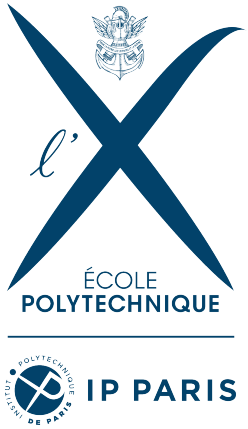
Version du March 14, 2026



[Partie Graphique] Programmation C++

Application à l'Informatique Graphique

[CSC-43043-EP]

2026

Damien ROHMER - damien.rohmer@polytechnique.edu

Contents

1 Introduction à l'Informatique Graphique

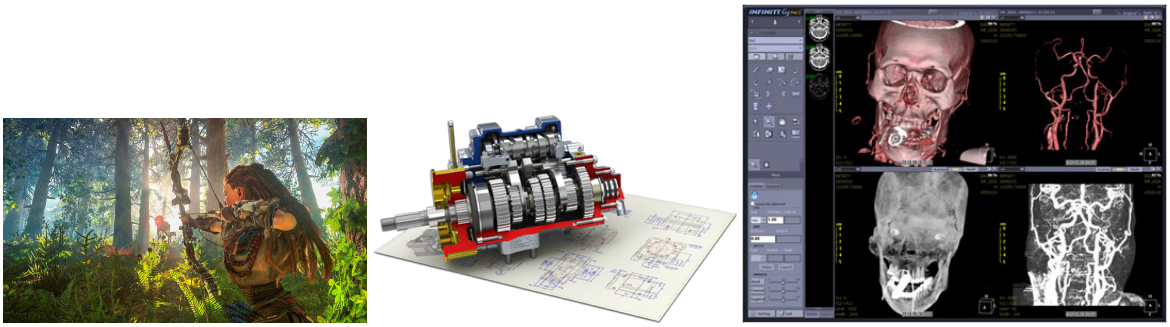
1.1 Généralités

Contexte

Les modèles 3D sont omniprésents dans de nombreux domaines :

- **Applications de loisirs** : cinéma et VFX, jeux vidéos, réalité virtuelle.
- **Autres domaines** : CAO (Conception Assistée par Ordinateur), imagerie médicale, simulation physique.

Quelques exemples de ces applications sont présentés ci-dessous.



Exemples d'applications 3D : jeu vidéo (gauche), CAO (centre), imagerie médicale (droite).

La création et la manipulation de contenus 3D restent cependant des tâches complexes et coûteuses :

- Les outils de modélisation 3D (Maya, Blender, 3DS Max, etc.) se sont améliorés mais restent très techniques (environ 3 ans de formation pour un infographiste). L'interface de Blender, visible sur la figure suivante, en donne un aperçu.

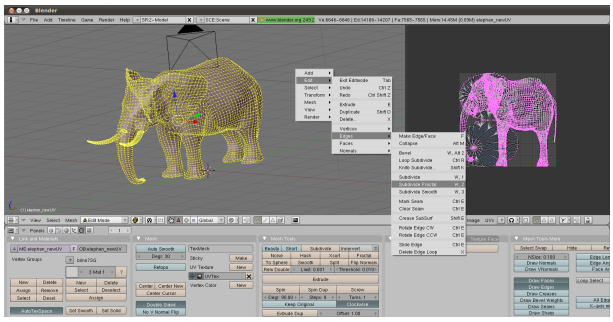


Figure 1: Interface du logiciel de modélisation 3D Blender.

- La quantité et la qualité de contenu demandées ont augmenté plus rapidement que les capacités des outils.
- Les coûts de production n'ont jamais été aussi élevés : un film d'animation ou un jeu vidéo AAA peut coûter plus de 100 millions de dollars et mobiliser des centaines de développeurs et des milliers d'artistes pendant plusieurs années.

Les outils automatiques basés sur l'IA générative commencent à émerger, mais restent très peu contrôlables et avec un coût environnemental important. Ils sont encore largement expérimentaux pour de vraies productions. La contrôlabilité, la qualité et l'efficience restent des défis majeurs.

Définition de l'informatique graphique

L'**informatique graphique** (Computer Graphics) désigne l'ensemble des sciences et techniques permettant de générer et manipuler des données visuelles :

- De manière efficace en temps, énergie et espace mémoire.
- De manière contrôlable et interactive.
- En permettant de générer des images de qualité.

Les applications sont nombreuses : loisirs, design, simulation de phénomènes naturels, médicaux, biologiques, etc.

L'informatique graphique se structure autour de trois grands domaines :

1. **Modélisation (Modeling)** : création et représentation de formes 3D. Cela recouvre la conception de la géométrie des objets (leur forme, leur structure) ainsi que leurs attributs visuels (textures, matériaux). La modélisation peut être manuelle (artiste utilisant un logiciel comme Blender ou Maya) ou procédurale (génération algorithmique de formes).
2. **Animation** : mise en mouvement des objets et personnages au cours du temps. L'animation englobe aussi bien les techniques cinématiques (déplacement explicite des objets image par image ou par interpolation de poses clés) que les techniques physiques (simulation de forces, gravité, collisions, fluides, tissus, etc. qui génèrent le mouvement automatiquement à partir de lois physiques).
3. **Synthèse d'images / Rendu (Rendering)** : génération d'images 2D à partir d'une scène 3D. Le rendu prend en entrée la description de la géométrie, des matériaux, des lumières et de la caméra, puis calcule l'image résultante. On distingue le rendu **temps réel** (interactif, typiquement $\geq$ 25-60 images par seconde, utilisé dans les jeux vidéos et la visualisation) et le rendu **hors-ligne** (photo-réaliste, pouvant prendre de quelques secondes à plusieurs heures par image, utilisé au cinéma et en architecture).

Ces trois domaines sont illustrés ci-dessous.

Modélisation (gauche), animation (centre) et rendu (droite).

Thèmes reliés et vocabulaire

Sous-thèmes

- **Réalité virtuelle (VR)** : interaction avec une scène 3D à l'aide d'un casque de réalité virtuelle et de capteurs.
- **Réalité augmentée (AR)** : superposition d'éléments virtuels sur une vue du monde réel.
- **Réalité mixte / étendue (XR)** : combinaison de VR et AR.
- **Visualisation (scientifique)** : représentation visuelle de données dans un objectif d'explication ou de communication.
- **Simulation 3D** : méthode d'animation basée sur un modèle inspiré de la physique.
- **CAO / CAD (Computer Aided Design)** : conception de design à partir de modeleurs 3D.

Autre vocabulaire

- **Infographie** : utilisation de logiciels de création de données visuelles (Maya, Blender, 3DS Max, etc.).
- **Analyse / traitement d'images** : extraction d'information à partir d'images. C'est en quelque sorte l'opposé de la synthèse d'images.
- **Computer Vision** : analyse d'images ou de vidéos pour comprendre une scène comme un humain le ferait (reconnaissance de formes, de mouvements, etc.).

1.2 Concepts d'une scène 3D

Description d'une scène 3D

Une scène 3D est décrite par trois composantes fondamentales :

- **Modèle 3D** : une surface ou un volume représentant les objets de la scène. Chaque objet possède une géométrie (forme) et des attributs visuels (couleur, matériau, texture). La scène peut contenir un ou plusieurs objets, chacun positionné et orienté dans un espace 3D commun appelé **espace monde** (world space).
- **Source de lumière** : une ou plusieurs sources éclairant la scène. Une source de lumière est caractérisée par sa position (ou direction), son intensité et sa couleur. Les types courants incluent les lumières ponctuelles (émission depuis un point), les lumières directionnelles (rayons parallèles, simulant le soleil), et les lumières surfaciques (émission depuis une surface, pour des ombres plus douces). L'interaction entre la lumière et les matériaux des objets détermine l'apparence visuelle de la scène.
- **Caméra** : le point de vue depuis lequel la scène est observée. La caméra est définie par sa position dans l'espace, son orientation (direction de visée), et ses paramètres optiques (champ de vision, focale, ratio d'aspect). Elle détermine quels objets sont visibles et comment ils sont projetés sur l'image finale.

La **sortie** du processus de rendu est une **image 2D** correspondant à ce que la caméra ”voit” de la scène 3D. Le schéma suivant résume l'agencement de ces composantes.

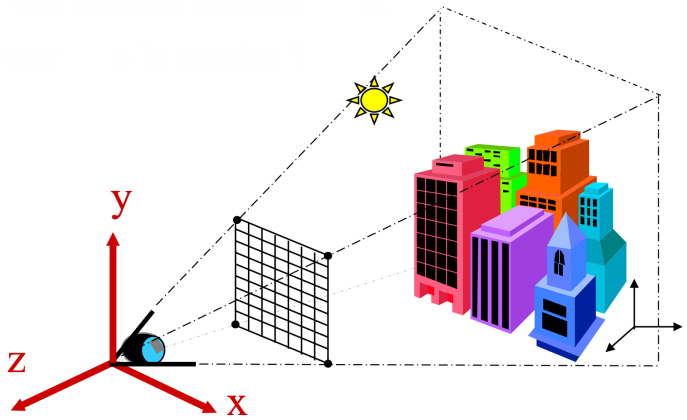


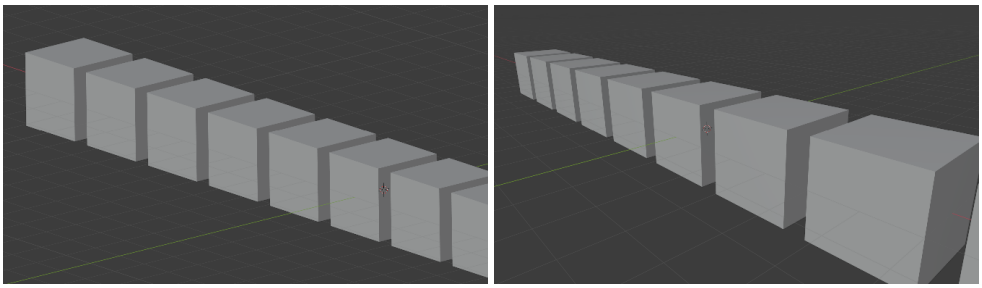
Figure 2: Schéma d'une scène 3D : objets, lumière et caméra dans l'espace monde.

Perception d'une scène 3D sur une image

Un enjeu fondamental de l'informatique graphique est de donner l'illusion de la profondeur et du volume sur un écran qui est intrinsèquement un support 2D. Plusieurs phénomènes visuels contribuent à cette perception.

Effet de perspective

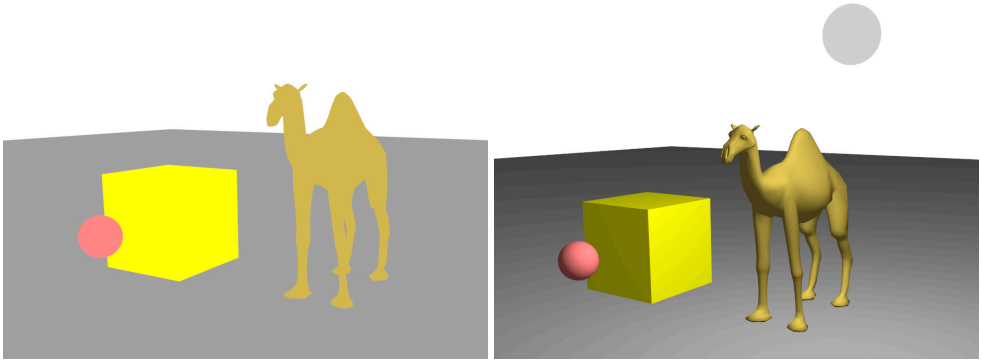
Les objets éloignés apparaissent plus petits que les objets proches. Deux lignes parallèles dans la scène convergent vers un point de fuite sur l'image. Cet effet est directement lié à la projection perspective réalisée par la caméra (voir section sur les coordonnées généralisées).



Projection orthographique (gauche) vs projection perspective (droite).

Illumination et ombrage

La couleur d'un point de la surface varie en fonction de l'orientation de cette surface par rapport à la source lumineuse. Une surface faisant face à la lumière apparaît claire, tandis qu'une surface tournée de l'autre côté apparaît sombre. Ce dégradé de luminosité, appelé **ombrage** (shading), donne des indices essentiels sur la courbure et le volume des objets. Par opposition, un objet affiché avec une couleur uniforme (sans ombrage) paraît plat, comme un disque plutôt qu'une sphère.



Couleur uniforme (gauche) vs illumination diffuse (droite) : l'ombrage révèle la géométrie.

Occultation

Les objets proches masquent les objets situés derrière eux. Ce phénomène simple mais puissant fournit un indice direct de l'ordre en profondeur des objets dans la scène.

Ombres portées

L'ombre qu'un objet projette sur un autre objet ou sur le sol ancre visuellement l'objet dans la scène et donne des informations sur sa position relative et la direction de la lumière.

Définition d'une image

Une image est une **grille 2D de pixels** de dimension $N_x \times N_y$.

- Chaque pixel correspond à une couleur.
- Le format standard est **RGB** (Red, Green, Blue), parfois étendu en **RGBA** (avec un canal alpha pour la transparence).
- En flottants : $c = (r, g, b)$ avec $r, g, b \in [0, 1]$.
  - Exemples : $(0, 0, 0)$ = noir, $(1, 0, 0)$ = rouge, $(1, 1, 0)$ = jaune, $(1, 1, 1)$ = blanc.
  - Il s'agit d'un format **additif** des couleurs.

Autres formats courants :

- **RGB en entiers** : $(r, g, b) \in [\![0, 255]\!]^3$.
- **CMYK** : Cyan, Magenta, Yellow, Black (utilisé pour l'impression).
- **HSV** : Hue, Saturation, Value.
- **YPbPr** : Luminance, Chrominance bleue, Chrominance rouge.
- **CIELAB** : espace perceptuellement uniforme.

Remarque : l'espace RGB n'est pas perceptuellement uniforme (deux couleurs proches en distance RGB ne sont pas nécessairement proches visuellement). Le cube RGB est représenté sur la figure ci-dessous.

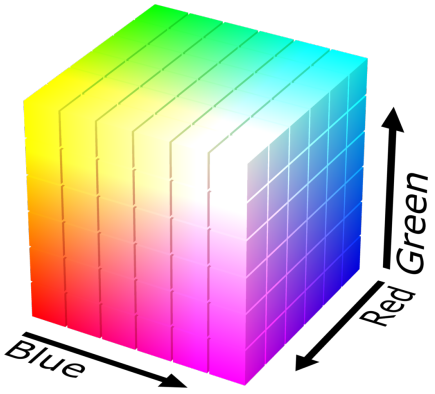


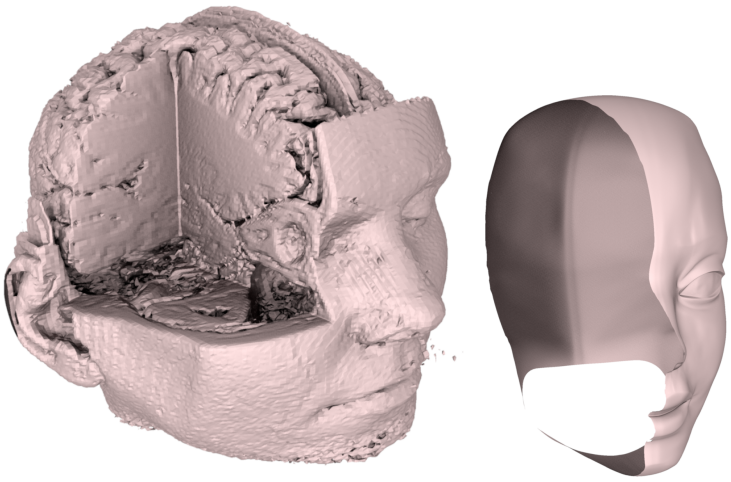
Figure 3: Visualisation du cube des couleurs RGB.

Représentation de modèles 3D

Un objet 3D peut être représenté de deux manières fondamentales :

- **Volume** : description complète de l'objet, incluant l'intérieur. On peut par exemple associer une densité, une température ou tout autre attribut physique en chaque point du volume. Cette représentation est indispensable en imagerie médicale (IRM, scanner), en simulation de fluides (fumée, nuages) et en physique des matériaux. Cependant, elle est très coûteuse en mémoire : un volume de résolution 256^3 contient déjà plus de 16 millions de voxels (pixel volumique).
- **Surface** : uniquement la partie extérieure et visible de l'objet. Seule la ”peau” de l'objet est décrite, sans information sur l'intérieur. Cette représentation est beaucoup plus économe en mémoire et surtout bien plus efficace pour le rendu sur GPU, car seule la surface contribue à l'image finale.

