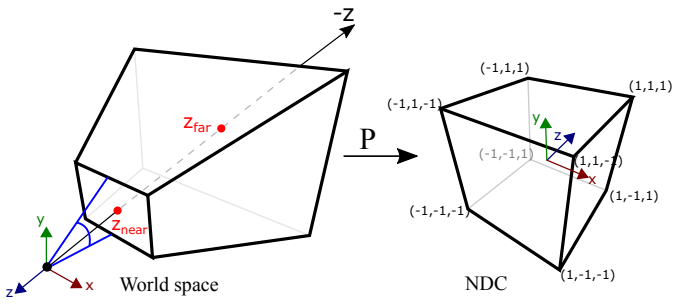


Figure 17: The frustum: the camera's visibility volume in perspective.

Projection formulation

Let a point with coordinates (x, y, z) in world space. The NDC coordinates $(x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}})$ are computed as follows:

x and y coordinates:

$$x_{\text{ndc}} = \frac{z_{\text{near}}}{-z} \cdot \frac{x}{w} = \frac{1}{\tan(\theta/2)} \cdot \frac{x}{-z}$$

$$y_{\text{ndc}} = \frac{z_{\text{near}}}{-z} \cdot \frac{y}{h} = \frac{r}{\tan(\theta/2)} \cdot \frac{y}{-z}$$

where θ is the **Field of View** (FOV) and $r = h/w$ is the aspect ratio.

The division by $-z$ produces the perspective effect: distant objects (large $|z|$) are reduced in size.

z coordinate (depth):

The NDC depth is a nonlinear function of z:

$$z_{\text{ndc}} = \frac{1}{-z}\left(-\frac{z_{\text{far}} + z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} \cdot z - \frac{2\, z_{\text{near}}\, z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}}\right)$$

with the mappings: $z = -z_{\text{near}} \to z_{\text{ndc}} = -1$ and $z = -z_{\text{far}} \to z_{\text{ndc}} = 1$.

The $1/z$ variation implies **finer precision** near the camera and coarser precision far away. This is a deliberate choice: nearby objects require better depth resolution to avoid visual artifacts.

Projection matrix

In homogeneous coordinates, the projection is expressed as a 4×4 matrix product:

$$p_{\text{ndc}} = \text{Proj} \times p$$

with:

$$\text{Proj} = \begin{pmatrix} \frac{1}{\tan(\theta/2)} & 0 & 0 & 0 \\ 0 & \frac{r}{\tan(\theta/2)} & 0 & 0 \\ 0 & 0 & -\frac{z_{\text{far}}+z_{\text{near}}}{z_{\text{far}}-z_{\text{near}}} & -\frac{2\, z_{\text{near}}\, z_{\text{far}}}{z_{\text{far}}-z_{\text{near}}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

After multiplication, the resulting vector has a component $w \neq 1$. **Homogenization** (division by w) yields the final NDC coordinates. This is exactly the mechanism of projective space seen in the previous chapter.

The result is a space **warped** into the cube $[-1, 1]^3$. The final image corresponds to the $(x_{\text{ndc}}, y_{\text{ndc}})$ view of this cube, while z_{ndc} represents the depth as seen from the camera in image space. Everything outside the cube $[-1, 1]^3$ is automatically discarded by the GPU (clipping).

Z-fighting

The $1/z$ variation of NDC depth has an important consequence: the precision depends strongly on the choice of z_{near}. The following diagram shows this non-uniform distribution of precision.

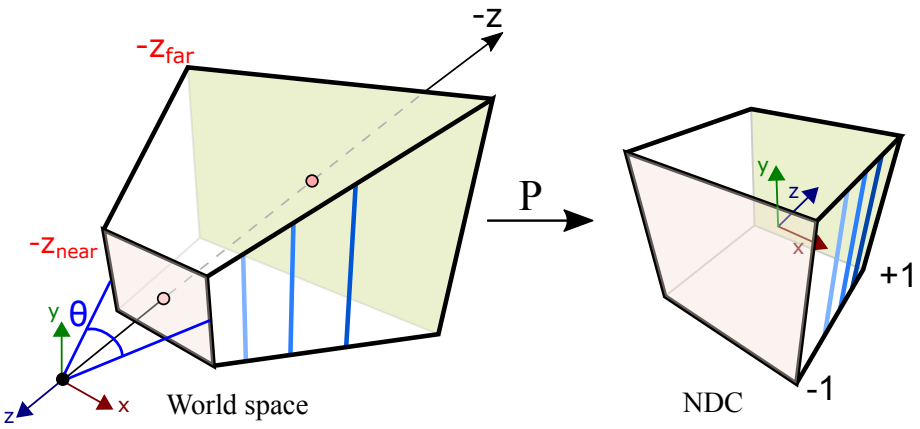


Figure 18: Distribution of depth precision as a function of the distance to the camera.

If z_{near} is **too close to 0**, all precision is concentrated at distances very close to the camera. Distant objects end up with nearly identical depth values after discretization. This causes a visual artifact called **z-fighting** (or depth-fighting): two surfaces close in depth "flicker" randomly because the GPU cannot determine which one is in front. The graph below illustrates the curve of z_{ndc} as a function of z.

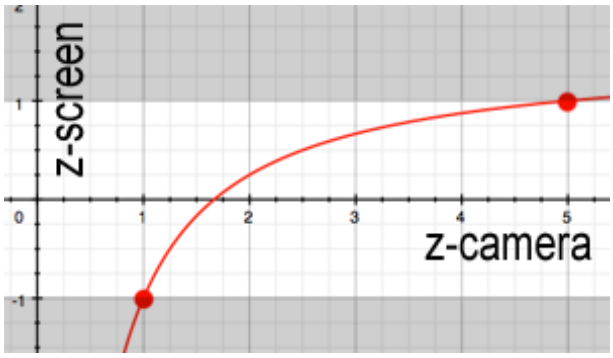


Figure 19: Graph of the depth value distribution z_{ndc} as a function of z.

Practical rule: choose z_{near} as large as possible (while keeping visible objects within the frustum) to maximize depth precision across the entire scene.

View matrix

The view matrix transforms coordinates from **world space** to **camera space** (view space). It positions and orients the camera in the scene, following the principle illustrated below.

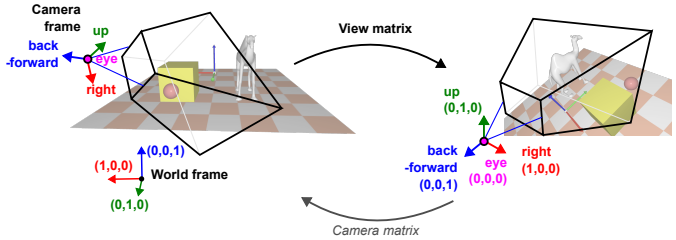


Figure 20: Transformation from world space to camera space.

The camera is described by a 4×4 matrix containing its local axes and its position:

$$\text{Cam} = \begin{pmatrix} \text{right}_x & \text{up}_x & \text{back}_x & \text{eye}_x \\ \text{right}_y & \text{up}_y & \text{back}_y & \text{eye}_y \\ \text{right}_z & \text{up}_z & \text{back}_z & \text{eye}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O & \text{eye} \\ 0 & 1 \end{pmatrix}$$

where $O = (\text{right}, \text{up}, \text{back})$ is the camera's rotation matrix (orthonormal basis) and eye is its position in the world.

The view matrix is the **inverse** of the camera matrix:

$$\text{View} = \text{Cam}^{-1} = \begin{pmatrix} O^T & -O^T \cdot \text{eye} \\ 0 & 1 \end{pmatrix}$$

This inversion is very efficient to compute because O is orthogonal ($O^{-1} = O^T$). The view matrix sends the camera position to the origin and aligns its axes with the coordinate frame axes.

Model matrix

The model matrix positions an object in the world. It transforms the object's **local** coordinates to **world space** coordinates:

$$\text{Model} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

where R is the rotation matrix (object orientation) and t is the translation vector (object position). The following figure shows this mapping from local coordinates to world space.

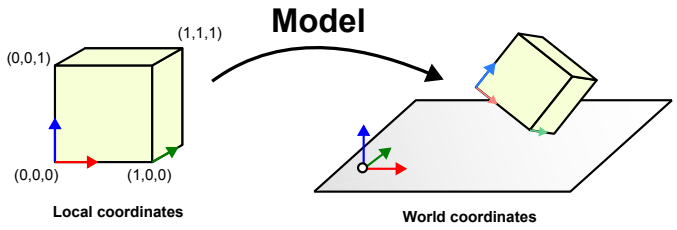


Figure 21: The model matrix places the object in the world.

The benefit of separating the object's geometry from its position is fundamental. The object's geometric coordinates (mesh) are loaded **only once** into VRAM, and to move or orient the object, one only needs to **modify the Model matrix**, which is a simple 4×4 matrix of 16 floats. This separation also enables **instancing**: the same mesh can be displayed at multiple locations in the scene by applying different Model matrices, without duplicating the geometric data in memory. For example, a forest composed of thousands of trees can store only a single tree mesh, with each instance having its own Model matrix (position, rotation, scale).

Summary: the transformation chain

The complete transformation of a vertex, from its local coordinates to NDC space, is the composition of the three matrices:

$$p_{\text{ndc}} = \text{Proj} \times \text{View} \times \text{Model} \times p$$

The vertex starts from its local coordinates $p = (x, y, z, 1)$ in **object space**, where the geometry is defined relative to the object's origin. The Model matrix places it in **world space**: $p_{\text{world}} = \text{Model} \times p$, applying the object's rotation, scale, and translation. The View matrix then transforms this position into **camera space**: $p_{\text{view}} = \text{View} \times p_{\text{world}}$, where the camera is at the origin and looks in the $-z$ direction. Finally, the Projection matrix converts the coordinates to **NDC**: $p_{\text{ndc}} = \text{Proj} \times p_{\text{view}}$, applying perspective and normalizing the coordinates into the cube $[-1, 1]^3$ after homogenization (division by w).