En informatique graphique temps réel, on utilise majoritairement des représentations par **surface**. Les représentations volumiques sont réservées à des cas spécifiques (effets atmosphériques, données médicales, simulation physique). La figure suivante compare ces deux approches.



Représentation volumique (gauche) vs représentation surfacique (droite).

Surfaces explicites et implicites

Il existe deux familles fondamentales de description mathématique d'une surface :

- **Représentation explicite (paramétrique)** : la surface est décrite par une fonction qui associe à chaque couple de paramètres (u, v) un point 3D de la surface : $S(u, v) = (x(u, v), y(u, v), z(u, v))$.
  - Avantage : la notion de voisinage est naturelle (deux points proches en (u, v) sont proches sur la surface), le calcul différentiel est direct (on peut calculer des tangentes et des normales par dérivation de S).
  - Inconvénient : il est difficile de représenter des surfaces de topologie complexe (avec des trous, des branches) par une seule fonction paramétrique.
  - Exemple (sphère de rayon R) :

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

- **Représentation implicite** : la surface est définie comme l'ensemble des zéros d'un champ scalaire :

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

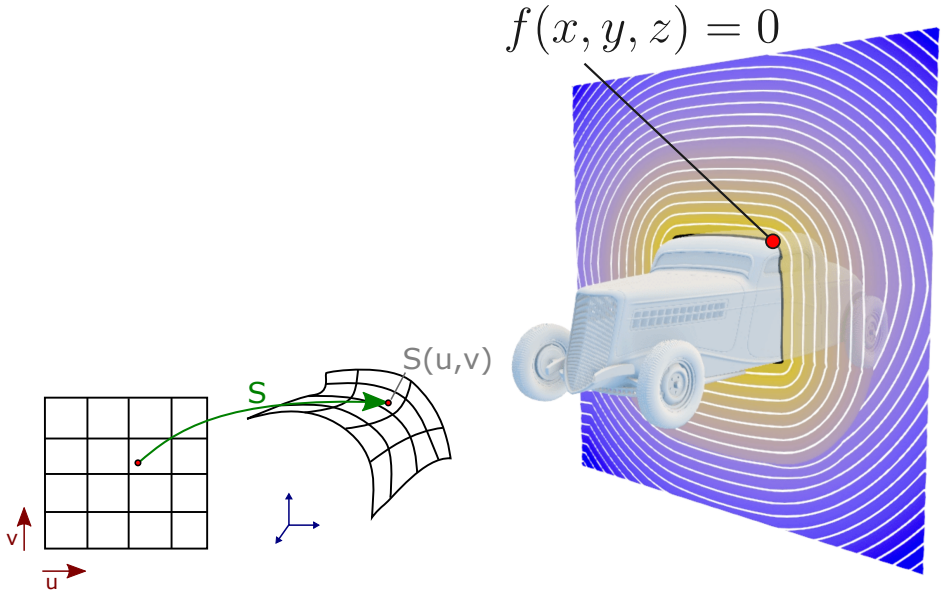
  La fonction f attribue une valeur à chaque point de l'espace ; la surface est le lieu où cette valeur est nulle.

  - Avantage : adaptation topologique naturelle. Lorsque deux objets implicites se rapprochent, leurs isosurfaces peuvent fusionner ou se séparer automatiquement, sans intervention manuelle. Cela est très utile pour simuler des fluides, de la métamatière, ou des opérations booléennes (union, intersection, différence de formes).
  - Inconvénient : il est plus difficile de parcourir la surface (pas de paramétrisation naturelle) et le rendu direct est plus coûteux.
  - Exemple (sphère de rayon R centrée à l'origine) :

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

    Les points où $f < 0$ sont à l'intérieur de la sphère, ceux où $f > 0$ sont à l'extérieur.

Les deux types de représentation sont comparés sur le schéma ci-dessous.



Surface explicite paramétrique (gauche) vs surface implicite (droite).

En pratique, pour des formes complexes comme un personnage ou un véhicule, aucune formule analytique simple ne peut décrire la surface. On recourt alors à des approximations discrètes.

Discrétisation et primitives

En pratique, les surfaces arbitraires sont rarement décrites par une seule fonction analytique. On utilise des **approximations par morceaux** à l'aide de primitives et d'éléments discrets.

Les propriétés idéales d'une description sont :

- Approximation précise de surfaces arbitraires.
- Faible nombre de primitives.
- Rendu efficace par le GPU.
- Manipulation aisée pour la modélisation.

Exemples de modèles :

- **Maillages (Mesh)** : maillages triangulaires ou polygonaux.
- **Polynômes** : courbes et surfaces de Bézier, Splines, NURBS.
- **Implicites** : voxels, squelettes, RBF, MLS.
- **Point sets**, **Gaussian Splats**.

Les cartes graphiques sont spécialisées pour le rendu de **maillages triangulaires**.

1.3 Rendu 3D

Deux approches de rendu

Lancé de rayons (Ray Tracing)

Le lancé de rayons simule le parcours physique de la lumière dans la scène. Le principe de base est le suivant : pour chaque pixel de l'image, on lance un **rayon** depuis la caméra à travers ce pixel et on cherche le premier objet de la scène que ce rayon intersecte. Une fois le point d'intersection trouvé, on calcule sa couleur en fonction

de l'éclairage, du matériau et éventuellement des réflexions et réfractions (en lançant récursivement de nouveaux rayons).

En pratique, les rayons sont souvent tracés dans le sens inverse de la lumière (de la caméra vers la scène), car la grande majorité des rayons émis par une source lumineuse n'atteignent jamais la caméra. Ce traçage inversé est beaucoup plus efficace.

Avantages :

- Rendu **photo-réaliste** : les ombres douces, les réflexions (miroirs), les réfractions (verre, eau), les caustiques et l'illumination globale (lumière indirecte rebondissant entre surfaces) sont gérés naturellement par le modèle.
- Compatible avec des **surfaces arbitraires** : le seul prérequis est de pouvoir calculer l'intersection d'un rayon avec la surface, ce qui est possible pour de nombreuses représentations (triangles, sphères, surfaces implicites, etc.).

Inconvénients :

- **Coût de calcul** élevé : chaque pixel nécessite potentiellement de nombreux calculs d'intersection et le nombre de rayons secondaires (réflexions, ombres) peut croître exponentiellement. Pour une image de haute qualité (cinéma), le rendu d'une seule image peut prendre de quelques minutes à plusieurs heures.
- Peu adapté au rendu temps réel, bien que les progrès récents des GPU (hardware ray tracing) commencent à le permettre dans certains cas.

Le principe est résumé sur le schéma suivant.

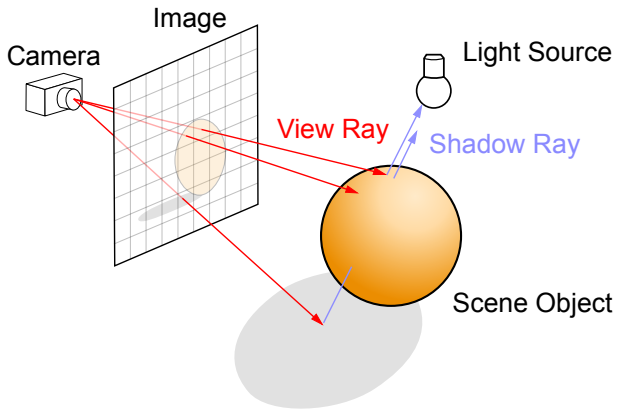


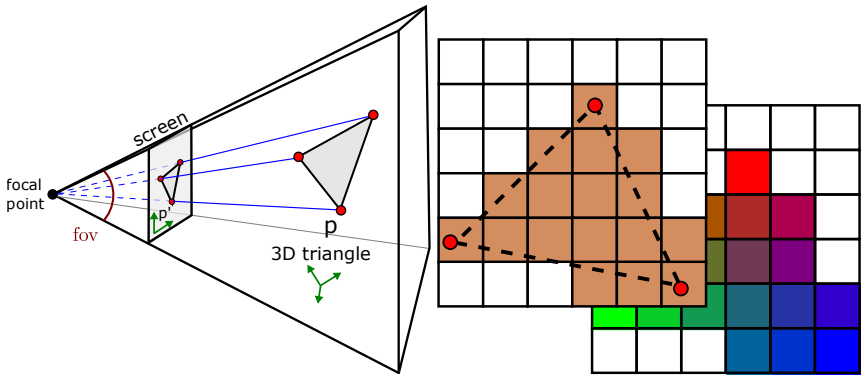
Figure 4: Principe du lancé de rayons.

Projection / Rasterization

L'approche par rasterization adopte une stratégie radicalement différente du ray tracing. Au lieu de partir de chaque pixel et de chercher quel objet est visible, on part de chaque objet (triangle) et on détermine quels pixels il recouvre. Cette approche suppose que la scène est composée de **triangles**. Le processus se déroule en deux étapes :

1. **Projection** : chaque sommet de triangle est projeté depuis l'espace 3D sur le plan 2D de la caméra. Cette opération est réalisée par une multiplication matricielle dans un espace projectif : $p' = M\,p$, où M est la matrice combinant la transformation de la scène, le placement de la caméra et la projection perspective. Cette opération est extrêmement rapide car elle se résume à un produit matrice-vecteur par sommet.
2. **Rasterization** : une fois les triangles projetés en 2D, chaque triangle est ”rempli” pixel par pixel. Pour chaque pixel couvert par le triangle, on calcule ses coordonnées barycentriques et on interpole les attributs des sommets (couleur, normale, coordonnées de texture, etc.) pour obtenir la valeur au pixel. Chaque pixel ainsi traité est appelé un **fragment**.

Ces deux étapes sont illustrées ci-dessous.



Projection des sommets sur le plan caméra (gauche) et rasterization des triangles en pixels (droite).

Un mécanisme de **z-buffer** (tampon de profondeur) permet de gérer l'occultation : pour chaque pixel, on ne conserve que le fragment le plus proche de la caméra, assurant que les objets en avant masquent ceux en arrière.

Avantages :

- Approche dédiée et massivement optimisée sur **GPU**. Les cartes graphiques sont conçues spécifiquement pour accélérer ce pipeline.
- Rendu à vitesse interactive ($\geq$ 25-60 fps), voire bien au-delà sur les GPU modernes.

Inconvénients :

- Limité aux **triangles** (ou à des primitives qui se décomposent en triangles).
- Pas d'effets physiquement corrects natifs : les ombres, la transparence, les réflexions et la lumière indirecte nécessitent des techniques supplémentaires (shadow maps, environment maps, screen-space effects, etc.) qui sont des approximations.

La rasterization est le **standard du rendu temps réel** sur GPU. C'est l'approche utilisée dans tous les jeux vidéos, les applications de visualisation interactive et les logiciels de CAO.

1.4 Maillages triangulaires

Structure

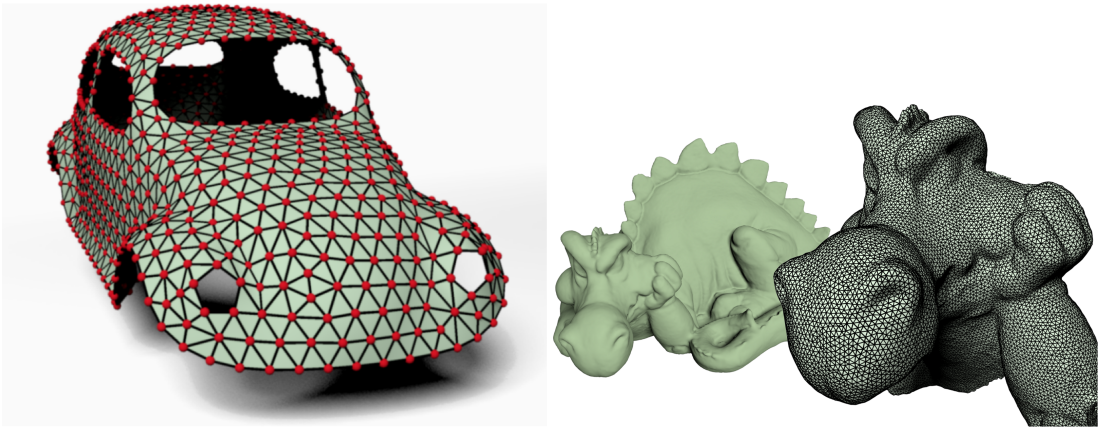
Un maillage triangulaire est la représentation la plus simple et la plus répandue pour approximer une surface continue par un ensemble de triangles. Plus le nombre de triangles est élevé, plus l'approximation est fidèle à la surface originale, mais plus le coût de stockage et de rendu est important. En pratique, un modèle 3D typique dans un jeu vidéo contient de quelques milliers à plusieurs centaines de milliers de triangles. Un modèle haute résolution pour le cinéma peut contenir plusieurs millions de triangles.

Un maillage est décrit par trois types d'éléments :

- **Sommets (vertices)** : les points 3D qui forment les coins des triangles. L'ensemble des sommets est noté $\mathcal{V} = (p_1, \ldots, p_{N_v})$ avec $p_i = (x_i, y_i, z_i) \in \mathbb{R}^3$. Chaque sommet peut porter des attributs supplémentaires : une normale (direction perpendiculaire à la surface en ce point), une couleur, des coordonnées de texture, etc.
- **Faces** : chaque face est un triangle défini par un triplet d'indices de sommets $\mathcal{F} = (f_1, \ldots, f_{N_f})$ avec $f_i = (p_{i_1}, p_{i_2}, p_{i_3})$. Les faces définissent la **connectivité** (ou topologie) du maillage, c'est-à-dire comment les sommets sont reliés entre eux.

- ／ **Arêtes (Edges)** (optionnel) : segments reliant deux sommets adjacents $\mathcal{E} = (e_1, \ldots, e_{N_e})$ avec $e_i = (p_{i_1}, p_{i_2})$. Les arêtes ne sont pas toujours stockées explicitement car elles peuvent être déduites des faces, mais elles sont utiles pour certains algorithmes (subdivision, simplification, etc.).

Des exemples concrets de maillages sont visibles ci-dessous.



Exemples de maillages triangulaires : modèle de voiture (gauche) et modèle de dragon (droite).

Propriétés des triangles

Un triangle T est défini par trois sommets (p_1, p_2, p_3). Le triangle est l'élément de base de la rasterization et possède des propriétés mathématiques très utiles que nous détaillons ci-dessous.

Paramétrisation

Tout point p à l'intérieur du triangle peut être exprimé comme une combinaison linéaire des deux arêtes issues de p_1 :

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

Les paramètres (u, v) forment un système de coordonnées local sur le triangle. Le sommet p_1 correspond à $(0, 0)$, p_2 à $(1, 0)$ et p_3 à $(0, 1)$.

Coordonnées barycentriques

Une manière équivalente et souvent plus pratique de paramétrer un triangle est d'utiliser les **coordonnées barycentriques** (α, β, γ) :

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

Les coordonnées barycentriques ont une interprétation géométrique intuitive : α représente le ”poids” ou l'influence du sommet p_1 sur le point p. Plus p est proche de p_1, plus α est grand (proche de 1) et plus β et γ sont petits. Aux sommets, on a respectivement $(\alpha, \beta, \gamma) = (1, 0, 0)$, $(0, 1, 0)$ et $(0, 0, 1)$. Au barycentre du triangle, les trois coordonnées valent $1/3$.

Le calcul des coordonnées barycentriques pour un point p se fait via le rapport des aires des sous-triangles :

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

avec $A_{123} = \frac{1}{2}\|(p_2 - p_1) \times (p_3 - p_1)\|$ l'aire totale du triangle, et A_{p23}, A_{p31}, A_{p12} les aires des sous-triangles formés par p avec les arêtes opposées :

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

Cette construction géométrique est illustrée sur la figure ci-dessous.

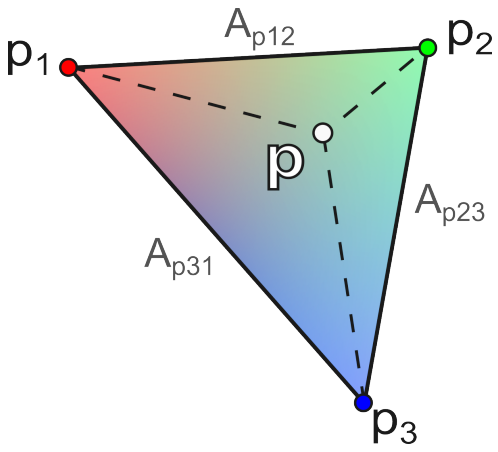


Figure 5: Coordonnées barycentriques : le point p est localisé par les aires des sous-triangles.

Applications

Interpolation barycentrique (linéaire)

Soit un triangle T avec des couleurs (c_1, c_2, c_3) associées aux sommets (p_1, p_2, p_3). La couleur interpolée en un point p intérieur au triangle est :

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

où (α, β, γ) sont les coordonnées barycentriques de p.

Cette interpolation est fondamentale en rendu : elle est utilisée pour interpoler les couleurs, les normales, les coordonnées de texture, etc., à l'intérieur de chaque triangle. Un exemple d'interpolation de couleurs est montré ci-dessous.

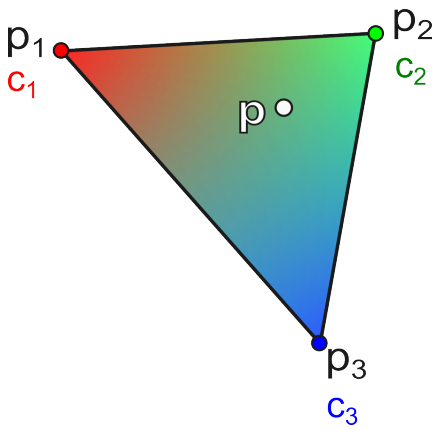


Figure 6: Interpolation barycentrique des couleurs à l'intérieur d'un triangle.

Test d'intersection

Pour tester si un point p est à l'intérieur d'un triangle $T = (p_1, p_2, p_3)$:

1. **Condition nécessaire** : p doit être dans le plan du triangle. On vérifie que $(p - p_1) \cdot n = 0$ avec $n = (p_2 - p_1) \times (p_3 - p_1)$ (normale au triangle).
2. **Condition suffisante** : calcul des coordonnées barycentriques (α, β, γ). On vérifie que $0 \leq \alpha, \beta, \gamma \leq 1$ et $\alpha + \beta + \gamma = 1$.

Le principe de ce test est représenté sur le schéma suivant.

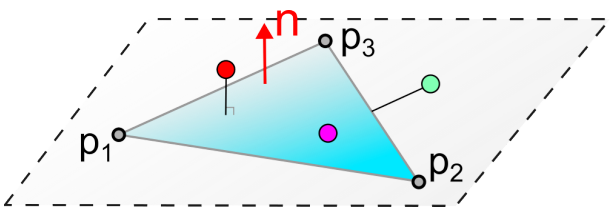


Figure 7: Test d'intersection rayon-triangle.

Description de la connectivité

Considérons un tétraèdre (p_0, p_1, p_2, p_3) comme exemple.

1ère solution : soupe de triangles

Chaque triangle est décrit par ses trois coordonnées, sans partage de sommets :

```
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)]
```

2ème solution : géométrie + connectivité (topologie)

On sépare les positions des sommets et les indices des faces :

```
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)]
```

Cette seconde approche est bien plus efficace en mémoire car les sommets partagés ne sont stockés qu'une seule fois. Dans l'exemple du tétraèdre, la première solution stocke $4 \times 3 = 12$ triplets de coordonnées (soit 36 flottants), tandis que la seconde ne stocke que 4 sommets (12 flottants) plus 4 triplets d'indices (12 entiers). Pour des maillages de grande taille, le gain est considérable. C'est la représentation standard utilisée en pratique et par les APIs graphiques (OpenGL, Vulkan, etc.).

Remarque : l'ordre des indices dans chaque face définit l'**orientation** de la face. Par convention (dite de la **main droite** ou **counterclockwise**), lorsqu'on regarde la face depuis l'extérieur de l'objet, les sommets apparaissent dans le sens trigonométrique (anti-horaire). Cette orientation permet de calculer la **normale sortante** de la face par un produit vectoriel : $n = (p_{i_2} - p_{i_1}) \times (p_{i_3} - p_{i_1})$. Le GPU utilise cette information pour le **back-face culling** : les triangles vus de dos (dont la normale est orientée à l'opposé de la caméra) ne sont pas affichés, ce qui réduit le coût de rendu de moitié environ. Cette convention est illustrée ci-dessous.

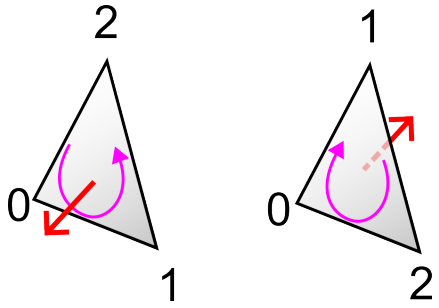


Figure 8: Convention d'orientation des faces (counterclockwise).

Format de fichier : OBJ

Le format **OBJ** (Wavefront) est l'un des formats texte les plus répandus pour stocker des maillages 3D. Sa simplicité le rend facile à lire et à écrire, aussi bien par des humains que par des programmes. Un fichier OBJ est un fichier texte où chaque ligne commence par un mot-clé suivi de valeurs :

- `v x y z` : définit un sommet (vertex) avec ses coordonnées 3D.
- `vt u v` : définit des coordonnées de texture (pour plaquer une image sur la surface).
- `vn x y z` : définit une normale au sommet (direction perpendiculaire à la surface, utilisée pour l'ombrage).
- `f v1 v2 v3` : définit une face triangulaire par les indices de ses sommets. Les indices commencent à **1** (et non 0). On peut aussi référencer les normales et textures avec la syntaxe `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3`.

Exemple minimal (tétraèdre) :

```
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 Coordonnées généralisées

Transformations affines

Les transformations affines sont les opérations fondamentales pour positionner, orienter et dimensionner des objets dans l'espace 3D. En informatique graphique, on les utilise en permanence : pour placer un objet dans la scène, pour animer un personnage (rotation des articulations), pour positionner la caméra, etc.

Translation

La translation déplace un point d'un vecteur constant (t_x, t_y, t_z) :

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

La translation n'est **pas** une transformation linéaire (elle ne peut pas être représentée par une multiplication par une matrice 3×3, car $t(0) \neq 0$ en général). Cette propriété est à l'origine du besoin des coordonnées homogènes décrites plus loin.

Homothétie (Scaling)

Le scaling multiplie chaque coordonnée par un facteur d'échelle :

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

En notation matricielle :

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

Si $s_x = s_y = s_z$, le scaling est **uniforme** (l'objet garde ses proportions). Sinon, il est **non uniforme** (l'objet est déformé, par exemple étiré selon un axe).

Rotation

Une rotation préserve les distances et les angles (c'est une isométrie). Elle est décrite par une matrice 3×3 orthogonale :

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

Les colonnes (et lignes) de R forment une base orthonormée : elles sont mutuellement orthogonales et de norme 1. La condition $\det(R) = 1$ distingue les rotations des réflexions (qui ont un déterminant de -1).

Il existe plusieurs manières de paramétrer une rotation en 3D : angles d'Euler (trois angles successifs autour des axes), axe-angle (un axe et un angle de rotation autour de cet axe), ou quaternions (représentation à 4 composantes, très utilisée en animation pour son efficacité et l'absence de singularités).

Transvection / Cisaillement (Shearing)

Le cisaillement décale une coordonnée proportionnellement à une autre. Par exemple :

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

En notation matricielle :

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

Le cisaillement préserve les volumes ($\det(Sh) = 1$) mais ne préserve pas les angles (ce n'est pas une isométrie). Il est rarement utilisé volontairement en informatique graphique, mais peut apparaître comme sous-produit de certaines décompositions matricielles. Son effet géométrique est visible sur la figure suivante.

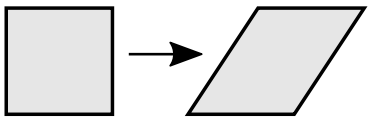


Figure 9: Effet d'un cisaillement sur un carré.

Coordonnées homogènes

La rotation et le scaling sont des transformations **linéaires**, représentables par des matrices 3×3. La translation, en revanche, est une transformation **non linéaire** qui ne peut pas être encodée directement par une matrice 3×3.

Cela pose un problème pratique : en informatique graphique, on enchaîne en permanence des rotations, scalings et translations pour positionner les objets dans la scène. Par exemple, pour placer un objet dans le monde, on peut lui appliquer un scaling (pour ajuster sa taille), puis une rotation (pour l'orienter), puis une translation (pour le positionner). Si seules les rotations et scalings étaient des matrices, il faudrait maintenir deux représentations séparées (une matrice pour la partie linéaire et un vecteur pour la translation), ce qui compliquerait considérablement le code.

Idée : ajouter une coordonnée supplémentaire pour obtenir une représentation **unifiée** en 4D, dans laquelle **toutes** les transformations affines (y compris la translation) s'expriment comme des produits de matrices.

- Un **point** 3D (x, y, z) est représenté par le vecteur 4D $(x, y, z, 1)$.
- Un **vecteur** 3D (x, y, z) est représenté par $(x, y, z, 0)$.

La transformation affine unifiée s'écrit alors comme un produit matriciel 4×4 :

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

avec L la partie linéaire (3×3) et t le vecteur de translation.

Principe en 2D

Pour un point $p = (x, y)$, on ajoute une coordonnée : $p = (x, y, 1)$.

La translation devient une opération linéaire :

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

De même pour les rotations et le scaling, qui s'expriment comme des matrices 3×3 en coordonnées homogènes 2D.

Extension en 3D

En 3D, on utilise des vecteurs à 4 composantes et des matrices 4×4.

Toute séquence de translations, rotations et scalings peut se factoriser en un **unique produit de matrices** :

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

L'avantage est considérable : quelle que soit la complexité de la chaîne de transformations, le résultat est toujours une unique matrice 4×4. Appliquer cette transformation à un point revient à une seule multiplication matrice-vecteur. Cette propriété est exploitée massivement par le GPU, qui peut transformer des millions de sommets en appliquant la même matrice M à chacun d'eux en parallèle.

[Attention] : l'ordre des multiplications est important. Les transformations matricielles ne sont **pas commutatives** en général : $T\,R \neq R\,T$. Par convention, la transformation la plus à droite est appliquée en premier. Ainsi, $M = T\,R\,S$ signifie : d'abord scaling, puis rotation, puis translation.

Points et vecteurs

L'intérêt des coordonnées généralisées est la différenciation naturelle entre **points** et **vecteurs** :

Point : $(x, y, z, \mathbf{1})$ — la translation s'applique.

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

Vecteur : $(x, y, z, \mathbf{0})$ — la translation ne s'applique pas.

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

Ce comportement est cohérent avec les opérations géométriques :

- vecteur $\pm$ vecteur $\rightarrow$ vecteur
- position $\pm$ vecteur $\rightarrow$ position
- position $-$ position $\rightarrow$ vecteur

Cette distinction est résumée sur le schéma ci-dessous.

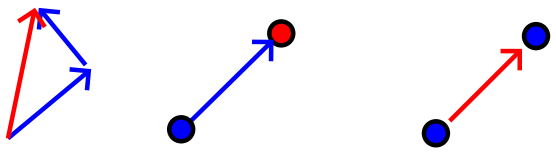


Figure 10: Distinction entre points et vecteurs en coordonnées généralisées.

Espace projectif

Considérons le cas d'un point généralisé avec coordonnée $w \neq 1$: $p_{4D} = (x, y, z, w)$.

La ”renormalisation” (ou projection) sur l'espace des points 3D donne : $p_{3D} = (x/w, y/w, z/w, 1)$.

Exemple :

La somme de deux points donne :

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

Après homogénéisation :

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

Ce qui correspond au **barycentre** des deux points.

L'intérêt de l'espace projectif est d'encoder des opérations **rationnelles** (comme la perspective) par une simple multiplication matricielle.

Application à la perspective

La perspective est modélisée par une division par la profondeur.

En 2D (projection 1D), pour un point (x, y) projeté sur un plan focal à distance f :

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

Ce modèle non linéaire s'écrit de manière linéaire en coordonnées homogènes :

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

Après homogénéisation (division par la dernière composante x), on obtient bien $y' = f\,y/x$.

Les points **réels** (2D ou 3D) sont ceux dont la dernière composante vaut $w = 1$ (obtenus après homogénéisation). Les **vecteurs** correspondent à $w = 0$. Ce mécanisme de projection est représenté sur la figure suivante.

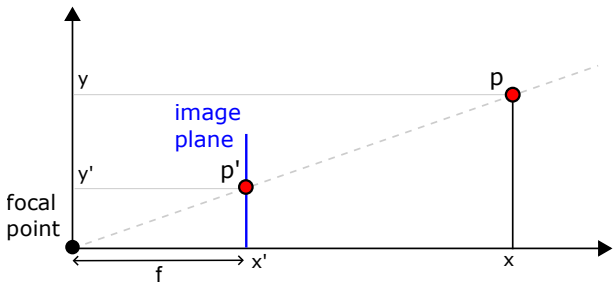


Figure 11: Projection perspective en 2D : le point est projeté sur le plan focal par division par la profondeur.

1.6 OpenGL et notion de Shaders

CPU et GPU

Un ordinateur dispose de deux processeurs principaux pour le calcul :

- **CPU (Central Processing Unit)** : le processeur central, conçu pour exécuter des tâches séquentielles complexes et variées (calcul, branchements conditionnels, accès mémoire, entrées/sorties, gestion du système d'exploitation, etc.). Le CPU dispose d'un petit nombre de cœurs très puissants, chacun capable d'exécuter des instructions arbitraires de manière indépendante. Mémoire associée : RAM.
  - Nombre de cœurs : 2 à 64 (indépendants, chacun avec son propre pipeline d'instructions).
  - RAM : 4 à 64 Go.
  - Optimisé pour la **latence** : chaque tâche individuelle est exécutée aussi vite que possible.
- **GPU (Graphics Processing Unit)** : le processeur graphique, conçu pour exécuter un très grand nombre d'opérations **identiques** en parallèle. L'architecture est de type **SIMD** (Single Instruction, Multiple Data) : une même instruction est appliquée simultanément à des milliers de données différentes. Par exemple, appliquer la même matrice de transformation à chacun des millions de sommets d'un maillage, ou calculer la couleur de chacun des millions de pixels de l'écran. Mémoire associée : VRAM (Video RAM).
  - Nombre de cœurs : 1000 à 16000 (organisés en groupes partageant des ressources).
  - VRAM : 4 à 24 Go.
  - Optimisé pour le **débit** : un très grand volume de données est traité par unité de temps, mais chaque tâche individuelle peut être plus lente que sur CPU.

Les figures ci-dessous montrent l'aspect physique et l'architecture interne de ces deux types de processeurs.

CPU (gauche) et GPU (droite).

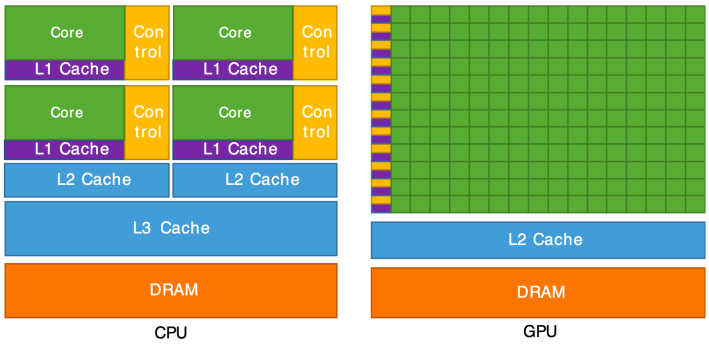
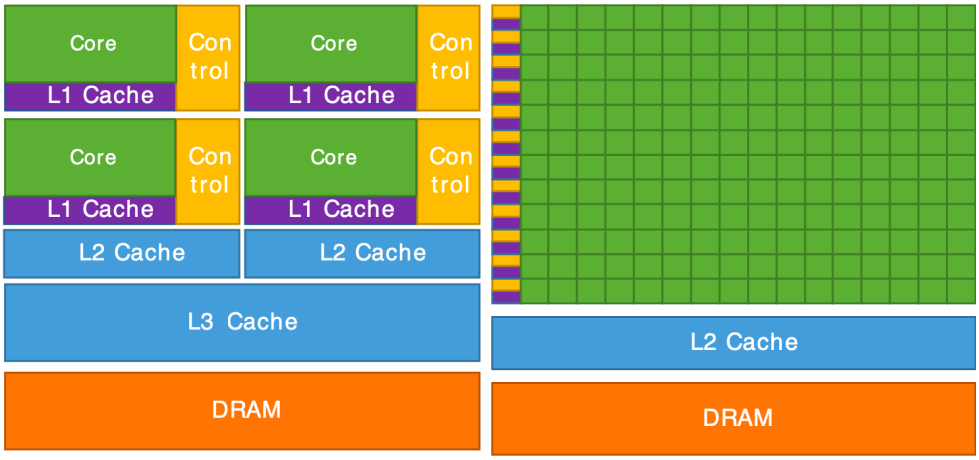


Figure 12: Comparaison des architectures CPU et GPU.



Architecture interne d'un CPU (gauche) et d'un GPU (droite).

Le GPU est en résumé un supercalculateur optimisé pour réaliser la **même opération sur un grand nombre de données** en parallèle. C'est exactement ce dont on a besoin pour le rendu graphique : transformer des millions de sommets (vertex shader) puis colorier des millions de pixels (fragment shader). Le CPU, quant à lui, se charge de la logique du programme (gestion de la scène, chargement des données, interactions utilisateur) et envoie les commandes de rendu au GPU.

OpenGL

OpenGL (Open Graphics Library) est une API (Application Programming Interface) pour communiquer avec le GPU, orientée graphique 3D.

Caractéristiques :

- Bas niveau, haute performance, multi-plateformes.
- Une API n'est ni un logiciel ni une bibliothèque : c'est un ensemble standardisé de variables et de fonctions. Différents systèmes et GPU ont des implémentations différentes d'OpenGL (l'installation dépend du driver graphique).
- Développée et maintenue par le Khronos Group.

Autres APIs graphiques : **Vulkan**, **WebGL**, **WebGPU**, **DirectX** (Windows), **Metal** (Mac).

Communication CPU/GPU avec OpenGL

Le CPU et le GPU possèdent chacun leur propre mémoire (RAM et VRAM respectivement) et ne peuvent pas accéder directement à la mémoire de l'autre. Le processus de rendu nécessite donc une communication explicite entre les deux, orchestrée par l'API OpenGL. Ce processus suit trois grandes étapes :

1. **Préparation des données (CPU)** : le programme C++ (s'exécutant sur le CPU) charge ou calcule les données géométriques de la scène : positions des sommets, couleurs, normales, coordonnées de textures, indices de connectivité, etc. Ces données sont organisées dans des tableaux en RAM.
2. **Envoi des données (CPU → GPU)** : les données sont transférées par blocs depuis la RAM vers la VRAM via des appels OpenGL. Ce transfert est relativement coûteux (le bus entre CPU et GPU a un débit limité), c'est pourquoi on cherche à le minimiser : idéalement, les données statiques (maillages qui ne changent pas) ne sont envoyées qu'une seule fois au démarrage. Seules les données qui changent à chaque image (matrices de transformation, paramètres d'animation) sont transmises à chaque frame.
3. **Exécution des Shaders (GPU)** : une fois les données en VRAM, le GPU exécute les programmes de shaders en parallèle sur chaque sommet puis sur chaque pixel. Le CPU n'intervient plus pendant cette phase : il se contente d'envoyer la commande ”dessine” et le GPU fait le reste de manière autonome. Le résultat est l'image finale, stockée dans le **framebuffer** de la VRAM, qui est ensuite affichée à l'écran.

Les schémas suivants détaillent ce pipeline de communication.

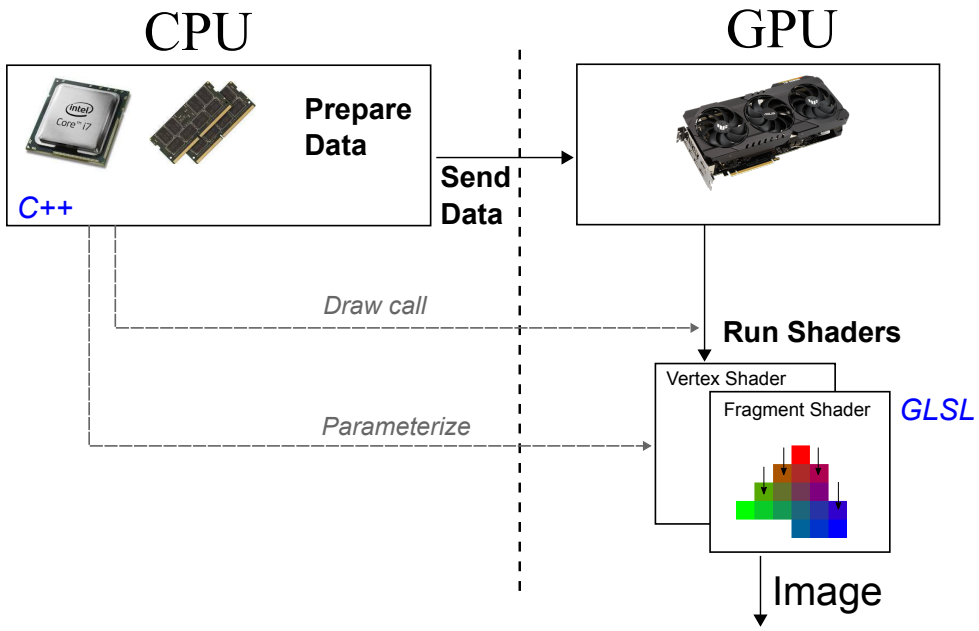


Figure 13: Schéma simplifié de la communication CPU/GPU via OpenGL.

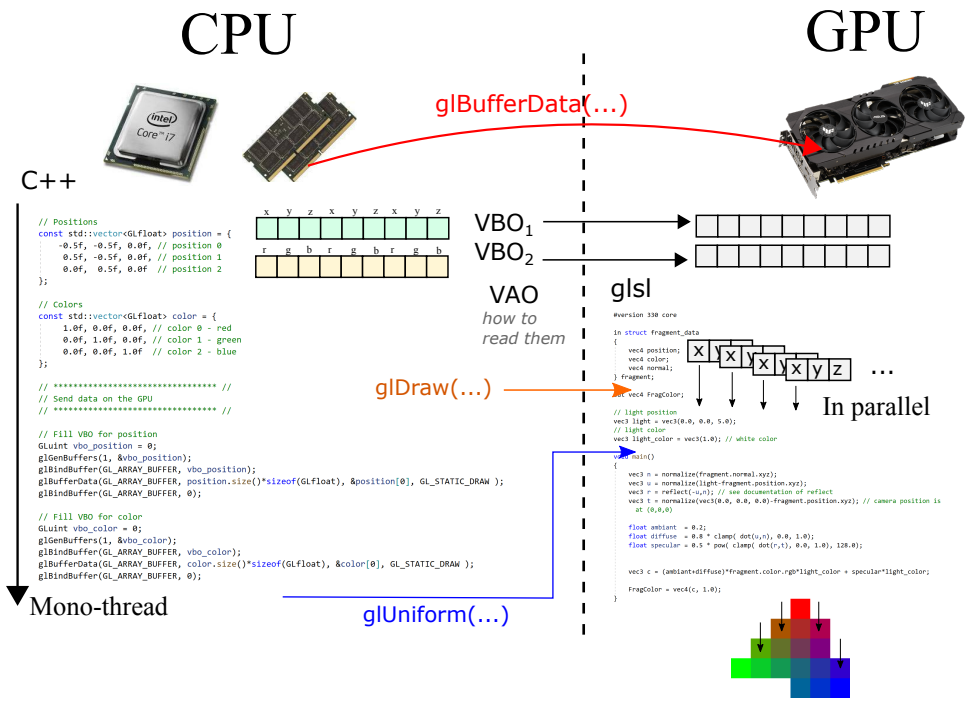


Figure 14: Vue d'ensemble du pipeline de rendu OpenGL.

Shaders

Les **shaders** sont de petits programmes qui s'exécutent directement sur le GPU. Ils sont écrits dans un langage dédié appelé **GLSL** (OpenGL Shading Language), dont la syntaxe est proche du C. Le terme ”shader” vient de ”shading” (ombrage), car leur rôle initial était de calculer l'ombrage des surfaces, mais ils sont aujourd'hui utilisés pour toutes sortes de calculs graphiques.

La particularité fondamentale des shaders est qu'ils sont exécutés **massivement en parallèle** : le même programme est lancé simultanément sur des milliers de sommets ou de pixels. Le programmeur écrit le code pour **un seul** sommet ou pixel, et le GPU se charge de l'exécuter pour tous.

Deux types de shaders principaux interviennent dans le pipeline de rendu :

Vertex Shader

Exécuté une fois **pour chaque sommet** du maillage. Son rôle principal est de transformer la position du sommet depuis l'espace 3D de la scène vers l'espace 2D de l'écran (en appliquant les matrices de transformation et de projection). Il peut aussi calculer et transmettre des attributs (couleurs, normales transformées, coordonnées de texture, etc.) qui seront utilisés par le 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;
}
```