This chain constitutes the **first stage** of the graphics pipeline, performed by the **vertex shader**.

```
#version 330 core

layout (location = 0) in vec3 vertex_position;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    gl_Position = projection * view * model * vec4(vertex_position, 1.0);
}
```

The three matrices are sent to the GPU as **uniform variables** from the C++ code. The GPU then applies this transformation **in parallel on all vertices**. This is much more efficient than computing the new positions on the CPU: instead of transferring N modified positions at each frame, only three 4×4 matrices (i.e., 48 floats) are transferred.

```
void main_loop() {
    // Update matrices
    glUniform(shader, Model);
    glUniform(shader, View);
    glUniform(shader, Projection);

    draw(mesh_drawable);
}
```

2.3 Rasterization

Principle

Rasterization is the conversion of vector data (triangles defined by vertices) into discrete elements: **pixels** (or fragments). It is an **automatic and non-programmable** stage in the OpenGL pipeline.

The fundamental operation is as follows: given a triangle defined by 3 2D points (after projection), determine the set of pixels it covers, as can be seen in the following figure.

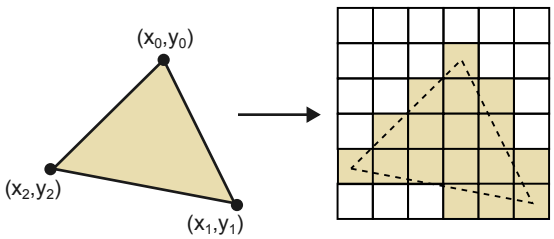


Figure 22: Rasterization of a triangle: conversion from vector shape to pixels.

Rasterization of simple primitives

Before addressing triangle rasterization, let us consider simpler primitives to understand the principle of discretization.

Point: a single pixel.

```
im(x0, y0) = c;
```

Rectangle: two nested loops.

```
for(int kx = x0; kx < x1; kx++)
    for(int ky = y0; ky < y1; ky++)
        im(kx, ky) = c;
```

Horizontal, vertical, or diagonal segment: a single loop suffices.

```
// Horizontal segment
for(int kx = x0; kx < x1; kx++)
    im(kx, y0) = c;
```

For an **arbitrary segment**, the discretization is no longer trivial: several pixelized approximations are possible, as shown in the image below. An efficient and deterministic algorithm must be chosen.

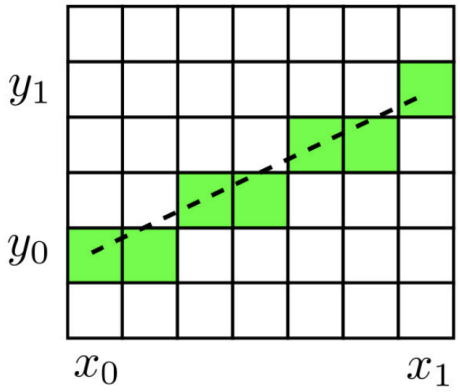


Figure 23: Discretization of a segment: several possible approximations.

Bresenham's algorithm

Bresenham's algorithm is a classic and highly efficient algorithm for rasterizing a segment. It relies solely on integer operations, making it fast and exact.

Principle: we advance along the x axis pixel by pixel. At each step, we evaluate whether the accumulated error in y exceeds 0.5:

- If yes: we increment y by 1 and correct the error.
- If no: we keep the same y.

The algorithm's progression is visualized in the following figure.

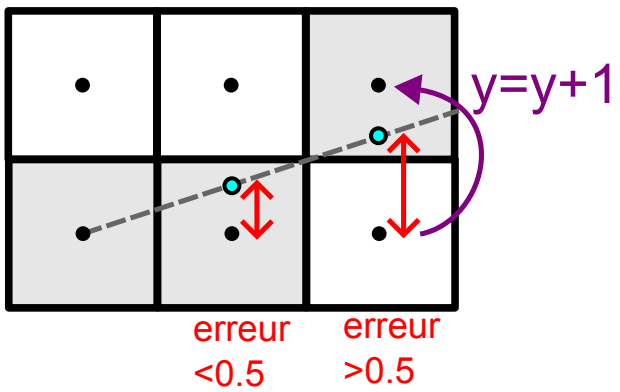


Figure 24: Bresenham's algorithm: progression along the segment with error correction.

```
int dx = x1 - x0, dy = y1 - y0;
float a = float(dy) / float(dx);
float error = 0.0f;

int y = y0;
for(int x = x0; x <= x1; x++)
{
    im(x, y) = c;
    error += a;
    if(error >= 0.5f)
    {
        y = y + 1;
        error = error - 1.0f;
    }
}
```

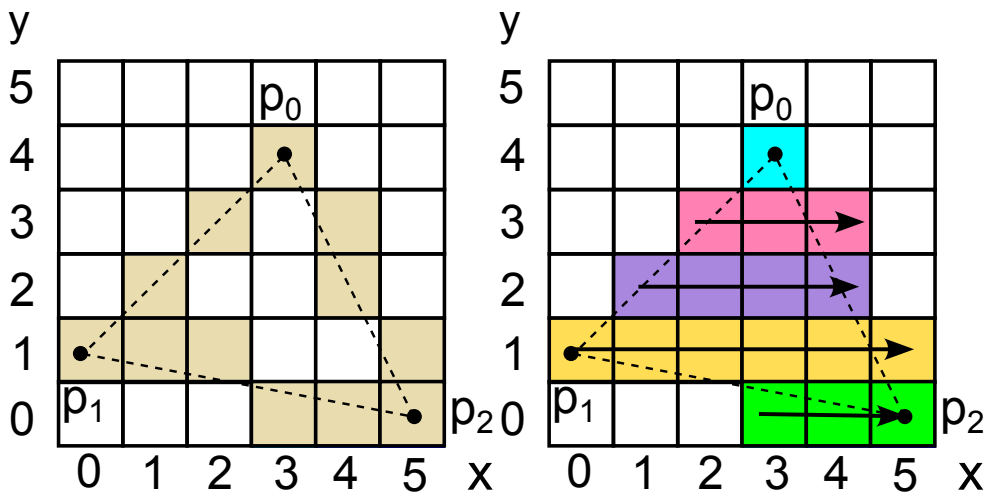
Note: by multiplying all values by $2\,dx$, we can eliminate divisions and floating-point numbers to obtain a purely integer algorithm. The algorithm above handles the case $0 \leq dy \leq dx$; the other octants are handled by symmetry.

Triangle rasterization: scanline algorithm

The rasterization of a triangle (p_0, p_1, p_2) is done using the **scanline algorithm**:

1. **Discretize the edges**: for each edge of the triangle $((p_0, p_1), (p_1, p_2), (p_2, p_0))$, apply Bresenham's algorithm to obtain the contour pixels.
2. **Store the horizontal bounds**: for each row y, store the x_{min} and x_{max} values of the contour pixels.
3. **Fill row by row**: for each row y, color all pixels from x_{min} to x_{max}.

The two steps are illustrated below: edge discretization followed by filling.



Triangle rasterization by scanline: edge discretization (left) and horizontal filling (right).

Scanline table example for a triangle:

y	x_{min}	x_{max}
0	3	5
1	0	5
2	1	4
3	2	4
4	3	3

During filling, the **barycentric coordinates** of each fragment are computed to interpolate the vertex attributes (color, normal, texture coordinates, depth) linearly within the triangle.

2.4 Illumination and shading

Phong illumination model

The **Phong model** is the most classic illumination model in real-time rendering. It is based on the observation that the appearance of an illuminated surface can be decomposed into three distinct physical phenomena. The first is the **ambient** component, which represents a uniform base color, independent of geometry and light position: it roughly models the indirect light that pervades the entire scene. The second is the **diffuse** component, which depends on the local orientation of the surface relative to the light: a surface facing the light is bright, a surface facing away is dark. The third is the **specular** component, which models the bright reflection of the light source visible on smooth surfaces, and which depends on the observer's viewpoint. These three contributions are shown in the following figure.

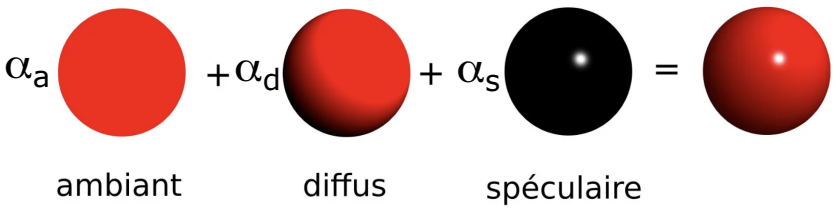


Figure 25: The three components of the Phong model: ambient, diffuse, and specular.

The final color of a point is the sum of these three components:

$$C = C_a + C_d + C_s$$

The computation of each component involves two color properties: the **light source color** $C_\ell = (r_\ell, g_\ell, b_\ell) \in [0,1]^3$ and the **surface color** (or albedo) $C_o = (r_o, g_o, b_o) \in [0,1]^3$. Their component-wise multiplication ($(r_\ell \cdot r_o,\ g_\ell \cdot g_o,\ b_\ell \cdot b_o)$) models the filtering of light by the material: a red surface lit by a white light appears red, while the same surface lit by a green light appears black (because the red component of the light is zero).

Ambient component

The ambient component models uniform illumination of the scene (approximated indirect light). It depends neither on the position nor the orientation of the surface:

$$C_a = \alpha_a\, C_\ell\, C_o$$

with $\alpha_a \in [0,1]$ the **ambient coefficient**. This term prevents surfaces not directly lit from being completely black.

Diffuse component

The diffuse component models light reflected uniformly in all directions by a matte surface (Lambertian reflection). It depends on the angle between the surface **normal** n and the **light direction** u_ℓ:

$$C_d = \alpha_d\, (n \cdot u_\ell)_+\, C_\ell\, C_o$$

The coefficient $\alpha_d \in [0, 1]$ controls the intensity of the material's diffuse reflection. The central term is the **diffuse factor** $(n \cdot u_\ell)_+ = \max(n \cdot u_\ell, 0)$, which is the dot product between the unit normal vector n to the surface and the unit direction u_ℓ from the point to the light source. This dot product measures the cosine of the angle between the normal and the light direction. When the surface directly faces the light (n parallel to u_ℓ), the factor equals 1 and the lighting is maximal. When the surface is at a grazing angle (n perpendicular to u_ℓ), the factor equals 0. When the surface faces away from the light, the dot product is negative, and the clamping $\max(\cdot, 0)$ forces the contribution to zero, avoiding physically absurd lighting.