Dans cet exemple :

- `#version 330 core` : indique la version de GLSL utilisée (correspondant à OpenGL 3.3).
- `layout(location = 0) in vec4 position` : déclare un attribut d'entrée (la position du sommet), reçu depuis les données envoyées par le CPU. Le `location = 0` indique l'index du buffer d'attributs.
- `out vec4 vertexColor` : déclare une variable de sortie qui sera transmise au fragment shader. Entre le vertex shader et le fragment shader, cette variable sera automatiquement interpolée par la rasterization.
- `gl_Position` : variable spéciale prédéfinie par OpenGL, dans laquelle le vertex shader doit écrire la position finale du sommet (en coordonnées projetées).

Fragment Shader

Exécuté une fois **pour chaque pixel (fragment)** couvert par un triangle après la rasterization. Son rôle est de déterminer la couleur finale du pixel en fonction des attributs interpolés (couleur, normale, coordonnées de texture), de l'éclairage, des textures, etc. C'est dans le fragment shader que l'on implémente les modèles d'illumination et les effets visuels.

```
#version 330 core

in vec4 vertexColor;
out vec4 fragColor;

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

Dans cet exemple :

- `in vec4 vertexColor` : variable d'entrée provenant du vertex shader. Sa valeur a été automatiquement interpolée par la rasterization entre les trois sommets du triangle courant, en utilisant les coordonnées barycentriques du fragment.
- `out vec4 fragColor` : la couleur de sortie du fragment, qui sera écrite dans l'image finale (framebuffer).
- Ici, le fragment shader est minimal : il se contente de transmettre la couleur interpolée. En pratique, c'est dans le fragment shader que l'on calcule l'éclairage, que l'on échantillonne les textures, et que l'on applique les effets visuels.

Variables uniformes

En plus des attributs (propres à chaque sommet) et des variables interpolées (propres à chaque fragment), les shaders peuvent recevoir des **variables uniformes** (uniform) : des valeurs constantes pour l'ensemble des sommets ou fragments d'un même appel de rendu. Elles sont définies côté CPU et envoyées au GPU. Les exemples typiques sont les matrices de transformation, la position de la caméra, la direction de la lumière, le temps courant, etc.

```
uniform mat4 modelViewProjection; // matrice de transformation (4x4)
uniform vec3 lightDirection;      // direction de la lumière
```

Le pipeline complet fonctionne ainsi :

1. Le **vertex shader** est exécuté en parallèle sur chaque sommet, appliquant les transformations géométriques.
2. Les triangles projetés sont **rasterisés** : découpés en fragments (pixels).
3. Le **fragment shader** est exécuté en parallèle sur chaque fragment, calculant la couleur finale.
4. Le résultat est l'**image finale** affichée à l'écran.

Les attributs transmis du vertex shader au fragment shader (comme `vertexColor`) sont automatiquement **interpolés** de manière barycentrique entre les sommets du triangle, ce qui correspond exactement à l'interpolation barycentrique décrite précédemment. L'ensemble de ce pipeline est résumé sur le schéma ci-dessous.

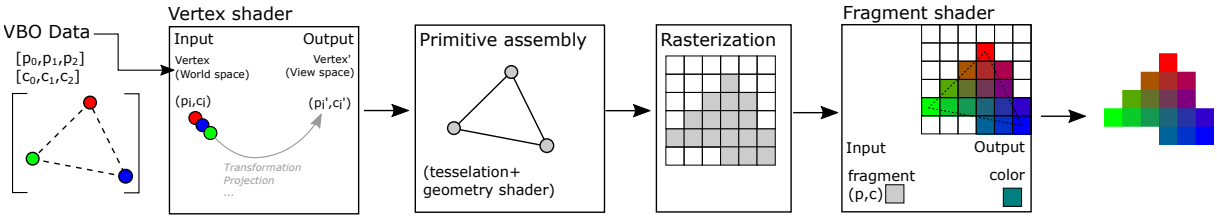


Figure 15: Pipeline de rendu : vertex shader, rasterization, fragment shader.

2 Pipeline de Rendu et Illumination

2.1 Vue d'ensemble du pipeline

De la scène 3D à l'image

Comme vu au chapitre précédent, une scène 3D est décrite par des modèles (surfaces), des sources de lumière et une caméra. Le rendu par rasterization produit une image 2D de cette scène.

Cependant, au niveau du GPU et d'OpenGL, il n'existe **pas de notion explicite** de caméra ni de lumière. Le GPU ne manipule que des **sommets** avec leurs attributs (position, couleur, normale, etc.) en entrée du **vertex shader**, et des **fragments** (pixels) colorés en sortie du **fragment shader**. Tout le travail du programmeur consiste à traduire les concepts de haut niveau (caméra, lumière, matériaux) en opérations sur les sommets et les fragments via les shaders.

Le pipeline de rendu se décompose en trois grandes étapes. La première est la **transformation des sommets**, réalisée par le vertex shader : chaque sommet est projeté depuis l'espace 3D de la scène vers l'espace 2D de l'écran en appliquant les matrices de transformation et de projection. La deuxième étape est la **rasterization** : les triangles projetés en 2D sont convertis en fragments (pixels) par un processus de discrétisation automatique. Enfin, la troisième étape est le **calcul de la couleur**, effectué par le fragment shader : pour chaque fragment, on détermine sa couleur finale en fonction de l'éclairage, du matériau et des textures.

2.2 Transformation des sommets — Projection et perspective

Principe de la projection

OpenGL n'affiche que les triangles dont les sommets se trouvent dans le cube $[-1, 1]^3$. Cet espace normalisé est appelé **Normalized Device Coordinates (NDC)**. Les deux premières composantes $(x_{\text{ndc}}, y_{\text{ndc}})$ correspondent aux coordonnées écran, tandis que z_{ndc} encode la **profondeur**, c'est-à-dire la distance à la caméra dans l'espace image. Ce cube normalisé est représenté sur la figure suivante.

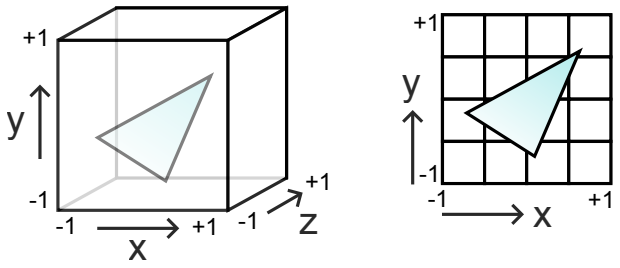


Figure 16: Espace NDC : le cube normalisé dans lequel OpenGL affiche les triangles.

L'objectif de la projection est de **convertir les coordonnées globales** (world space) en coordonnées NDC. Cette conversion doit d'une part définir un **modèle de caméra** qui spécifie quelle partie de l'espace 3D est visible, et d'autre part appliquer la **perspective** pour que les objets éloignés apparaissent plus petits que les objets proches.

Le modèle de perspective le plus courant utilise un volume de visibilité en forme de **tronc de pyramide** (frustum), délimité par un plan proche (z_{near}) et un plan lointain (z_{far}), comme illustré ci-dessous.

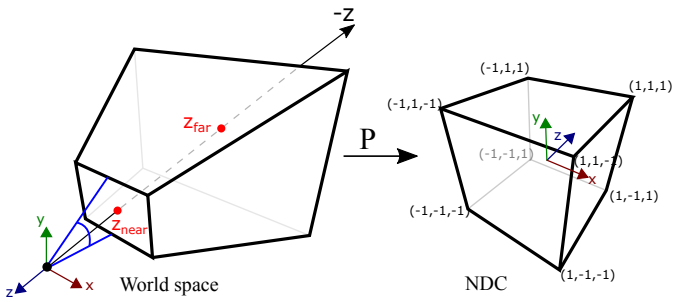


Figure 17: Le frustum : volume de visibilité de la caméra en perspective.

Formulation de la projection

Soit un point de coordonnées (x, y, z) dans l'espace monde. Les coordonnées NDC $(x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}})$ sont calculées comme suit :

Coordonnées x et y :

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

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

où θ est le **champ de vision** (Field of View, FOV) et $r = h/w$ est le ratio d'aspect.

La division par $-z$ produit l'effet de perspective : les objets éloignés (grand $|z|$) sont réduits en taille.

Coordonnée z (profondeur) :

La profondeur NDC est une fonction non linéaire de z :

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

avec les correspondances : $z = -z_{\text{near}} \to z_{\text{ndc}} = -1$ et $z = -z_{\text{far}} \to z_{\text{ndc}} = 1$.

La variation en $1/z$ implique une **précision plus fine** près de la caméra et plus grossière au loin. C'est un choix délibéré : les objets proches nécessitent une meilleure résolution en profondeur pour éviter les artefacts visuels.

Matrice de projection

En coordonnées homogènes, la projection s'exprime comme un produit matriciel 4×4 :

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

avec :

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

Après multiplication, le vecteur obtenu a une composante $w \neq 1$. L'**homogénéisation** (division par w) produit les coordonnées NDC finales. C'est exactement le mécanisme de l'espace projectif vu au chapitre précédent.

Le résultat est un espace **déformé** dans le cube $[-1, 1]^3$. L'image finale correspond à la vue $(x_{\text{ndc}}, y_{\text{ndc}})$ de ce cube, tandis que z_{ndc} représente la profondeur vue depuis la caméra dans l'espace image. Tout ce qui se trouve en dehors du cube $[-1, 1]^3$ est automatiquement écarté par le GPU (clipping).

Z-fighting

La variation en $1/z$ de la profondeur NDC a une conséquence importante : la précision dépend fortement du choix de z_{near}. Le schéma suivant montre cette distribution non uniforme de la précision.

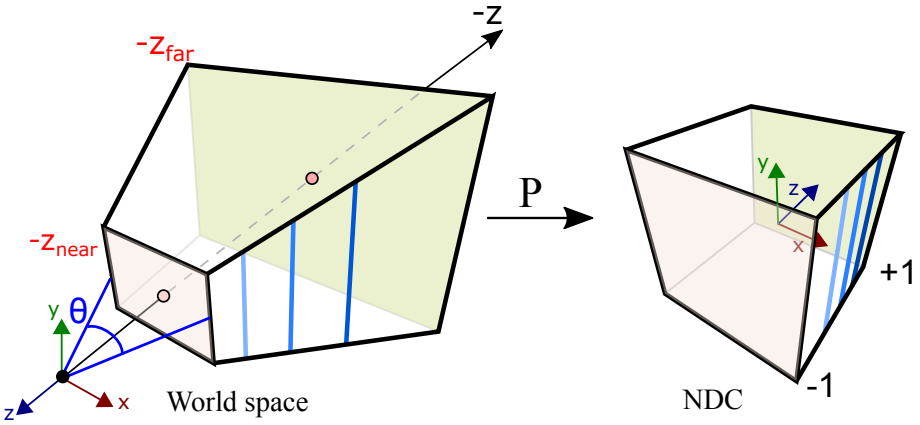


Figure 18: Distribution de la précision en profondeur selon la distance à la caméra.

Si z_{near} est **trop proche de 0**, toute la précision est concentrée sur les distances très proches de la caméra. Les objets éloignés se retrouvent avec des valeurs de profondeur presque identiques après discrétisation. Cela provoque un artefact visuel appelé **z-fighting** (ou depth-fighting) : deux surfaces proches en profondeur "clignotent" de manière aléatoire car le GPU ne peut pas déterminer laquelle est devant. Le graphique ci-dessous illustre la courbe de z_{ndc} en fonction de z.

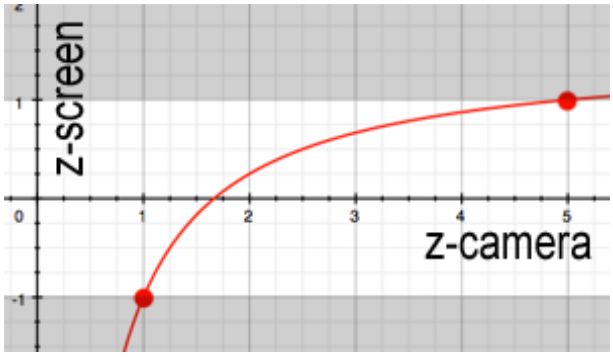


Figure 19: Graphique de la distribution des valeurs de profondeur z_{ndc} en fonction de z.

Règle pratique : choisir z_{near} aussi grand que possible (tout en gardant les objets visibles dans le frustum) pour maximiser la précision en profondeur sur toute la scène.

Matrice de vue (View)

La matrice de vue transforme les coordonnées de l'**espace monde** vers l'**espace caméra** (view space). Elle positionne et oriente la caméra dans la scène, selon le principe illustré ci-dessous.

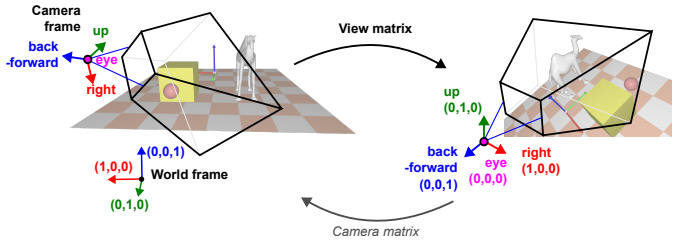


Figure 20: Transformation de l'espace monde vers l'espace caméra.

La caméra est décrite par une matrice 4×4 contenant ses axes locaux et sa position :

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

où $O = (\text{right}, \text{up}, \text{back})$ est la matrice de rotation de la caméra (base orthonormée) et eye est sa position dans le monde.

La matrice de vue est l'**inverse** de la matrice caméra :

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

Cette inversion est très efficace à calculer car O est orthogonale ($O^{-1} = O^T$). La matrice de vue envoie la position de la caméra sur l'origine et aligne ses axes sur les axes du repère.

Matrice de modèle (Model)

La matrice de modèle positionne un objet dans le monde. Elle transforme les coordonnées **locales** de l'objet vers les coordonnées de l'**espace monde** :

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

où R est la matrice de rotation (orientation de l'objet) et t est le vecteur de translation (position de l'objet). La figure suivante montre ce passage des coordonnées locales à l'espace monde.

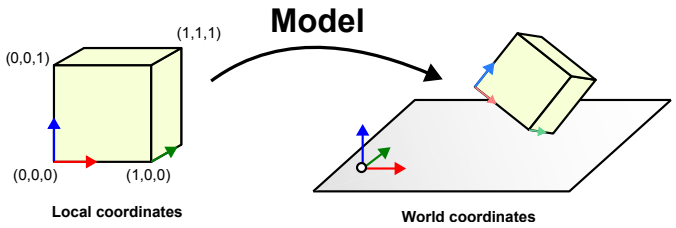


Figure 21: La matrice de modèle place l'objet dans le monde.

L'intérêt de séparer la géométrie de l'objet de sa position est fondamental. Les coordonnées géométriques de l'objet (maillage) sont chargées **une seule fois** en VRAM, et pour déplacer ou orienter l'objet, il suffit de **modifier la matrice Model**, soit une simple matrice 4×4 de 16 flottants. Cette séparation permet également l'**instanciation** : le même maillage peut être affiché à plusieurs endroits de la scène en appliquant des matrices Model différentes, sans dupliquer les données géométriques en mémoire. Par exemple, une forêt composée de milliers d'arbres peut ne stocker qu'un seul maillage d'arbre, chaque instance ayant sa propre matrice Model (position, rotation, échelle).

Synthèse : la chaîne de transformations

La transformation complète d'un sommet, depuis ses coordonnées locales jusqu'à l'espace NDC, est la composition des trois matrices :

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

Le sommet part de ses coordonnées locales $p = (x, y, z, 1)$ dans l'**espace objet**, où la géométrie est définie relativement à l'origine de l'objet. La matrice Model le place dans l'**espace monde** : $p_{\text{world}} = \text{Model} \times p$, en appliquant la rotation, l'échelle et la translation de l'objet. La matrice View transforme ensuite cette position dans l'**espace caméra** : $p_{\text{view}} = \text{View} \times p_{\text{world}}$, où la caméra se trouve à l'origine et regarde dans la direction $-z$. Enfin, la matrice de Projection convertit les coordonnées en **NDC** : $p_{\text{ndc}} = \text{Proj} \times p_{\text{view}}$, en appliquant la perspective et en normalisant les coordonnées dans le cube $[-1, 1]^3$ après homogénéisation (division par w).

Cette chaîne constitue la **première étape** du pipeline graphique, réalisée par le **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);
}
```

Les trois matrices sont envoyées au GPU comme **variables uniformes** depuis le code C++. Le GPU applique ensuite cette transformation **en parallèle sur tous les sommets**. Cela est bien plus efficace que de calculer les nouvelles positions sur le CPU : au lieu de transférer N positions modifiées à chaque image, on ne transfère que trois matrices 4×4 (soit 48 flottants).

```
void main_loop() {
    // Mise à jour des matrices
    glUniform(shader, Model);
    glUniform(shader, View);
    glUniform(shader, Projection);

    draw(mesh_drawable);
}
```

2.3 Rasterization

Principe

La **rasterization** (ou rastérisation, ou facétisation) est la conversion de données vectorielles (triangles définis par des sommets) en éléments discrets : des **pixels** (ou fragments). C'est une étape **automatique et non programmable** dans le pipeline OpenGL.

L'opération fondamentale est la suivante : étant donné un triangle défini par 3 points 2D (après projection), déterminer l'ensemble des pixels qu'il recouvre, comme on peut le voir sur la figure suivante.

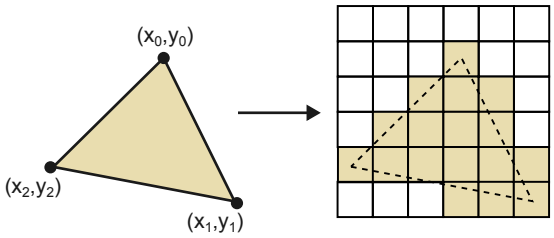


Figure 22: Rasterization d'un triangle : conversion de la forme vectorielle en pixels.

Rasterization de primitives simples

Avant d'aborder la rasterization des triangles, considérons des primitives plus simples pour comprendre le principe de discrétisation.

Point : un seul pixel.

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

Rectangle : deux boucles imbriquées.

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

Segment horizontal, vertical ou diagonal : une seule boucle suffit.

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

Pour un **segment quelconque**, la discrétisation n'est plus triviale : plusieurs approximations pixelisées sont possibles, comme le montre l'image ci-dessous. Il faut choisir un algorithme efficace et déterministe.

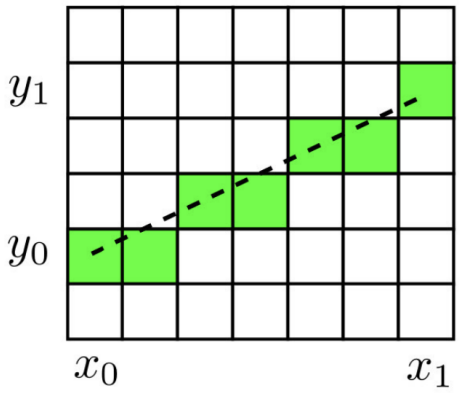


Figure 23: Discrétisation d'un segment : plusieurs approximations possibles.

Algorithme de Bresenham

L'**algorithme de Bresenham** est un algorithme classique et très efficace pour rasteriser un segment. Il repose uniquement sur des opérations entières, ce qui le rend rapide et exact.

Principe : on avance le long de l'axe x pixel par pixel. À chaque pas, on évalue si l'erreur accumulée en y dépasse 0.5 :

- Si oui : on incrémente y de 1 et on corrige l'erreur.
- Si non : on garde le même y.

La progression de l'algorithme est visualisée sur la figure suivante.

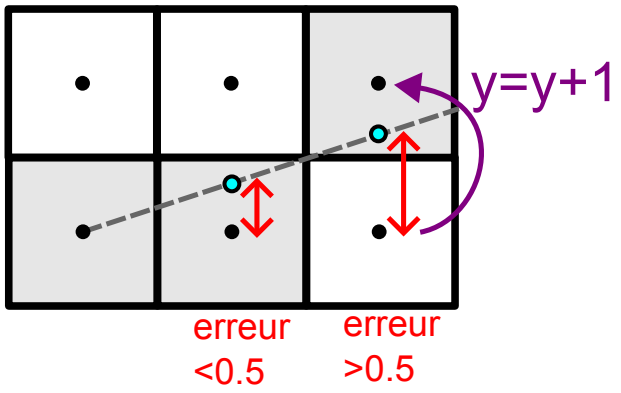


Figure 24: Algorithme de Bresenham : progression le long du segment avec correction d'erreur.

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

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

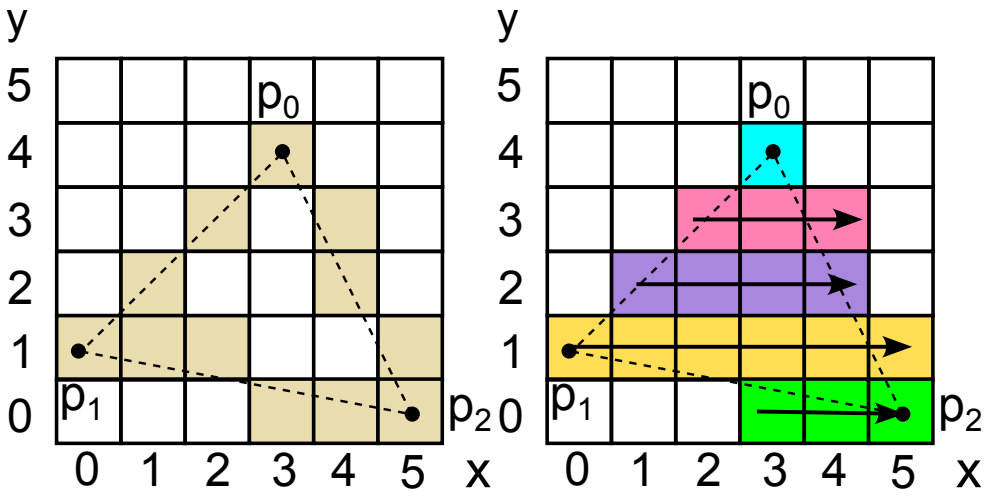
Remarque : en multipliant toutes les valeurs par $2\,dx$, on peut éliminer les divisions et les flottants pour obtenir un algorithme purement entier. L'algorithme ci-dessus traite le cas $0 \leq dy \leq dx$; les autres octants se traitent par symétrie.

Rasterization d'un triangle : algorithme scanline

La rasterization d'un triangle (p_0, p_1, p_2) se fait par l'**algorithme scanline** :

1. **Discrétiser les arêtes** : pour chaque arête du triangle ((p_0, p_1), (p_1, p_2), (p_2, p_0)), appliquer l'algorithme de Bresenham pour obtenir les pixels de contour.
2. **Stocker les bornes horizontales** : pour chaque ligne y, stocker les valeurs x_{min} et x_{max} des pixels de contour.
3. **Remplir ligne par ligne** : pour chaque ligne y, colorier tous les pixels de x_{min} à x_{max}.

Les deux étapes sont illustrées ci-dessous : discrétisation des arêtes puis remplissage.



Rasterization d'un triangle par scanline : discrétisation des arêtes (gauche) et remplissage horizontal (droite).

Exemple de table scanline pour un triangle :

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

Lors du remplissage, les **coordonnées barycentriques** de chaque fragment sont calculées pour interpoler les attributs des sommets (couleur, normale, coordonnées de texture, profondeur) de manière linéaire à l'intérieur du triangle.