This geometric mechanism is illustrated below.

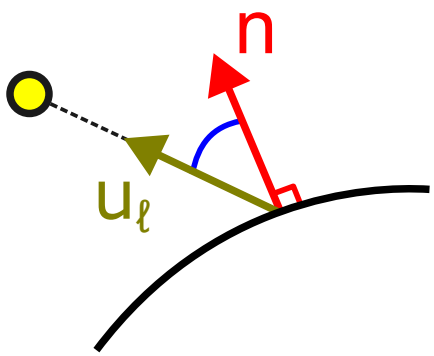


Figure 26: Diffuse component: brightness depends on the angle between the normal and the light direction.

Specular component

The specular component models the **reflection** of the light source on the surface. Unlike the diffuse component, it depends on the camera's **viewpoint**:

$$C_s = \alpha_s \, (u_r \cdot u_v)_+^{s_{\text{exp}}} \, C_\ell$$

The coefficient $\alpha_s \in [0, 1]$ controls the intensity of the reflection. The shininess exponent s_{exp} (typically between 64 and 256) determines the **size** of the reflection: the higher it is, the more the reflection is concentrated into a narrow, sharp point (highly polished surface), while a low exponent produces a wide, diffuse reflection (slightly rough surface).

The vector u_r is the **reflection direction** of the light with respect to the normal, computed by the formula $u_r = 2\,(u_\ell \cdot n)\,n - u_\ell$ (in GLSL: `reflect(-u_l, n)`). The vector u_v is the unit direction from the surface point to the camera. The dot product $(u_r \cdot u_v)$ measures the alignment between the reflection direction and the view direction: the reflection is maximal when the observer is exactly in the mirror reflection direction of the light.

The specular component does **not** depend on the surface color C_o but only on C_ℓ, which corresponds to the physical behavior of reflections on dielectric materials (plastic, ceramic, etc.): the reflection of a white light on a red surface remains white. The geometry of specular reflection and the influence of the shininess exponent are detailed in the following figures.

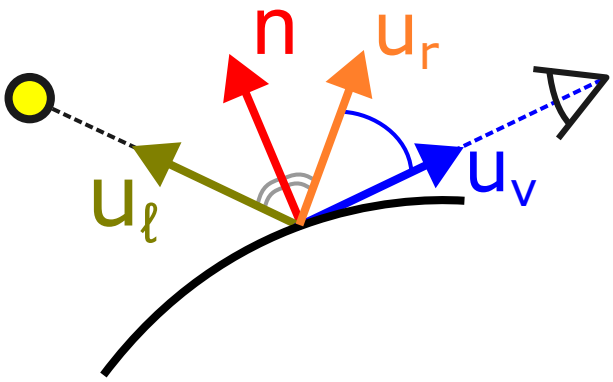


Figure 27: Specular component: the reflection depends on the reflection direction and the viewpoint.

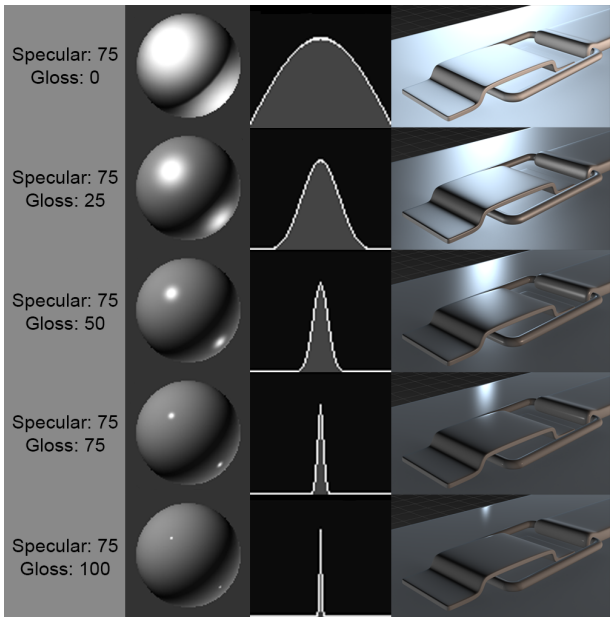


Figure 28: Effect of the shininess exponent: the higher s_{exp}, the more concentrated the reflection.

Illumination interpolation: Flat, Gouraud, Phong

The Phong model defines the illumination formula at a point on the surface, but the question remains of **where** and **how often** this formula is evaluated. Three classic strategies exist, with a trade-off between computational cost and visual quality.

Flat shading is the simplest approach: the color is computed **once per triangle**, using the geometric normal of the face. All pixels of the triangle receive exactly the same color. The result has a characteristic faceted appearance where each triangle is clearly visible, which may be acceptable for non-photorealistic rendering but is unsuitable for smooth surfaces.

Gouraud shading improves quality by computing the illumination **at the vertices** of the triangle (in the vertex shader), then **linearly interpolating** the resulting color inside the triangle during rasterization. Color transitions between adjacent triangles become smooth, giving the appearance of a smooth surface. However, this approach has an important flaw: if a specular highlight falls in the middle of a large triangle, far from any vertex, it will be completely missed because no vertex ”saw” the highlight.

Phong shading solves this problem by interpolating not the color, but the **normal** between the triangle's vertices. The illumination is then computed **for each fragment** (in the fragment shader) using this interpolated normal. The specular highlight is thus correctly computed at every point on the surface, even at the center of a large triangle. The visual difference between these three approaches is compared in the following figure.

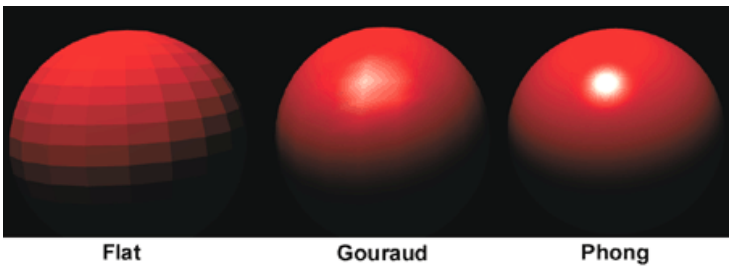


Figure 29: Comparison of the three shading models: Flat, Gouraud, and Phong.

In practice, **Phong shading** is the standard. The overhead compared to Gouraud shading is negligible on modern GPUs (the fragment shader performs a few additional operations per pixel, but the GPU has thousands of cores to execute them in parallel), and the visual quality is significantly better.

Multiple lights and effects

Multiple lights

The Phong model naturally extends to multiple light sources by summing the contributions of each light:

$$C = \sum_i \left(\alpha_a \, C_{\ell_i} \, C_o + \alpha_d \, (n \cdot u_{\ell_i})_+ \, C_{\ell_i} \, C_o + \alpha_s \, (u_{r_i} \cdot u_v)_+^{s_{\text{exp}}} \, C_{\ell_i} \right)$$

Distance attenuation

In physical reality, the light intensity of a point source decreases proportionally to the inverse square of the distance ($1/r^2$). In real-time rendering, a simplified attenuation model is often used that decreases linearly up to a maximum distance, beyond which the light has no effect:

$$C_\ell(p) = \left(1 - \min \left(\frac{\|p - p_\ell\|}{d_{\text{att}}}, \, 1 \right) \right) \, C_\ell^0$$

The vector p is the position of the point on the surface, p_ℓ the position of the light, d_{att} the characteristic attenuation distance (beyond which the contribution is zero), and C_ℓ^0 the color of the light at the source. This linear model is less realistic than the $1/r^2$ law, but it has the advantage of guaranteeing that the light completely vanishes beyond d_{att}, which allows optimizing rendering by ignoring lights that are too far away.

Fog effect

A fog effect simulates the absorption and scattering of light by the atmosphere. The principle is to progressively blend the computed color with a uniform fog color as a function of the distance to the camera:

$$C_{\text{final}} = (1 - \gamma(p)) \, C + \gamma(p) \, C_{\text{fog}}$$

The blending factor $\gamma(p) = \min \left(\frac{\|p - p_{\text{eye}}\|}{d_{\text{fog}}}, \, 1 \right)$ varies from 0 (nearby object, color unchanged) to 1 (distant object, completely engulfed in fog). The parameter d_{fog} controls the distance at which the fog becomes fully opaque, and C_{fog} is the fog color (often a light gray or the sky color). In practice, this effect gives an impression of atmospheric depth and naturally hides the far clipping plane (z_{far}), avoiding the abrupt disappearance of objects at the horizon.

Example effect: Toon Shading

Toon shading (or cel shading) is a non-photorealistic effect that gives a cartoon-like appearance. The principle is to **discretize** the color values into steps:

$$C_{\text{toon}} = \frac{\text{floor}(C \times N)}{N}$$

where N is the desired number of steps. This downsampling creates discrete color bands characteristic of the cartoon style, an example of which is shown below.

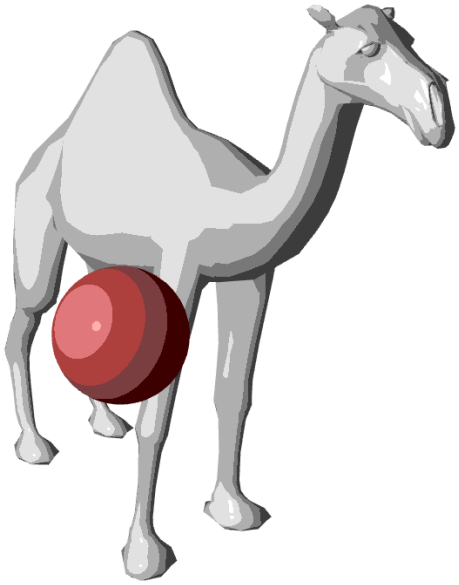


Figure 30: Toon shading: color discretization for a stylized rendering.

2.5 Depth buffer

Occlusion problem

When multiple triangles overlap on screen, we need to determine which one is visible for each pixel. Without an occlusion mechanism, the drawing order of triangles determines the result, which is incorrect, as highlighted by the following image.

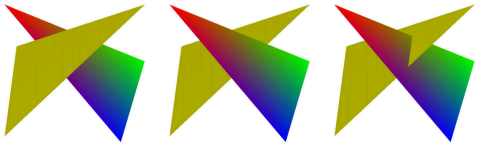


Figure 31: The drawing order of triangles affects the result without a depth mechanism.

Z-buffer algorithm

The **Z-buffer** (or depth buffer) is an auxiliary image of the same resolution as the final image, which stores the depth value z_{ndc} of the fragment closest to the camera for each pixel.

The algorithm is simple: for each fragment to be drawn, its depth is compared with the value stored in the Z-buffer:

```
def draw(position, color, z, image, zbuffer):
    if z < zbuffer(position):
        image(position) = color
        zbuffer(position) = z
    # otherwise: draw nothing (occluded fragment)
```

The Z-buffer contents can be visualized as a grayscale image.

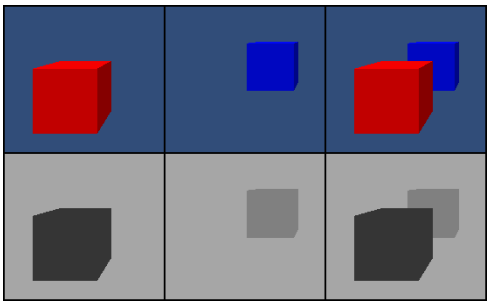


Figure 32: Depth buffer visualization: bright areas are close to the camera, dark areas are far away.

The Z-buffer is initialized to the maximum depth value (1.0, corresponding to the far plane) at the beginning of each frame. Each fragment updates the Z-buffer if its depth is smaller (closer to the camera) than the stored value. This guarantees that only the closest fragment is displayed, **regardless of the drawing order** of the triangles.

This step is performed automatically by the GPU (non-programmable), provided that the depth test is enabled (`glEnable(GL_DEPTH_TEST)` in OpenGL).

2.6 Shaders: pipeline summary

Complete pipeline

The complete rendering pipeline chains several stages in sequence. The **vertex shader** first transforms each vertex by applying the Projection × View × Model matrix chain, and passes the transformed attributes (world-space position, normal, color) to the subsequent stages. The transformed vertices are then grouped into triangles during **primitive assembly**, using the connectivity table sent from the CPU.

Rasterization takes each 2D-projected triangle and determines the set of pixels it covers. For each covered pixel (called a fragment), the attributes of the triangle's three vertices are interpolated barycentrically. The **fragment shader** receives these interpolated attributes and computes the fragment's final color by applying the Phong illumination model. The **depth test** (Z-buffer) then compares the fragment's depth with the stored value for that pixel: only the fragment closest to the camera is kept, the others are discarded. The result of all these stages is the **final image**, stored in the framebuffer and displayed on screen.

The complete sequence of these stages is summarized in the diagram below.

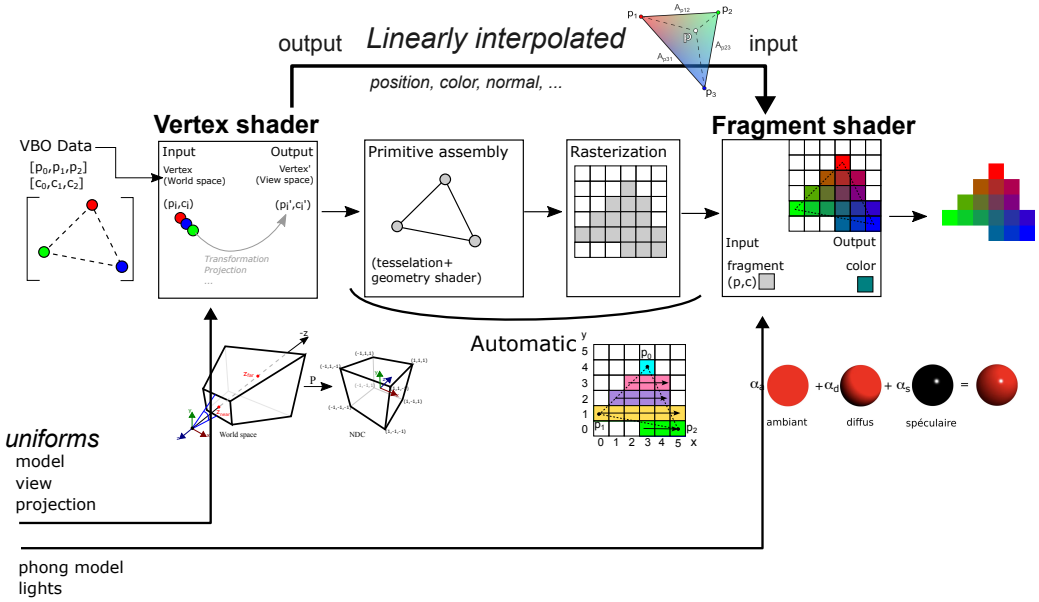


Figure 33: Complete rendering pipeline: from vertex to final image.

Complete code example

Vertex shader: transforms positions and normals, passes attributes to the fragment shader.

```
#version 330 core

layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_normal;
layout (location = 2) in vec3 vertex_color;

out struct fragment_data {
    vec3 position;
    vec3 normal;
    vec3 color;
} fragment;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    vec4 position = model * vec4(vertex_position, 1.0);

    mat4 modelNormal = transpose(inverse(model));
    vec4 normal = modelNormal * vec4(vertex_normal, 0.0);

    fragment.position = position.xyz;
    fragment.normal   = normal.xyz;
    fragment.color    = vertex_color;

    gl_Position = projection * view * position;
}
```

Note: the normal is transformed by the **transpose of the inverse** of the Model matrix (`transpose(inverse(model))`), not by Model directly. Indeed, if Model contains a non-uniform scaling, direct multiplication would distort the normals and the shading would be incorrect.

Fragment shader: computes Phong illumination for each fragment.

```
#version 330 core

in struct fragment_data {
    vec3 position;
    vec3 normal;
    vec3 color;
} fragment;

uniform vec3 light_position;
uniform vec3 light_color;

layout(location = 0) out vec4 FragColor;

void main() {
    vec3 N = normalize(fragment.normal);
    vec3 L = normalize(light_position - fragment.position);

    float diffuse_magnitude = max(dot(N, L), 0.0);

    vec3 c = 0.1 * fragment.color * light_color;                    // ambient
    c = c + 0.7 * diffuse_magnitude * fragment.color * light_color;  // diffuse
    // + specular component (omitted for simplicity)

    FragColor = vec4(c, 1.0);
}
```

The variables `fragment.position`, `fragment.normal`, and `fragment.color` are automatically **interpolated barycentrically** by the GPU between the values of the current triangle's three vertices.

3 Meshes and Textures

3.1 Mesh Geometry

Structure of a Mesh

A **mesh** is a set of polygons sharing vertices. It is described by:

- N_f **faces** (polygons).
- N_v **vertices**.
- N_e **edges**.

Depending on the type of polygons used, we distinguish:

- **Triangulation**: all faces are triangles. This is the standard format for GPU rendering.
- **Quad mesh**: all faces are quadrilaterals (ideally planar). Used in modeling and subdivision.
- **Poly mesh**: faces are arbitrary polygons (variable number of sides).

These three categories are illustrated below.

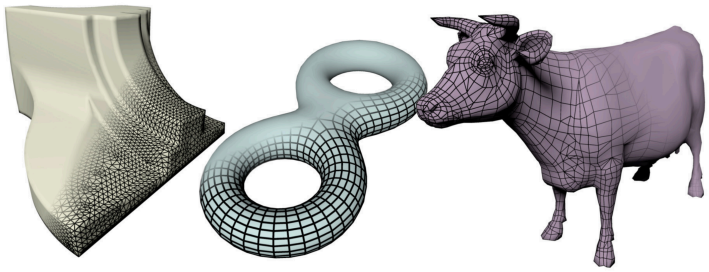


Figure 34: Mesh types: triangulation, quad mesh, and poly mesh.

A mesh represents a **manifold** if each edge is shared by **at most 2 faces**, as illustrated in the following figure. This property is fundamental: it guarantees that the surface is locally equivalent to a plane (or a half-plane at the boundary), which is required by most mesh processing algorithms.

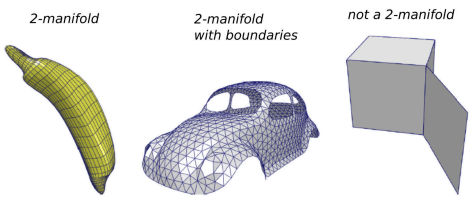


Figure 35: Manifold property: each edge is shared by at most 2 faces.

Data Structure

The standard encoding of a triangular mesh in C++ uses two arrays:

```cpp
struct vec3 { float x, y, z; };
struct int3 { int i, j, k; };

std::vector<vec3> position;      // vertex positions
std::vector<int3> connectivity;  // face indices (triplets)
```

Using `std::vector` guarantees **memory contiguity**, which is essential for efficient transfer to the GPU.

Example: tetrahedron

```
std::vector<vec3> position = { {0,0,0}, {1,0,0}, {0,1,0}, {0,0,1} };
std::vector<int3> connectivity = { {0,1,2}, {0,1,3}, {0,2,3}, {1,2,3} };
```

Example: planar mesh

```
std::vector<vec3> position = { p0, p1, p2, p3, p4, p5, p6, p7, p8 };
std::vector<int3> connectivity = { {0,1,3}, {1,2,3}, {0,3,4}, {3,5,4},
                                   {2,5,3}, {2,6,5}, {5,6,7}, {5,7,8}, {4,5,8} };
```

The following diagram shows the correspondence between indices and the mesh geometry.