2.4 Illumination et ombrage (Shading)

Modèle d'illumination de Phong

Le **modèle de Phong** est le modèle d'illumination le plus classique en rendu temps réel. Il repose sur l'observation que l'apparence d'une surface éclairée peut être décomposée en trois phénomènes physiques distincts. Le premier est la composante **ambiante**, qui représente une couleur de base uniforme, indépendante de la géométrie et de la position de la lumière : elle modélise de manière grossière la lumière indirecte qui baigne l'ensemble de la scène. Le deuxième est la composante **diffuse**, qui dépend de l'orientation locale de la surface par rapport à la lumière : une surface tournée vers la lumière est claire, une surface tournée dans l'autre sens est sombre. Le troisième est la composante **spéculaire**, qui modélise le reflet brillant de la source lumineuse visible sur les surfaces lisses, et qui dépend du point de vue de l'observateur. Ces trois contributions sont représentées sur la figure suivante.

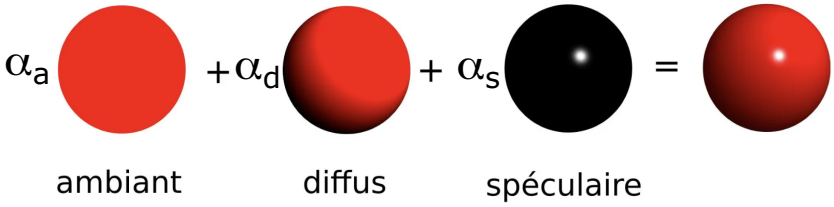


Figure 25: Les trois composantes du modèle de Phong : ambiante, diffuse et spéculaire.

La couleur finale d'un point est la somme de ces trois composantes :

$$C = C_a + C_d + C_s$$

Le calcul de chaque composante fait intervenir deux propriétés de couleur : la **couleur de la source lumineuse** $C_\ell = (r_\ell, g_\ell, b_\ell) \in [0,1]^3$ et la **couleur de la surface** (ou albedo) $C_o = (r_o, g_o, b_o) \in [0,1]^3$. Leur multiplication composante par composante ($(r_\ell \cdot r_o,\ g_\ell \cdot g_o,\ b_\ell \cdot b_o)$) modélise le filtrage de la lumière par le matériau : une surface rouge éclairée par une lumière blanche apparaît rouge, tandis que la même surface éclairée par une lumière verte apparaît noire (car la composante rouge de la lumière est nulle).

Composante ambiante

La composante ambiante modélise une illumination uniforme de la scène (lumière indirecte approximée). Elle ne dépend ni de la position ni de l'orientation de la surface :

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

avec $\alpha_a \in [0,1]$ le **coefficient ambiant**. Ce terme évite que les surfaces non éclairées directement soient complètement noires.

Composante diffuse

La composante diffuse modélise la lumière réfléchie de manière uniforme dans toutes les directions par une surface mate (réflexion lambertienne). Elle dépend de l'angle entre la **normale** n à la surface et la **direction de la lumière** u_ℓ :

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

Le coefficient $\alpha_d \in [0,1]$ contrôle l'intensité de la réflexion diffuse du matériau. Le terme central est le **facteur diffus** $(n \cdot u_\ell)_+ = \max(n \cdot u_\ell,\ 0)$, qui est le produit scalaire entre le vecteur normal unitaire n à la surface et la direction unitaire u_ℓ du point vers la source lumineuse. Ce produit scalaire mesure le cosinus de l'angle entre la normale et la direction de la lumière. Lorsque la surface fait directement face à la lumière (n parallèle à u_ℓ), le facteur vaut 1 et l'éclairage est maximal. Lorsque la surface est rasante (n perpendiculaire à u_ℓ), le facteur vaut 0. Lorsque la surface est tournée à l'opposé de la lumière, le produit scalaire est négatif, et la troncature $\max(\cdot, 0)$ force la contribution à zéro, évitant un éclairage physiquement absurde.

Ce mécanisme géométrique est schématisé ci-dessous.

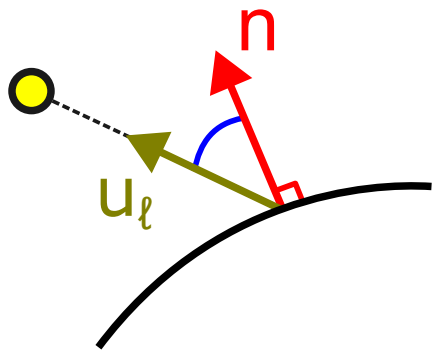


Figure 26: Composante diffuse : la luminosité dépend de l'angle entre la normale et la direction de la lumière.

Composante spéculaire

La composante spéculaire modélise le **reflet** de la source lumineuse sur la surface. Contrairement à la composante diffuse, elle dépend du **point de vue** de la caméra :

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

Le coefficient $\alpha_s \in [0, 1]$ contrôle l'intensité du reflet. L'exposant de brillance s_{exp} (typiquement entre 64 et 256) détermine la **taille** du reflet : plus il est élevé, plus le reflet est concentré en un point étroit et net (surface très polie), tandis qu'un exposant faible produit un reflet large et diffus (surface légèrement rugueuse).

Le vecteur u_r est la **direction de réflexion** de la lumière par rapport à la normale, calculée par la formule $u_r = 2\left(u_\ell \cdot n\right) n - u_\ell$ (en GLSL : `reflect(-u_l, n)`). Le vecteur u_v est la direction unitaire du point de la surface vers la caméra. Le produit scalaire $\left(u_r \cdot u_v\right)$ mesure l'alignement entre la direction de réflexion et la direction de vue : le reflet est maximal lorsque l'observateur se trouve exactement dans la direction de réflexion miroir de la lumière.

La composante spéculaire ne dépend **pas** de la couleur de la surface C_o mais uniquement de C_ℓ, ce qui correspond au comportement physique des reflets sur les matériaux diélectriques (plastique, céramique, etc.) : le reflet d'une lumière blanche sur une surface rouge reste blanc. La géométrie de la réflexion spéculaire et l'influence de l'exposant de brillance sont détaillées dans les figures suivantes.

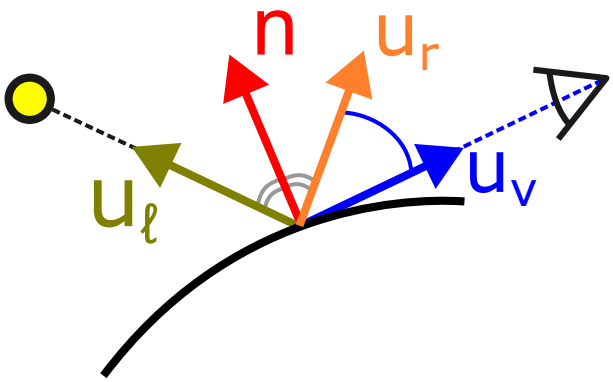


Figure 27: Composante spéculaire : le reflet dépend de la direction de réflexion et du point de vue.

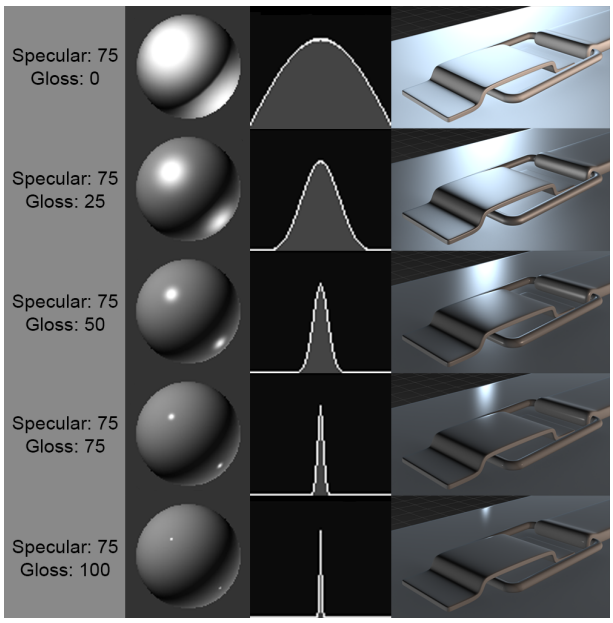


Figure 28: Effet de l'exposant de brillance : plus s_{exp} est élevé, plus le reflet est concentré.

Interpolation de l'illumination : Flat, Gouraud, Phong

Le modèle de Phong définit la formule d'illumination en un point de la surface, mais reste la question de **où** et **à quelle fréquence** cette formule est évaluée. Trois stratégies classiques existent, avec un compromis entre coût de calcul et qualité visuelle.

Le **flat shading** est l'approche la plus simple : la couleur est calculée **une seule fois par triangle**, en utilisant la normale géométrique de la face. Tous les pixels du triangle reçoivent exactement la même couleur. Le résultat a un aspect facetté caractéristique où chaque triangle est clairement visible, ce qui peut être acceptable pour du rendu non photo-réaliste mais est inadapté pour des surfaces lisses.

Le **Gouraud shading** améliore la qualité en calculant l'illumination **aux sommets** du triangle (dans le vertex shader), puis en **interpolant linéairement** la couleur résultante à l'intérieur du triangle lors de la rasterization. Les transitions de couleur entre triangles adjacents deviennent douces, donnant une apparence de surface lisse. Cependant, cette approche présente un défaut important : si un reflet spéculaire tombe au milieu d'un grand triangle, loin de tout sommet, il sera complètement manqué car aucun sommet n'a ”vu” le reflet.

Le **Phong shading** résout ce problème en interpolant non pas la couleur, mais la **normale** entre les sommets du triangle. L'illumination est ensuite calculée **pour chaque fragment** (dans le fragment shader) avec cette normale interpolée. Le reflet spéculaire est ainsi correctement calculé en chaque point de la surface, même au centre d'un grand triangle. La différence visuelle entre ces trois approches est comparée sur la figure suivante.

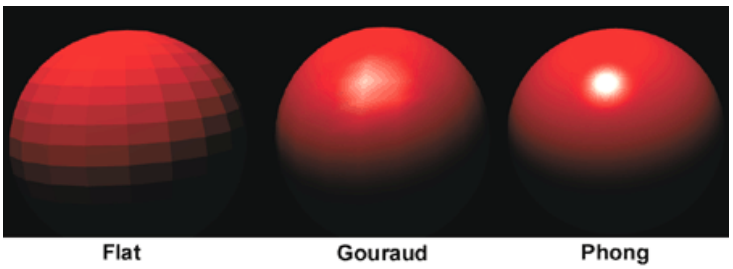


Figure 29: Comparaison des trois modèles d'ombrage : Flat, Gouraud et Phong.

En pratique, le **Phong shading** est le standard. Le surcoût par rapport au Gouraud shading est négligeable sur les GPU modernes (le fragment shader effectue quelques opérations supplémentaires par pixel, mais le GPU dispose de milliers de cœurs pour les exécuter en parallèle), et la qualité visuelle est nettement supérieure.

Lumières multiples et effets

Lumières multiples

Le modèle de Phong s'étend naturellement à plusieurs sources lumineuses en sommant les contributions de chaque lumière :

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

Atténuation par la distance

Dans la réalité physique, l'intensité lumineuse d'une source ponctuelle diminue proportionnellement à l'inverse du carré de la distance ($1/r^2$). En rendu temps réel, on utilise souvent un modèle d'atténuation simplifié qui décroît linéairement jusqu'à une distance maximale, au-delà de laquelle la lumière n'a plus d'effet :

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

Le vecteur p est la position du point sur la surface, p_ℓ la position de la lumière, d_{att} la distance caractéristique d'atténuation (au-delà de laquelle la contribution est nulle), et C_ℓ^0 la couleur de la lumière à la source. Ce modèle linéaire est moins réaliste que la loi en $1/r^2$, mais il a l'avantage de garantir que la lumière s'éteint complètement au-delà de d_{att}, ce qui permet d'optimiser le rendu en ignorant les lumières trop éloignées.

Effet de brouillard (Fog)

Un effet de brouillard simule l'absorption et la diffusion de la lumière par l'atmosphère. Le principe est de mélanger progressivement la couleur calculée avec une couleur de brouillard uniforme en fonction de la distance à la caméra :

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

Le facteur de mélange $\gamma(p) = \min\left(\frac{\|p - p_{\text{eye}}\|}{d_{\text{fog}}},\, 1\right)$ varie de 0 (objet proche, couleur inchangée) à 1 (objet lointain, complètement noyé dans le brouillard). Le paramètre d_{fog} contrôle la distance à laquelle le brouillard devient totalement opaque, et C_{fog} est la couleur du brouillard (souvent un gris clair ou la couleur du ciel). En pratique, cet effet donne une impression de profondeur atmosphérique et masque naturellement le plan de clipping lointain (z_{far}), évitant la disparition brutale des objets à l'horizon.

Exemple d'effet : Toon Shading

Le **toon shading** (ou cel shading) est un effet non photo-réaliste qui donne un aspect dessin animé. Le principe est de **discrétiser** les valeurs de couleur en paliers :

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

où N est le nombre de paliers souhaités. Ce sous-échantillonnage crée des bandes de couleur discrètes caractéristiques du style cartoon, dont on voit un exemple ci-dessous.

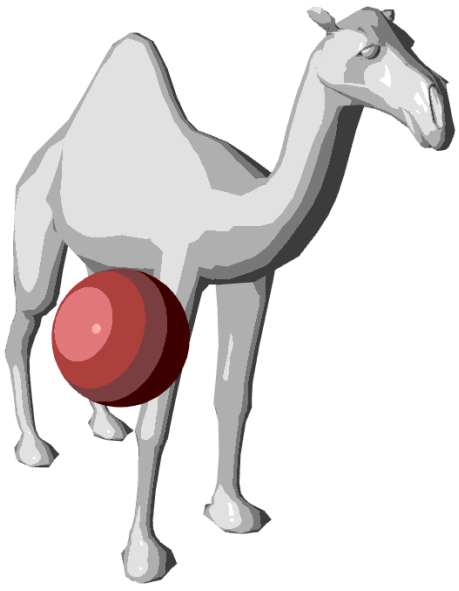


Figure 30: Toon shading : discrétisation des couleurs pour un rendu stylisé.

2.5 Buffer de profondeur (Depth Buffer)

Problème de l'occultation

Lorsque plusieurs triangles se chevauchent à l'écran, il faut déterminer lequel est visible pour chaque pixel. Sans mécanisme d'occultation, l'ordre d'affichage des triangles détermine le résultat, ce qui est incorrect, comme le met en évidence l'image suivante.

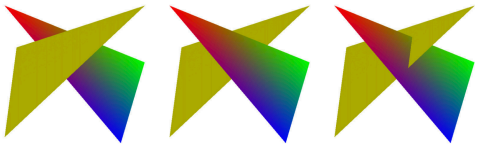


Figure 31: L'ordre de dessin des triangles affecte le résultat sans mécanisme de profondeur.

Algorithme du Z-buffer

Le **Z-buffer** (ou depth buffer) est une image auxiliaire de la même résolution que l'image finale, qui stocke la valeur de profondeur z_{ndc} du fragment le plus proche de la caméra pour chaque pixel.

L'algorithme est simple : pour chaque fragment à dessiner, on compare sa profondeur avec la valeur stockée dans le Z-buffer :

```
def draw(position, couleur, z, image, zbuffer):
    if z < zbuffer(position):
        image(position) = couleur
        zbuffer(position) = z
    # sinon: ne rien dessiner (fragment masqué)
```

On peut visualiser le contenu du Z-buffer sous forme d'image en niveaux de gris.

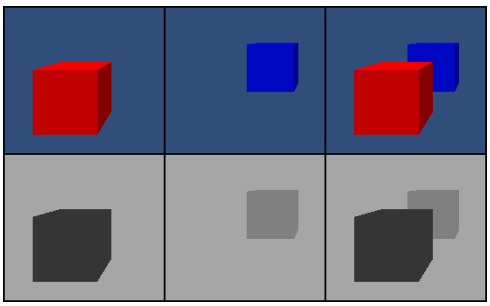


Figure 32: Visualisation du depth buffer : les zones claires sont proches de la caméra, les zones sombres sont éloignées.

Le Z-buffer est initialisé à la valeur maximale de profondeur (1.0, correspondant au plan lointain) à chaque début de frame. Chaque fragment met à jour le Z-buffer si sa profondeur est inférieure (plus proche de la caméra) à la valeur stockée. Cela garantit que seul le fragment le plus proche est affiché, **indépendamment de l'ordre de dessin** des triangles.

Cette étape est réalisée automatiquement par le GPU (non programmable), à condition d'activer le test de profondeur (`glEnable(GL_DEPTH_TEST)` en OpenGL).