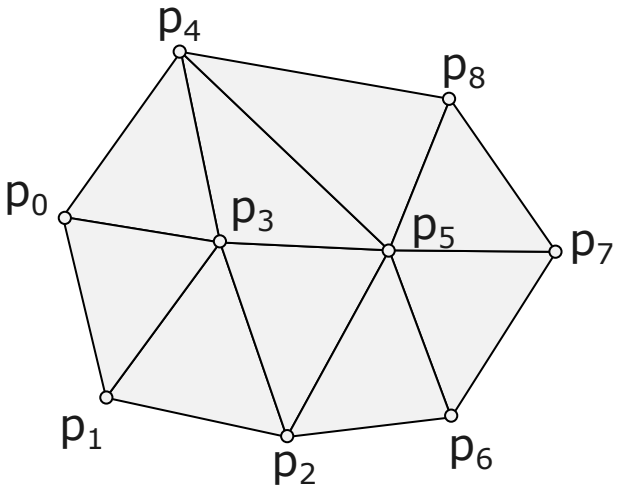


Figure 36: Structure of a triangular mesh: vertices and connectivity.

Note on orientation: the triplets $(0, 1, 3)$, $(1, 3, 0)$, and $(3, 0, 1)$ are equivalent (cyclic permutation). However, $(0, 1, 3)$ has the **opposite** orientation to $(3, 1, 0)$, $(1, 0, 3)$, or $(0, 3, 1)$. The orientation determines the direction of the normal and thus the front side of the face (see previous chapter).

CPU/GPU Data Flow

The typical workflow for displaying a mesh with OpenGL follows these steps:

1. **Define** a `mesh_drawable` variable (shared across methods).
2. **Initialize** a `mesh` structure containing the data arrays in RAM (CPU).
3. **Transfer** the data from the `mesh` to the `mesh_drawable` in VRAM (GPU).
4. **Each frame**:
    - Update uniform parameters (matrices, lights, etc.).
    - Call `draw()` which activates the shader, sends the uniforms, and triggers rendering.

This flow is summarized in the diagram below.

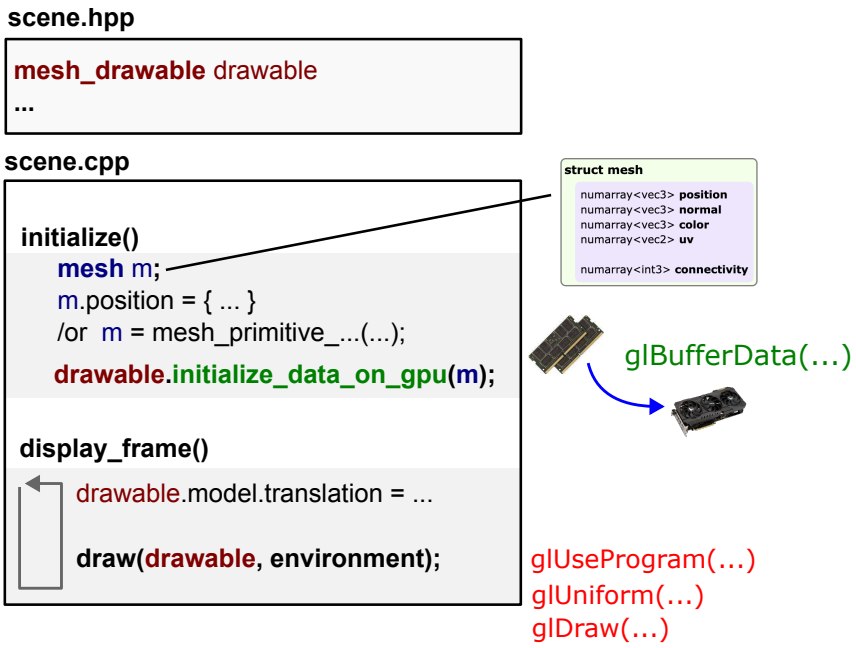


Figure 37: Data flow from CPU to GPU for mesh rendering.

Vertex Attributes

In practice, each vertex carries **additional attributes** beyond its position: normals, texture coordinates, colors, etc.

```
std::vector<vec3> position     = { p_0, p_1, p_2, p_3 };
std::vector<vec3> normal       = { n_0, n_1, n_2, n_3 };
std::vector<vec3> color        = { c_0, c_1, c_2, c_3 };
std::vector<vec2> uv           = { uv_0, uv_1, uv_2, uv_3 };
std::vector<int3> connectivity = { {0,1,2}, {0,1,3}, {0,2,3}, {1,2,3} };
```

with $p_i = (x_i, y_i, z_i)$, $n_i = (n_{x_i}, n_{y_i}, n_{z_i})$ with $\|n_i\| = 1$, and $c_i = (r_i, g_i, b_i)$. The following figure shows these different attributes on a mesh.

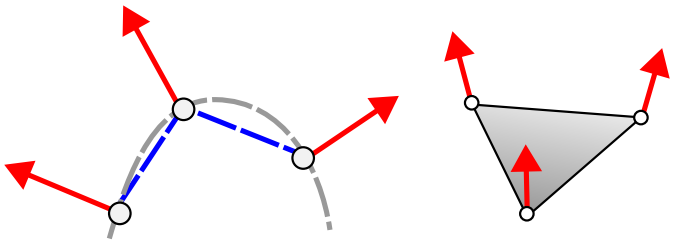


Figure 38: Vertex attributes: position, normal, color, and texture coordinates.

Key points:

- **Normals are defined per vertex** (not per face), which allows a smooth appearance through illumination interpolation in the fragment shader.
- A **single connectivity table** is shared by all attributes for GPU rendering. This means that the indices in the face table apply simultaneously to positions, normals, colors, and texture coordinates.

Vertex Duplication

Sometimes it is necessary to **duplicate** a vertex when it occupies the same position but with **different attributes** depending on the face considered. The most common case is that of **sharp edges**.

Consider a vertex located at position p_A where two groups of faces meet at a right angle. If we want a rendering with a sharp (non-smoothed) edge, this vertex must carry two different normals, one for each group of faces. Since the connectivity table is unique, we must create **two distinct entries** in the vertex array, sharing the same position but with different normals.

```
// Before duplication: vertex v4 at position pA with a single normal
std::vector<vec3> position = { p0, p1, p2, p3, pA, pB };
std::vector<vec3> normal   = { n0, n0, n1, n1, n2, n2 };

// After duplication: v4 and v6 share pA but with different normals
std::vector<vec3> position = { p0, p1, p2, p3, pA, pB, pA, pB };
std::vector<vec3> normal   = { n0, n0, n1, n1, n1, n1, n0, n0 };
```

The principle is shown in the following diagram.

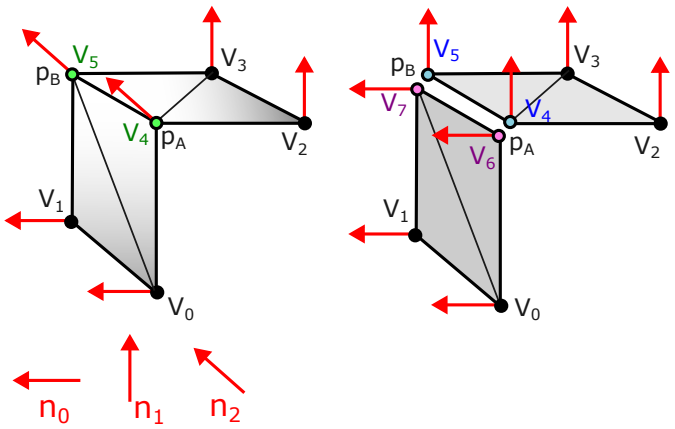


Figure 39: Vertex duplication for sharp edges: same position, different normals.

For a **cube**, each geometric vertex is shared by 3 perpendicular faces. Since each face requires a different normal, each vertex is duplicated **3 times**, resulting in $8 \times 3 = 24$ vertices instead of 8.

Parametric Surfaces and Grids

A common case of mesh construction is the **discretization of parametric surfaces**.

Let a surface $S(u, v) = (S_x(u, v),\ S_y(u, v),\ S_z(u, v))$ with $(u, v) \in [0, 1]^2$, uniformly sampled on an $N_u \times N_v$ grid, whose structure is shown below.

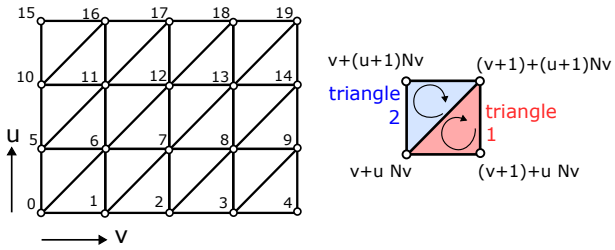


Figure 40: Grid structure for a parametric surface.

The mesh construction proceeds in two steps:

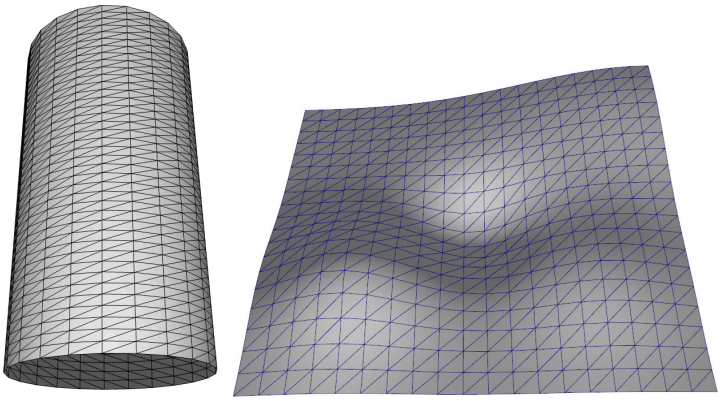
Vertex positions:

```
for(int ku = 0; ku < Nu; ku++) {
    for(int kv = 0; kv < Nv; kv++) {
        float u = ku / (Nu - 1.0f);
        float v = kv / (Nv - 1.0f);
        S.position[kv + Nv * ku] = { Sx(u,v), Sy(u,v), Sz(u,v) };
    }
}
```

Connectivity (two triangles per grid cell):

```
for(int ku = 0; ku < Nu - 1; ku++) {
    for(int kv = 0; kv < Nv - 1; kv++) {
        unsigned int idx = kv + Nv * ku;
        uint3 triangle_1 = { idx, idx + 1 + Nv, idx + 1 };
        uint3 triangle_2 = { idx, idx + Nv, idx + 1 + Nv };
        S.connectivity.push_back(triangle_1);
        S.connectivity.push_back(triangle_2);
    }
}
```

This scheme applies to many surfaces, as shown in the following examples.



Examples of parametric surfaces: cylinder (left) and terrain (right).

Normal Computation

Normals are not always provided (files without normals, procedural meshes, real-time deformations). It is then necessary to **compute them from the geometry**.

The standard method consists of computing the **unweighted average of the normals of the neighboring triangles** for each vertex:

1. For each vertex i, identify the neighboring triangles $t \in \mathcal{N}_i$ (called the **1-neighborhood** or **1-ring** of the vertex).
2. Compute the normal of each neighboring triangle: $n_t = (p_b - p_a) \times (p_c - p_a)$.
3. Sum and normalize:

$$n_i = \frac{\sum_{t \in \mathcal{N}_i} n_t}{\left\| \sum_{t \in \mathcal{N}_i} n_t \right\|}$$

This process is illustrated below.

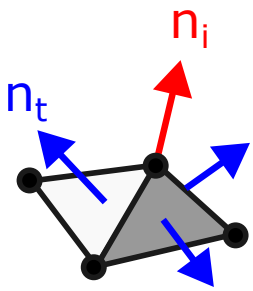


Figure 41: Computing a vertex normal by averaging the normals of neighboring triangles.

Implementation:

```
std::vector<vec3> normal(position.size(), {0,0,0});

// Accumulation per triangle
for(int t = 0; t < connectivity.size(); t++) {
    int a = connectivity[t][0];
    int b = connectivity[t][1];
    int c = connectivity[t][2];

    vec3 n = cross(position[b] - position[a], position[c] - position[a]);
    n = normalize(n);

    normal[a] += n;
    normal[b] += n;
    normal[c] += n;
}

// Final normalization
for(int k = 0; k < normal.size(); k++) {
    normal[k] = normalize(normal[k]);
}
```

If vertices are duplicated (sharp edges), the computed normals will naturally differ for each copy, since they do not share the same neighboring triangles, as shown in the following figure.

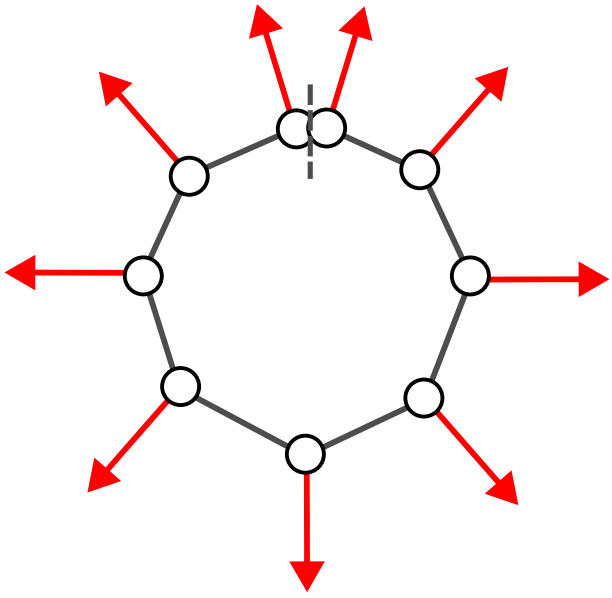


Figure 42: Effect of vertex duplication on computed normals.

1-Neighborhood (1-Ring)

The **1-ring** of a vertex is the set of triangles that share that vertex. This neighborhood structure is useful for many algorithms (normal computation, smoothing, subdivision, etc.).

```
std::vector<std::vector<int>> one_ring;
one_ring.resize(position.size());
for(int k_tri = 0; k_tri < connectivity.size(); k_tri++) {
    for(int k = 0; k < 3; k++) {
        one_ring[connectivity[k_tri][k]].push_back(k_tri);
    }
}
```

For example:

```
one_ring[235] = {842, 120, 108, 110, 20, 114};
one_ring[236] = {108, 112, 851, 850, 120, 722};
```

The figure below visualizes this neighborhood concept.

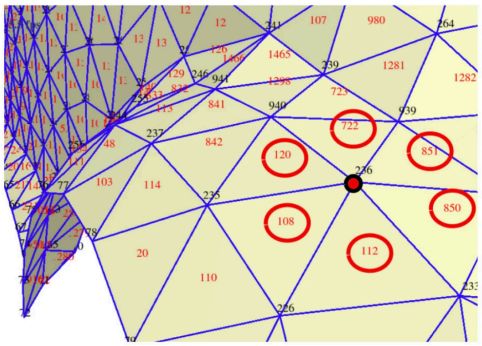


Figure 43: Visualization of the 1-ring: the set of neighboring triangles of a vertex.

Half-Edge Data Structure

The **half-edge** data structure is a richer data structure than a simple connectivity table. It encodes edges in an oriented manner: each edge is represented by two half-edges with opposite directions, as shown in the following diagram. Faces appear as loops along the half-edges.

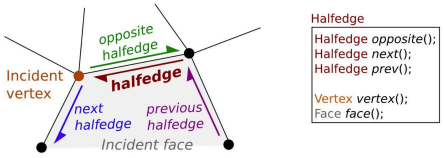


Figure 44: Half-edge data structure: each oriented edge points to its opposite.

Advantages:

- Adding and removing elements in $O(1)$ (split, collapse, flip operations).
- Efficient computation of corner angles, shown in the following figure (cotangent weighting for the discrete Laplacian).

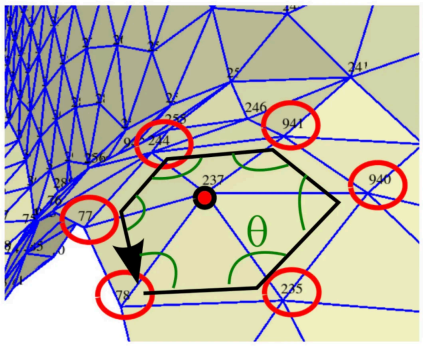


Figure 45: Corner angles of triangles, useful for cotangent weights.

Limitations:

- Restricted to **orientable 2-manifolds** (each edge shared by at most 2 faces, no non-manifold configuration).
- **Non-contiguous in memory** (pointer-based structure), which can impact performance.

Example with CGAL:

```
typedef CGAL::Cartesian<double> Kernel;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;

int main() {
    Polyhedron mesh;
    std::ifstream stream("mesh.off");
    stream >> mesh;

    int face_number = 0;
    for(auto it_face = mesh.facets_begin();
        it_face != mesh.facets_end(); ++it_face)
    {
        auto halfedge = it_face->halfedge();
        auto const halfedge_end = halfedge;
        do {
            const auto p = halfedge->vertex()->point();
            std::cout << p << std::endl;
            halfedge = halfedge->next();
        } while(halfedge != halfedge_end);
        face_number++;
    }
}
```

3.2 Textures

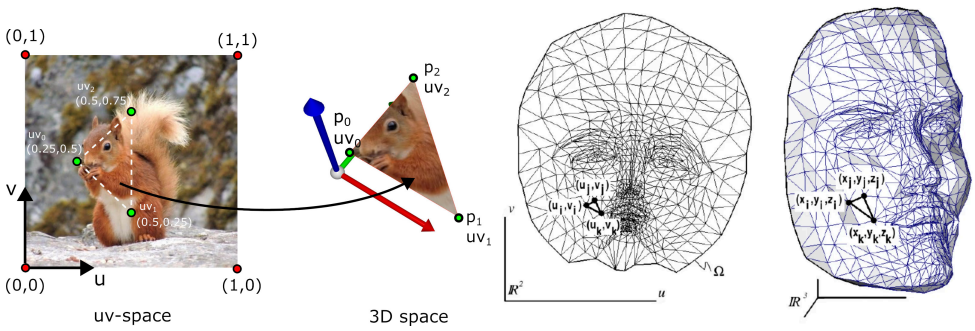
UV-Mapping Principle

The purpose of textures is to provide a **finer level of color detail than the geometry**. A triangle can have a uniform color or a color interpolated between its vertices, but this remains limited. Textures allow mapping a detailed 2D image onto the 3D surface.

The classic approach is **UV-mapping**: a 2D image (the **texture**) is associated with the 3D surface via **texture coordinates** (u, v) (also denoted (s, t)).

- Each vertex v_i of the mesh is assigned a 2D coordinate $uv_i = (u_i, v_i) \in [0, 1]^2$ in texture space.
- At render time, texture coordinates are barycentrically interpolated between the triangle's vertices, and the texture color is read at the interpolated position.

The mechanism is illustrated below.



UV-mapping principle: association between 2D texture coordinates and 3D surface.

The convention is that $(0, 0)$ corresponds to the bottom-left corner of the image and $(1, 1)$ to the top-right corner.

UV Unwrapping

For complex models, UV unwrapping is done in **patches** (or islands). The process includes the following steps:

1. **Cutting into patches**: the surface is cut along **seams** chosen by the artist or an automatic algorithm. Seams are placed in less visible areas (folds, back of the object).
2. **Unwrapping**: each patch is flattened into the 2D plane while minimizing distortions. Classic algorithms include ABF (Angle-Based Flattening), LSCM (Least Squares Conformal Maps), and Slim (Scalable Locally Injective Mappings).
3. **Packing**: the unwrapped patches are arranged in the $[0, 1]^2$ space, maximizing space utilization (to avoid wasting texture resolution).
4. **Painting**: the artist paints the texture directly in the unwrapped 2D space, or uses 3D painting tools (such as Substance Painter) that project directly onto the surface.

The seams between patches correspond to locations where the texture may exhibit **discontinuities** (the vertex at the seam is duplicated with different UV coordinates on each side). An example of unwrapping is shown below.