2.6 Shaders : synthèse du pipeline

Pipeline complet

Le pipeline de rendu complet enchaîne plusieurs étapes en séquence. Le **vertex shader** transforme d'abord chaque sommet en appliquant la chaîne de matrices Projection $\times$ View $\times$ Model, et transmet les attributs transformés (position en espace monde, normale, couleur) vers les étapes suivantes. Les sommets transformés sont ensuite regroupés en triangles lors de l'**assemblage des primitives**, en utilisant la table de connectivité envoyée depuis le CPU.

La **rasterization** prend chaque triangle projeté en 2D et détermine l'ensemble des pixels qu'il recouvre. Pour chaque pixel couvert (appelé fragment), les attributs des trois sommets du triangle sont interpolés de manière barycentrique.

Le **fragment shader** reçoit ces attributs interpolés et calcule la couleur finale du fragment en appliquant le modèle d'illumination de Phong. Le **test de profondeur** (Z-buffer) compare ensuite la profondeur du fragment avec la valeur stockée pour ce pixel : seul le fragment le plus proche de la caméra est conservé, les autres sont écartés. Le résultat de l'ensemble de ces étapes est l'**image finale**, stockée dans le framebuffer et affichée à l'écran.

L'enchaînement complet de ces étapes est résumé dans le schéma ci-dessous.

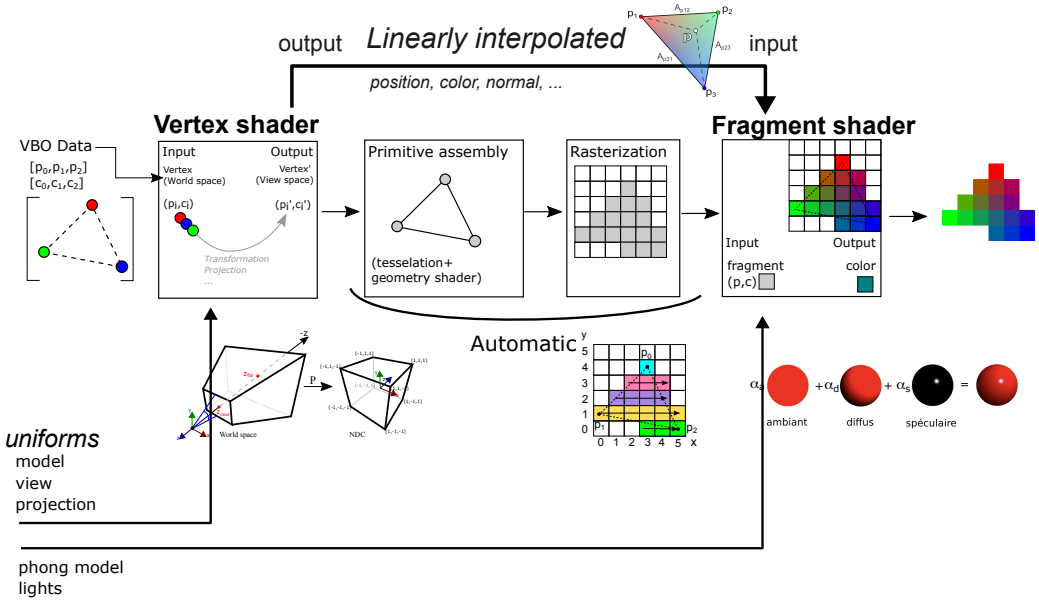


Figure 33: Pipeline complet de rendu : du sommet à l'image finale.

Exemple de code complet

Vertex shader : transforme les positions et normales, transmet les attributs au 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;
}
```

Remarque : la normale est transformée par la **transposée de l'inverse** de la matrice Model (`transpose(inverse(model))`),

et non par Model directement. En effet, si Model contient un scaling non uniforme, la multiplication directe déformerait les normales et l'ombrage serait incorrect.

Fragment shader : calcule l'illumination de Phong pour chaque 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;                    // ambiante
    c = c + 0.7 * diffuse_magnitude * fragment.color * light_color;  // diffuse
    // + composante spéculaire (omise pour simplifier)

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

Les variables `fragment.position`, `fragment.normal` et `fragment.color` sont automatiquement **interpolées de manière barycentrique** par le GPU entre les valeurs des trois sommets du triangle courant.

3 Maillages et Textures

3.1 Géométrie des maillages

Structure d'un maillage

Un **maillage** (mesh) est un ensemble de polygones partageant des sommets. Il est décrit par :

- N_f **faces** (polygones).
- N_v **sommets** (vertices).
- N_e **arêtes** (edges).

Selon le type de polygones utilisés, on distingue :

- **Triangulation** : toutes les faces sont des triangles. C'est le format standard pour le rendu GPU.
- **Quad mesh** : toutes les faces sont des quadrangles (idéalement planaires). Utilisé en modélisation et en subdivision.
- **Poly mesh** : les faces sont des polygones quelconques (nombre de côtés variable).

Ces trois catégories sont illustrées ci-dessous.

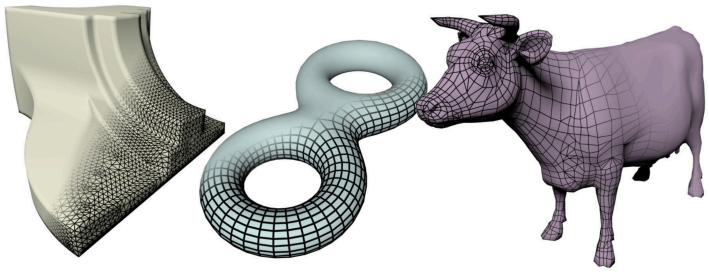


Figure 34: Types de maillages : triangulation, quad mesh et poly mesh.

Un maillage représente une **variété** (manifold) si chaque arête est partagée par **au plus 2 faces**, comme illustré sur la figure suivante. Cette propriété est fondamentale : elle garantit que la surface est localement équivalente à un plan (ou un demi-plan au bord), ce qui est nécessaire pour la plupart des algorithmes de traitement de maillage.

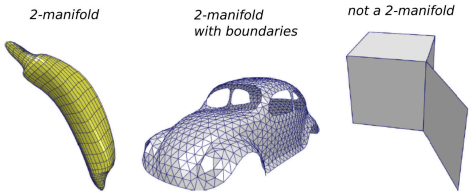


Figure 35: Propriété de variété : chaque arête est partagée par au plus 2 faces.

Structure de données

L'encodage standard d'un maillage triangulaire en C++ utilise deux tableaux :

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

std::vector<vec3> position;      // positions des sommets
std::vector<int3> connectivity;  // indices des faces (triplets)
```

L'utilisation de `std::vector` garantit la **contiguïté mémoire**, essentielle pour le transfert efficace vers le GPU.

Exemple : tétraèdre

```
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} };
```

Exemple : maillage plan

```
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} };
```

Le schéma suivant montre la correspondance entre les indices et la géométrie du maillage.

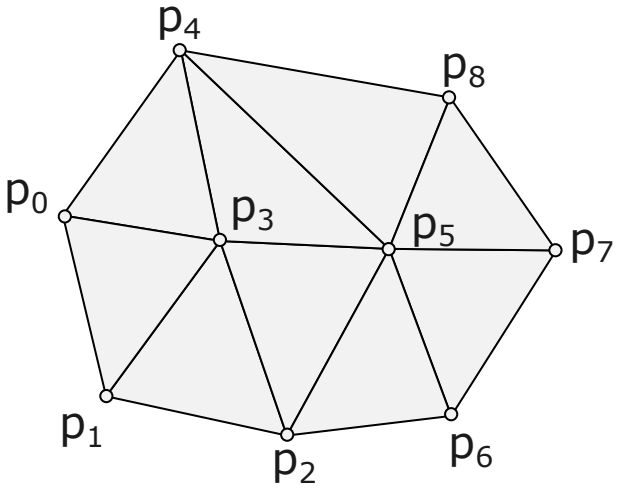


Figure 36: Structure d'un maillage triangulaire : sommets et connectivité.

Remarque sur l'orientation : les triplets $(0, 1, 3)$, $(1, 3, 0)$ et $(3, 0, 1)$ sont équivalents (permutation cyclique). En revanche, $(0, 1, 3)$ a une orientation **inverse** à $(3, 1, 0)$, $(1, 0, 3)$ ou $(0, 3, 1)$. L'orientation détermine la direction de la normale et donc le sens de la face (voir chapitre précédent).

Flux de données CPU/GPU

Le flux de travail typique pour afficher un maillage avec OpenGL suit les étapes suivantes :

1. **Définir** une variable `mesh_drawable` (partagée entre les méthodes).
2. **Initialiser** une structure `mesh` contenant les tableaux de données en RAM (CPU).
3. **Transférer** les données du `mesh` vers le `mesh_drawable` en VRAM (GPU).
4. **À chaque image** :
    - Mettre à jour les paramètres uniformes (matrices, lumières, etc.).
    - Appeler `draw()` qui active le shader, envoie les uniformes et lance le rendu.

Ce flux est résumé dans le diagramme ci-dessous.

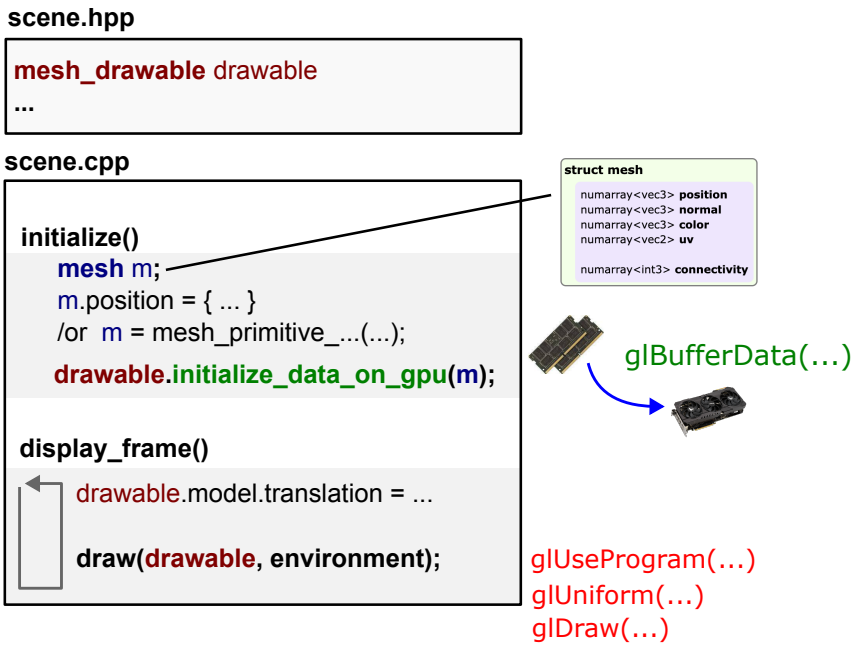


Figure 37: Flux de données du CPU au GPU pour le rendu d'un maillage.

Attributs de sommets

En pratique, chaque sommet porte des **attributs supplémentaires** en plus de sa position : normales, coordonnées de texture, couleurs, 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} };
```

avec $p_i = (x_i, y_i, z_i)$, $n_i = (n_{x_i}, n_{y_i}, n_{z_i})$ avec $\|n_i\| = 1$, et $c_i = (r_i, g_i, b_i)$. La figure ci-après représente ces différents attributs sur un maillage.

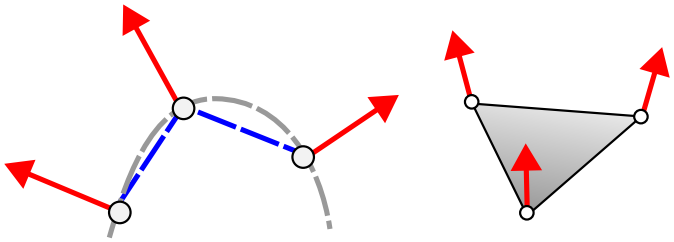


Figure 38: Attributs de sommets : position, normale, couleur et coordonnées de texture.

Points importants :

- Les **normales sont définies par sommet** (et non par face), ce qui permet une apparence lisse grâce à l'interpolation de l'illumination dans le fragment shader.
- Une **unique table de connectivité** est partagée par tous les attributs pour le rendu GPU. Cela signifie que les indices de la table de faces s'appliquent simultanément aux positions, normales, couleurs et coordonnées de texture.

Duplication de sommets

Parfois, il est nécessaire de **dupliquer** un sommet lorsqu'il occupe la même position mais avec des **attributs différents** selon la face considérée. Le cas le plus courant est celui des **arêtes vives** (sharp edges).

Considérons un sommet situé à une position p_A où deux groupes de faces se rencontrent à angle droit. Si l'on souhaite un rendu avec une arête nette (non lissée), il faut que ce sommet porte deux normales différentes, une pour chaque groupe de faces. Comme la table de connectivité est unique, on doit créer **deux entrées distinctes** dans le tableau de sommets, partageant la même position mais avec des normales différentes.

```
// Avant duplication : sommet v4 à la position pA avec une seule normale
std::vector<vec3> position = { p0, p1, p2, p3, pA, pB };
std::vector<vec3> normal   = { n0, n0, n1, n1, n2, n2 };

// Après duplication : v4 et v6 partagent pA mais avec des normales différentes
std::vector<vec3> position = { p0, p1, p2, p3, pA, pB, pA, pB };
std::vector<vec3> normal   = { n0, n0, n1, n1, n1, n1, n0, n0 };
```

Le principe est visible sur le schéma suivant.

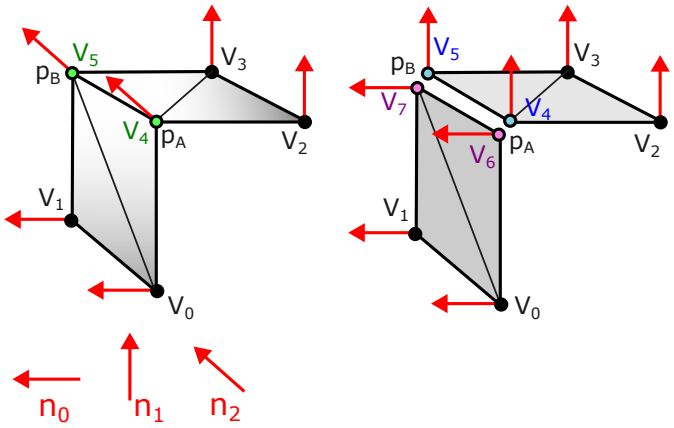


Figure 39: Duplication de sommets pour les arêtes vives : même position, normales différentes.

Pour un **cube**, chaque sommet géométrique est partagé par 3 faces perpendiculaires. Comme chaque face nécessite une normale différente, chaque sommet est dupliqué **3 fois**, soit $8 \times 3 = 24$ sommets au lieu de 8.

Surfaces paramétriques et grilles

Un cas fréquent de maillage est la **discrétisation de surfaces paramétriques**.

Soit une surface $S(u, v) = (S_x(u, v),\ S_y(u, v),\ S_z(u, v))$ avec $(u, v) \in [0, 1]^2$, échantillonnée uniformément sur une grille $N_u \times N_v$, dont la structure est représentée ci-dessous.

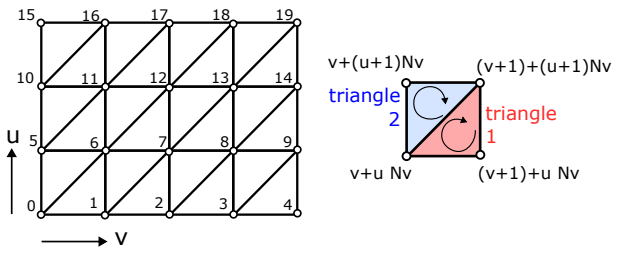


Figure 40: Structure de grille pour une surface paramétrique.

La construction du maillage se fait en deux étapes :

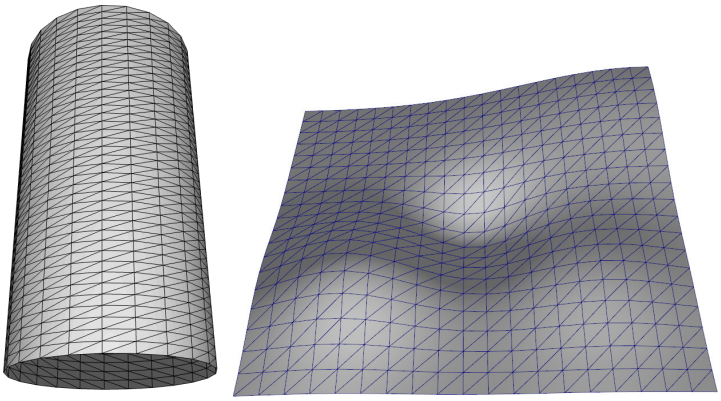
Positions des sommets :