Figure 46: UV unwrapping in practice: the surface is cut into patches arranged in texture space.

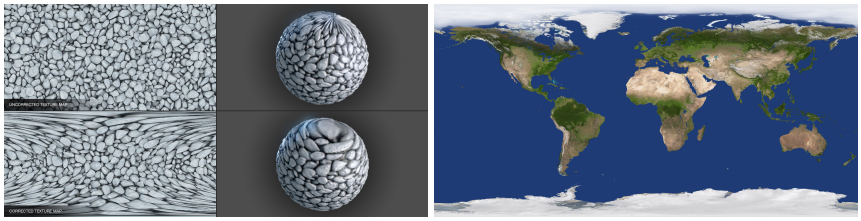
Distortions and Limitations

UV-mapping consists of ”unfolding” the 3D surface onto a 2D plane. This unfolding necessarily introduces **distortions** in length and angle for any surface whose **Gaussian curvature** is non-zero.

In particular, it is **impossible** to map a planar image onto a sphere without distortion. This is a fundamental result of differential geometry: the Gaussian curvature is an **intrinsic invariant** of the surface (Gauss's theorema egregium).

A sphere has a constant positive Gaussian curvature ($K = 1/R^2$), while a plane has zero curvature ($K = 0$). No stretch-free deformation can transform one into the other.

Only **developable** surfaces (zero Gaussian curvature, such as cylinders and cones) can be unfolded without distortion. In practice, one seeks to **minimize** distortions during UV unwrapping, accepting a trade-off between angle distortion (conformal) and area distortion (authalic). The following figure shows a typical case of distortion on a sphere.



Distortion when mapping a texture onto a sphere: stretching at the poles.

Types of Textures

Beyond color (diffuse map), textures are used to store many types of information about the surface:

- **Diffuse map** (albedo): base color of the surface.
- **Normal map**: perturbed normals to simulate relief (see next section).
- **Specular map**: variable specular coefficient across the surface (shiny vs matte areas).
- **Roughness map**: surface roughness (physically based rendering, PBR).
- **Ambient occlusion map**: pre-computed ambient occlusion (shading in crevices).
- **Height map** (displacement map): height for parallax mapping or actual geometry displacement.
- **Emissive map**: areas emitting light (screens, neon lights, etc.).

Textures in OpenGL

Setup

Texture coordinates are stored as a vertex attribute, just like positions and normals:

```
std::vector<vec2> uv = { uv_0, uv_1, uv_2, ... };
```

On the C++/OpenGL side, the texture is loaded and sent to the GPU:

```
// Image loading
int width, height, channels;
unsigned char* data = stbi_load("texture.png", &width, &height, &channels, 4);

// OpenGL texture creation
GLuint texture_id;
glGenTextures(1, &texture_id);
glBindTexture(GL_TEXTURE_2D, texture_id);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, data);

// Activation in the fragment shader (texture unit 0)
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture_id);
glUniform1i(glGetUniformLocation(shader, "texture_image"), 0);
```

Interpolation and Sampling

During rendering, the fragment shader receives the texture coordinates (u, v) barycentrically interpolated between the three vertices of the current triangle. It then reads the color from the texture image at this position using the `texture()` function:

```
uniform sampler2D texture_image;  // the texture, sent from the CPU

in vec2 fragment_uv;              // interpolated coordinates
out vec4 FragColor;

void main() {
    vec4 color = texture(texture_image, fragment_uv);
    FragColor = color;
}
```

The `sampler2D` variable represents a reference to a texture stored in VRAM. The `texture(sampler, uv)` function performs **sampling**: it converts the normalized coordinates $(u, v) \in [0, 1]^2$ into pixel coordinates in the image, then returns the corresponding color.

Texture Filtering

The interpolated (u, v) coordinates rarely fall exactly on the center of a texture pixel (called a **texel**). The GPU must therefore **interpolate** between neighboring texels. This process is called **texture filtering**. It occurs in two cases:

- **Magnification**: the triangle is larger than the texture on screen, multiple pixels cover a single texel. Without filtering, the texture appears pixelated.
- **Minification**: the triangle is smaller than the texture on screen, a single pixel covers many texels. Without filtering, the texture flickers (aliasing).

The most common filtering modes are:

- **Nearest** (`GL_NEAREST`): selects the nearest texel. Pixelated but fast rendering. Useful for a pixel-art style.
- **Bilinear** (`GL_LINEAR`): linearly interpolates between the 4 nearest texels. Smooth rendering, standard for magnification.
- **Trilinear** (`GL_LINEAR_MIPMAP_LINEAR`): bilinear interpolation + interpolation between two **mipmap** levels (see below). Standard for minification.

OpenGL configuration:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Mipmaps

Mipmaps are pre-computed versions of the texture at decreasing resolutions (each level halves the resolution). During minification, the GPU selects the mipmap level whose resolution best matches the triangle's size on screen, then interpolates if necessary.

For a 1024×1024 texture, the mipmap chain contains the levels: 1024^2, 512^2, 256^2, ..., 2^2, 1^2 (i.e., 11 levels). The total memory usage is only $\frac{4}{3}$ of the original texture.

Mipmaps reduce **aliasing** (flickering of textures seen from afar) and improve performance (distant textures are read from low-resolution levels, which are better utilized by the GPU's memory cache).

Automatic generation in OpenGL:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

Wrapping Modes

When texture coordinates fall outside the $[0, 1]$ range, the behavior depends on the **wrapping mode**:

- **Repeat** (`GL_REPEAT`): the texture repeats indefinitely. The coordinate $u = 2.3$ is interpreted as $u = 0.3$. Useful for tileable textures (bricks, grass, etc.).
- **Mirrored Repeat** (`GL_MIRRORED_REPEAT`): the texture repeats while alternating its orientation, avoiding visible seams at the edges.
- **Clamp to Edge** (`GL_CLAMP_TO_EDGE`): coordinates outside $[0, 1]$ are clamped to the texture boundary. Out-of-bounds pixels take the color of the nearest edge.

OpenGL configuration:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Multiple Textures in the Fragment Shader

All textures (color, normals, specular, etc.) share the same UV coordinates and use the same sampling mechanism. They are simply read from different `sampler2D` variables in the fragment shader:

```
uniform sampler2D diffuse_map;
uniform sampler2D normal_map;
uniform sampler2D specular_map;

void main() {
    vec3 color    = texture(diffuse_map,  fragment.uv).rgb;
    vec3 normal   = texture(normal_map,   fragment.uv).rgb * 2.0 - 1.0;
    float spec    = texture(specular_map, fragment.uv).r;
    // ...
}
```

Advanced Texture Effects

Skybox

A **skybox** is a textured box representing the distant environment (sky, landscape). It is always centered on the camera position, which gives the **impression of an infinite landscape**, as seen below.

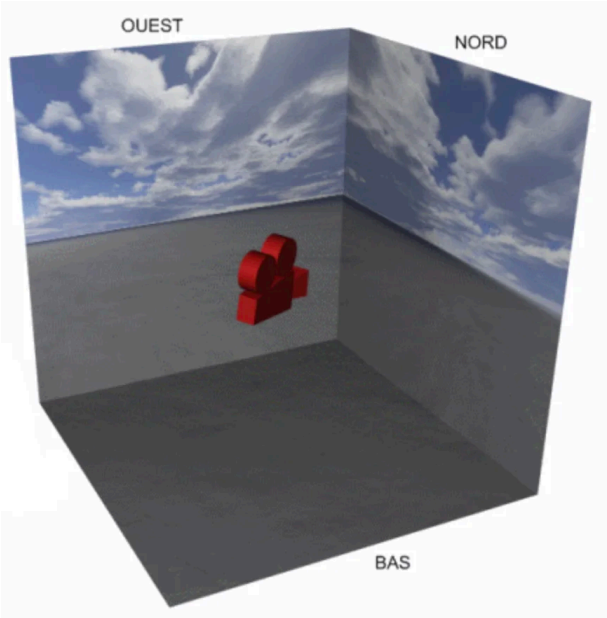


Figure 47: Skybox: a textured box simulating an infinite environment.

The implementation principle is as follows:

1. Render the skybox **first**, with depth buffer writing disabled.
2. Then render all other objects in the scene (which always appear in front of the skybox).

```
void display() {
    glDepthMask(GL_FALSE);   // disable depth buffer writing
    draw(skybox, environment);

    glDepthMask(GL_TRUE);    // re-enable writing
    // draw other objects ...
}
```

Environment Mapping

Environment mapping simulates the reflection of the environment (skybox) on a shiny surface. This gives the impression that the object reflects the world around it.

The principle, shown in the diagram below, is to compute for each fragment the reflection direction of the view vector with respect to the normal, then read the corresponding color from the skybox texture.

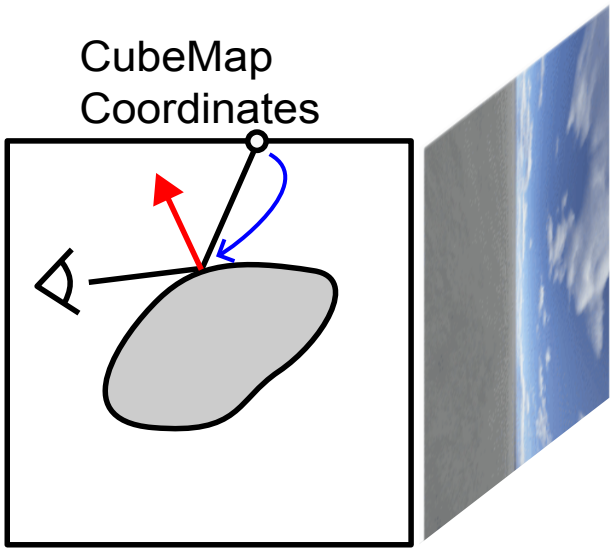


Figure 48: Environment mapping: reflection of the skybox on a surface.

```
vec3 V = normalize(camera_position - fragment.position);
vec3 R_skybox = reflect(-V, N);
vec4 color_env = texture(image_skybox, R_skybox);
```

Normal Mapping

Normal mapping consists of modifying the surface normals using an image (the **normal map**) to simulate geometric details finer than the mesh triangles. The actual geometry of the triangle does not change: only the normal used in the illumination computation is modified, which creates the illusion of relief visible in the following comparison.

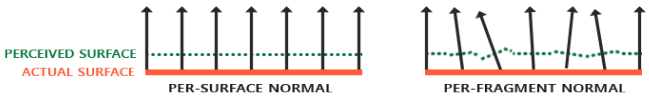


Figure 49: Normal mapping: the geometry remains flat (left) but the illumination simulates relief (right).

Tangent Space

The normals of a normal map are not stored in world coordinates (which would depend on the object's orientation), but in a local frame of reference relative to the triangle called **tangent space** (T, B, N):

- T (**tangent**): direction aligned with the texture's u axis on the surface.
- B (**binormal** or bitangent): direction aligned with the texture's v axis.
- N (**normal**): geometric normal of the surface.

This frame is defined by the texture coordinates (u, v) and the triangle vertex positions:

$$\begin{cases} p_0 - p_1 = (u_0 - u_1)\,T + (v_0 - v_1)\,B \\ p_2 - p_1 = (u_2 - u_1)\,T + (v_2 - v_1)\,B \\ N = T \times B \end{cases}$$

This system of two vector equations (i.e., 6 scalar equations) allows solving for T and B (6 unknowns). In matrix notation, for the x components for example:

$$\begin{pmatrix} T_x \\ B_x \end{pmatrix} = \frac{1}{(u_0 - u_1)(v_2 - v_1) - (u_2 - u_1)(v_0 - v_1)} \begin{pmatrix} v_2 - v_1 & -(v_0 - v_1) \\ -(u_2 - u_1) & u_0 - u_1 \end{pmatrix} \begin{pmatrix} (p_0 - p_1)_x \\ (p_2 - p_1)_x \end{pmatrix}$$

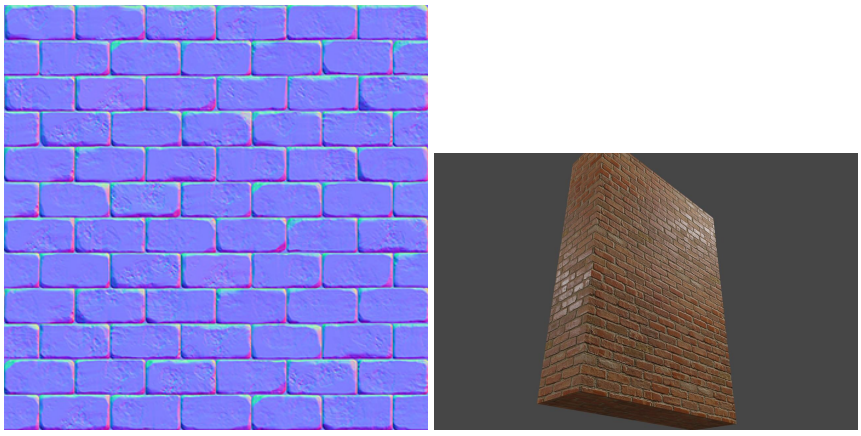
and similarly for the y and z components. The vectors T and B are then normalized.

Normal Map Encoding

Each pixel of the normal map stores a perturbed normal as $(r, g, b) \in [0, 1]^3$ components, encoding the tangent space components on $[-1, 1]$:

$$n_{\text{tangent}} = 2 \begin{pmatrix} r \\ g \\ b \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

The b (blue) component corresponds to the geometric normal direction N. This is why normal maps appear **bluish**: most normals are close to $(0, 0, 1)$ in tangent space, which gives $(0.5, 0.5, 1.0)$ in RGB. This encoding can be observed in the images below.



Normal map: normals are color-encoded in tangent space (left). Application on a brick wall (right).

The final normal in world coordinates is obtained by a change of basis:

$$n_{\text{world}} = \text{TBN} \times n_{\text{tangent}} = T \cdot n_x + B \cdot n_y + N \cdot n_z$$

where $\text{TBN} = (T \mid B \mid N)$ is the 3×3 matrix whose columns are the tangent frame vectors.

GLSL Implementation

Vertex shader: computation of the tangent frame and passing it to the fragment shader.

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;
layout(location = 2) in vec2 vertex_uv;
layout(location = 3) in vec3 vertex_tangent;

out struct fragment_data {
    vec3 position;
    vec2 uv;
    mat3 TBN;
} fragment;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    vec4 pos_world = model * vec4(vertex_position, 1.0);
    mat3 normalMatrix = mat3(transpose(inverse(model)));

    vec3 T = normalize(normalMatrix * vertex_tangent);
    vec3 N = normalize(normalMatrix * vertex_normal);
    vec3 B = cross(N, T);

    fragment.position = pos_world.xyz;
    fragment.uv = vertex_uv;
    fragment.TBN = mat3(T, B, N);

    gl_Position = projection * view * pos_world;
}
```

Fragment shader: reading the normal map and computing illumination.

```
in struct fragment_data {
    vec3 position;
    vec2 uv;
    mat3 TBN;
} fragment;

uniform sampler2D normal_map;
uniform vec3 light_position;

out vec4 FragColor;

void main() {
    // Read the normal from the normal map and convert from [0,1] to [-1,1]
    vec3 n_tangent = texture(normal_map, fragment.uv).rgb * 2.0 - 1.0;

    // Transform to world space
    vec3 N = normalize(fragment.TBN * n_tangent);

    // Illumination with the perturbed normal
    vec3 L = normalize(light_position - fragment.position);
    float diffuse = max(dot(N, L), 0.0);

    // ...
}
```

The advantage of tangent space is that the same normal map can be reused on different objects or different parts of the same object, regardless of their orientation in the world.

Parallax Mapping

Parallax mapping goes further than normal mapping by deforming the **texture coordinates** themselves based on a **height map**. While normal mapping only modifies the illumination (the silhouette remains flat), parallax mapping creates an impression of **geometric offset**: high areas appear to come forward and low areas to recede, especially when viewing the surface at a grazing angle.

Geometric Principle

Consider a fragment on a flat surface. Without parallax mapping, the texture would be read at the (u, v) coordinates of this fragment. But if the surface actually had relief, the view ray would have intersected the surface at a different point, shifted horizontally. Parallax mapping approximates this shift, as shown in the diagram below.

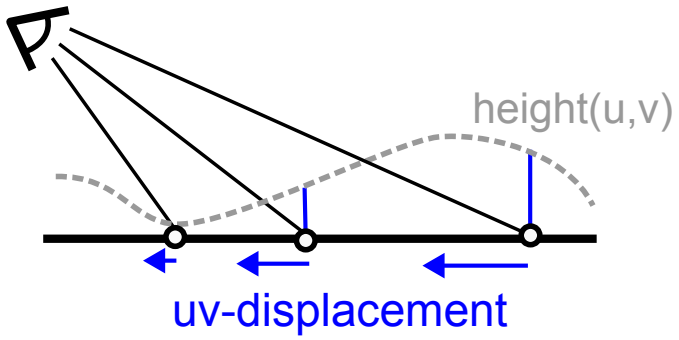


Figure 50: Parallax mapping: the view ray (in red) should intersect the relief surface at point B, but the fragment is at A. The texture coordinates are shifted to read at point B.

Let V be the view vector expressed in tangent space and h the height read from the height map at the current point. The texture coordinate offset is:

$$(u', v') = (u, v) + h \cdot \frac{(V_x, V_y)}{V_z}$$

The division by V_z amplifies the offset at grazing angles (when V_z is small), which corresponds to the expected physical behavior: parallax is more visible at oblique angles.

GLSL Implementation

Fragment shader with parallax mapping:

```
in struct fragment_data {
    vec3 position;
    vec2 uv;
    mat3 TBN;
} fragment;

uniform sampler2D diffuse_map;
uniform sampler2D height_map;
uniform sampler2D normal_map;
uniform vec3 camera_position;
uniform float height_scale;  // controls the relief amplitude (~0.05–0.1)

out vec4 FragColor;

void main() {
    // View vector in tangent space
    vec3 V_world = normalize(camera_position - fragment.position);
    vec3 V_tangent = normalize(transpose(fragment.TBN) * V_world);

    // Read the height and compute the offset
    float h = texture(height_map, fragment.uv).r;
    vec2 uv_offset = h * height_scale * V_tangent.xy / V_tangent.z;
    vec2 uv_parallax = fragment.uv + uv_offset;

    // Use the offset coordinates for the texture and normal map
    vec3 color = texture(diffuse_map, uv_parallax).rgb;
    vec3 n_tangent = texture(normal_map, uv_parallax).rgb * 2.0 - 1.0;
    vec3 N = normalize(fragment.TBN * n_tangent);

    // Illumination with the offset normal and color
    // ...
}
```

Steep Parallax Mapping

Simple parallax mapping is a first-order approximation that works well for shallow relief. For more pronounced relief, it produces artifacts (sliding, distortions). **Steep Parallax Mapping** (or Parallax Occlusion Mapping) improves quality by marching the view ray **step by step** through the height map:

1. Discretize the view ray into N steps in tangent space.
2. At each step, compare the current ray height with the height read from the height map.
3. Stop when the ray passes below the surface (the ray height becomes less than the height map value).
4. Interpolate between the last two steps to obtain a precise intersection point.

```
vec2 steep_parallax(vec2 uv, vec3 V_tangent, float height_scale) {
    const int num_steps = 32;
    float step_size = 1.0 / float(num_steps);
    float current_depth = 0.0;
    vec2 delta_uv = V_tangent.xy / V_tangent.z * height_scale / float(num_steps);

    vec2 current_uv = uv;
    float current_height = texture(height_map, current_uv).r;

    for(int i = 0; i < num_steps; i++) {
        if(current_depth >= current_height) break;
        current_uv += delta_uv;
        current_height = texture(height_map, current_uv).r;
        current_depth += step_size;
    }

    return current_uv;
}
```

This method is more expensive (one texture read per step), but produces visually convincing results even for significant relief, with correct handling of self-occlusion of relief details.