```
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) };
    }
}
```

Connectivité (deux triangles par cellule de la grille) :

```
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);
    }
}
```

Ce schéma s'applique à de nombreuses surfaces, comme le montrent les exemples suivants.



Exemples de surfaces paramétriques : cylindre (gauche) et terrain (droite).

Calcul des normales

Les normales ne sont pas toujours fournies (fichiers sans normales, maillages procéduraux, déformations en temps réel). Il est alors nécessaire de les **calculer à partir de la géométrie**.

La méthode standard consiste à calculer la **moyenne non pondérée des normales des triangles voisins** de chaque sommet :

1. Pour chaque sommet i, identifier les triangles voisins $t \in \mathcal{N}_i$ (appelés le **1-voisinage** ou **1-ring** du sommet).
2. Calculer la normale de chaque triangle voisin : $n_t = (p_b - p_a) \times (p_c - p_a)$.
3. Sommer et normaliser :

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

Ce procédé est illustré ci-dessous.

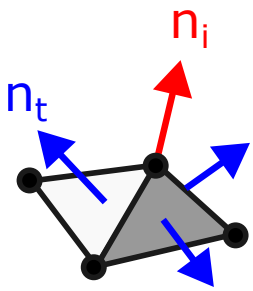


Figure 41: Calcul de la normale d'un sommet par moyenne des normales des triangles voisins.

Implémentation :

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

// Accumulation par 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;
}

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

Si des sommets sont dupliqués (arêtes vives), les normales calculées seront naturellement différentes pour chaque copie, car elles ne partagent pas les mêmes triangles voisins, comme on peut le voir sur la figure suivante.

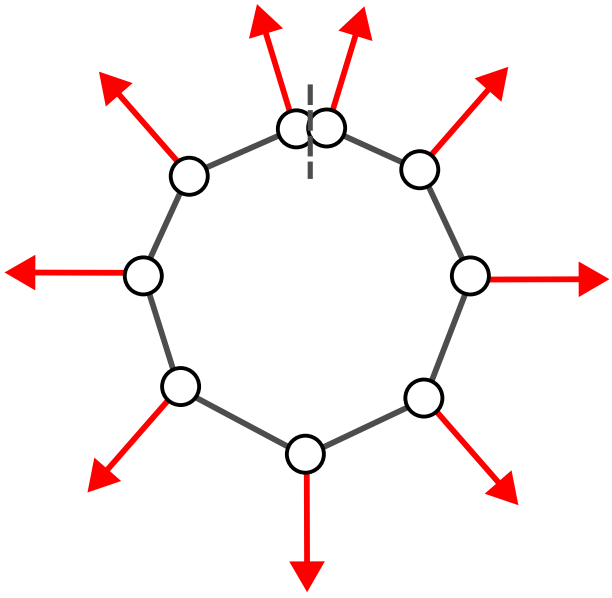


Figure 42: Effet de la duplication de sommets sur les normales calculées.

1-voisinage (1-ring)

Le **1-ring** d'un sommet est l'ensemble des triangles qui partagent ce sommet. Cette structure de voisinage est utile pour de nombreux algorithmes (calcul de normales, lissage, 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);
    }
}
```

Par exemple :

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

La figure ci-dessous visualise cette notion de voisinage.

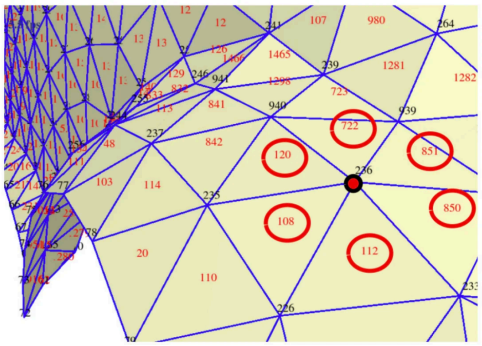


Figure 43: Visualisation du 1-ring : l'ensemble des triangles voisins d'un sommet.

Structure en demi-arête (half-edge)

La structure en **demi-arête** (half-edge) est une structure de données plus riche que la simple table de connectivité. Elle encode les arêtes de manière orientée : chaque arête est représentée par deux demi-arêtes de directions opposées, comme le montre le schéma ci-après. Les faces apparaissent comme des boucles le long des demi-arêtes.

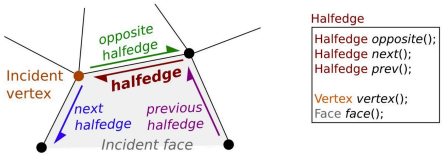


Figure 44: Structure en demi-arête : chaque arête orientée pointe vers son opposée.

Avantages :

- Ajout et suppression d'éléments en $O(1)$ (opérations de split, collapse, flip).
- Calcul efficace des angles aux coins, représentés sur la figure suivante (pondération par cotangente pour le laplacien discret).

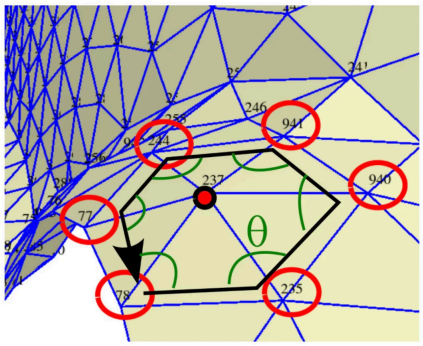


Figure 45: Angles aux coins des triangles, utiles pour les pondérations cotangentes.

Limitations :

- Restreinte aux **2-variétés orientables** (chaque arête partagée par au plus 2 faces, pas de configuration non-manifold).
- **Non contiguë en mémoire** (structure à base de pointeurs), ce qui peut pénaliser les performances.

Exemple avec 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

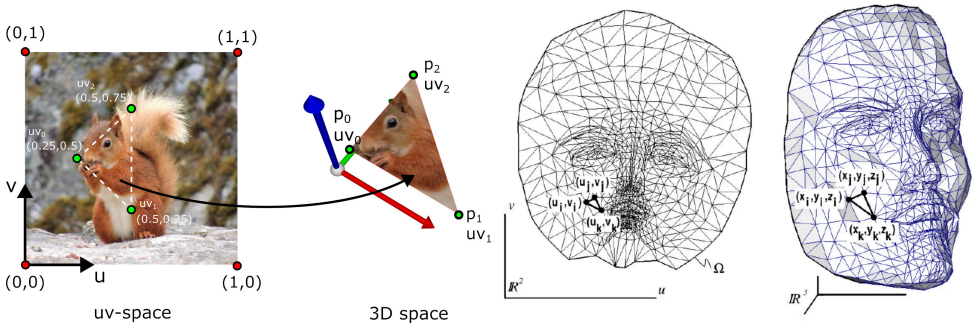
Principe de l'UV-mapping

L'objectif des textures est de fournir un **niveau de détail en couleur plus fin que la géométrie**. Un triangle peut avoir une couleur uniforme ou interpolée entre ses sommets, mais cela reste limité. Les textures permettent de plaquer une image 2D détaillée sur la surface 3D.

Le cas classique est l'**UV-mapping** : une image 2D (la **texture**) est associée à la surface 3D via des **coordonnées de texture** (u, v) (aussi notées (s, t)).

- À chaque sommet v_i du maillage, on associe une coordonnée 2D $uv_i = (u_i, v_i) \in [0, 1]^2$ dans l'espace de la texture.
- Au moment du rendu, les coordonnées de texture sont interpolées de manière barycentrique entre les sommets du triangle, et la couleur de la texture est lue à la position interpolée.

Le mécanisme est illustré ci-dessous.



Principe de l'UV-mapping : association entre coordonnées 2D de la texture et surface 3D.

La convention est que $(0, 0)$ correspond au coin inférieur gauche de l'image et $(1, 1)$ au coin supérieur droit.

Dépliage UV

Pour des modèles complexes, le dépliage UV se fait par **morceaux** (patches ou îlots). Le processus comprend les étapes suivantes :

1. **Découpage en patches** : la surface est découpée le long de **coutures** (seams) choisies par l'artiste ou un algorithme automatique. Les coutures sont placées dans des zones peu visibles (plis, arrière de l'objet).
2. **Dépliage** : chaque patch est déplié dans le plan 2D en minimisant les déformations. Des algorithmes classiques incluent ABF (Angle-Based Flattening), LSCM (Least Squares Conformal Maps) et Slim (Scalable Locally Injective Mappings).
3. **Packing** : les patches dépliés sont disposés dans l'espace $[0, 1]^2$ en maximisant l'occupation de l'espace (pour éviter de gaspiller la résolution de texture).
4. **Peinture** : l'artiste peint la texture directement dans l'espace 2D déplié, ou utilise des outils de peinture 3D (comme Substance Painter) qui projettent directement sur la surface.

Les coutures entre les patches correspondent aux endroits où la texture peut présenter des **discontinuités** (le sommet à la couture est dupliqué avec des coordonnées UV différentes de chaque côté). Un exemple de dépliage est visible ci-dessous.

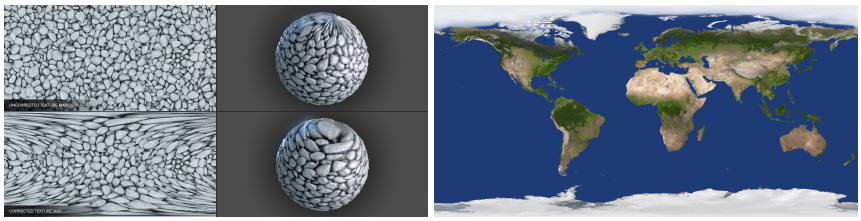
Figure 46: Dépliage UV en pratique : la surface est découpée en patches disposés dans l'espace texture.

Déformations et limites

L'UV-mapping consiste à ”déplier” la surface 3D sur un plan 2D. Ce dépliage introduit nécessairement des **déformations** en longueur et en angle pour toute surface dont la **courbure de Gauss** est non nulle.

En particulier, il est **impossible** de plaquer une image plane sur une sphère sans déformation. C'est un résultat fondamental de géométrie différentielle : la courbure de Gauss est un **invariant intrinsèque** de la surface (theorema egregium de Gauss). Une sphère a une courbure de Gauss positive constante ($K = 1/R^2$), tandis qu'un plan a une courbure nulle ($K = 0$). Aucune déformation sans étirement ne peut transformer l'une en l'autre.

Seules les surfaces **développables** (courbure de Gauss nulle, comme les cylindres et les cônes) peuvent être dépliées sans distorsion. En pratique, on cherche à **minimiser** les déformations lors du dépliage UV, en acceptant un compromis entre la distorsion des angles (conforme) et la distorsion des aires (authalique). La figure suivante montre un cas typique de déformation sur une sphère.



Déformation lors du placage d'une texture sur une sphère : étirement aux pôles.

Types de textures

Au-delà de la couleur (diffuse map), les textures sont utilisées pour stocker de nombreux types d'informations sur la surface :

- **Diffuse map** (albedo) : couleur de base de la surface.
- **Normal map** : normales perturbées pour simuler du relief (voir section suivante).
- **Specular map** : coefficient spéculaire variable sur la surface (zones brillantes vs mates).
- **Roughness map** : rugosité de la surface (rendu physiquement réaliste, PBR).
- **Ambient occlusion map** : pré-calcul de l'occultation ambiante (ombrage dans les creux).
- **Height map** (displacement map) : hauteur pour le parallax mapping ou la déformation réelle de la géométrie.
- **Emissive map** : zones émettant de la lumière (écrans, néons, etc.).

Textures en OpenGL

Mise en place

Les coordonnées de texture sont stockées comme attribut de sommet, au même titre que les positions et les normales :

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

Côté C++/OpenGL, la texture est chargée et envoyée au GPU :

```
// Chargement de l'image
int width, height, channels;
unsigned char* data = stbi_load("texture.png", &width, &height, &channels, 4);

// Création de la texture OpenGL
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 dans le fragment shader (unité de texture 0)
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture_id);
glUniform1i(glGetUniformLocation(shader, "texture_image"), 0);
```

Interpolation et échantillonnage

Lors du rendu, le fragment shader reçoit les coordonnées de texture (u, v) interpolées de manière barycentrique entre les trois sommets du triangle courant. Il lit ensuite la couleur dans l'image de texture à cette position grâce à la fonction `texture()` :

```
uniform sampler2D texture_image;  // la texture, envoyée depuis le CPU

in vec2 fragment_uv;              // coordonnées interpolées
out vec4 FragColor;

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

La variable `sampler2D` représente une référence vers une texture stockée en VRAM. La fonction `texture(sampler, uv)` réalise l'**échantillonnage** : elle convertit les coordonnées normalisées $(u, v) \in [0, 1]^2$ en coordonnées pixel dans l'image, puis renvoie la couleur correspondante.

Filtrage de texture

Les coordonnées (u, v) interpolées tombent rarement exactement sur le centre d'un pixel de la texture (appelé **texel**). Le GPU doit donc **interpoler** entre les texels voisins. Ce processus est appelé **filtrage de texture**. Il intervient dans deux cas :

- **Magnification** (agrandissement) : le triangle est plus grand que la texture à l'écran, plusieurs pixels couvrent un seul texel. Sans filtrage, la texture apparaît pixelisée.
- **Minification** (réduction) : le triangle est plus petit que la texture à l'écran, un seul pixel couvre de nombreux texels. Sans filtrage, la texture scintille (aliasing).

Les modes de filtrage les plus courants sont :

- **Nearest** (`GL_NEAREST`) : sélectionne le texel le plus proche. Rendu pixelisé mais rapide. Utile pour un style pixel-art.
- **Bilinéaire** (`GL_LINEAR`) : interpole linéairement entre les 4 texels les plus proches. Rendu lisse, standard pour la magnification.
- **Trilinéaire** (`GL_LINEAR_MIPMAP_LINEAR`) : interpolation bilinéaire + interpolation entre deux niveaux de **mipmap** (voir ci-dessous). Standard pour la minification.

Configuration en OpenGL :

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

Mipmaps

Les **mipmaps** sont des versions pré-calculées de la texture à résolutions décroissantes (chaque niveau divise la résolution par 2). Lors de la minification, le GPU choisit le niveau de mipmap dont la résolution est la plus adaptée à la taille du triangle à l'écran, puis interpole si nécessaire.

Pour une texture de 1024×1024, la chaîne de mipmaps contient les niveaux : 1024^2, 512^2, 256^2, ..., 2^2, 1^2 (soit 11 niveaux). L'espace mémoire total est seulement $\frac{4}{3}$ de la texture originale.

Les mipmaps réduisent l'**aliasing** (scintillement des textures vues de loin) et améliorent les performances (les textures lointaines sont lues dans des niveaux de résolution faible, mieux exploités par le cache mémoire du GPU).

Génération automatique en OpenGL :

```
glGenerateMipmap(GL_TEXTURE_2D);
```

Modes de répétition (Wrapping)

Lorsque les coordonnées de texture sortent de l'intervalle $[0, 1]$, le comportement dépend du **mode de wrapping** :

- **Repeat** (`GL_REPEAT`) : la texture se répète indéfiniment. La coordonnée $u = 2.3$ est interprétée comme $u = 0.3$. Utile pour les textures carrelables (briques, herbe, etc.).
- **Mirrored Repeat** (`GL_MIRRORED_REPEAT`) : la texture se répète en alternant son orientation, évitant les coutures visibles aux bords.
- **Clamp to Edge** (`GL_CLAMP_TO_EDGE`) : les coordonnées hors $[0, 1]$ sont ramenées au bord de la texture. Les pixels hors limites prennent la couleur du bord le plus proche.

Configuration en OpenGL :

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

Multiples textures dans le fragment shader

Toutes les textures (couleur, normales, spéculaire, etc.) partagent les mêmes coordonnées UV et utilisent le même mécanisme d'échantillonnage. Elles sont simplement lues dans des `sampler2D` différents dans le 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;
    // ...
}
```

Effets avancés de textures

Skybox

Une **skybox** est une boîte texturée représentant l'environnement distant (ciel, paysage). Elle est toujours centrée sur la position de la caméra, ce qui donne l'**impression d'un paysage infini**, comme on le voit ci-dessous.

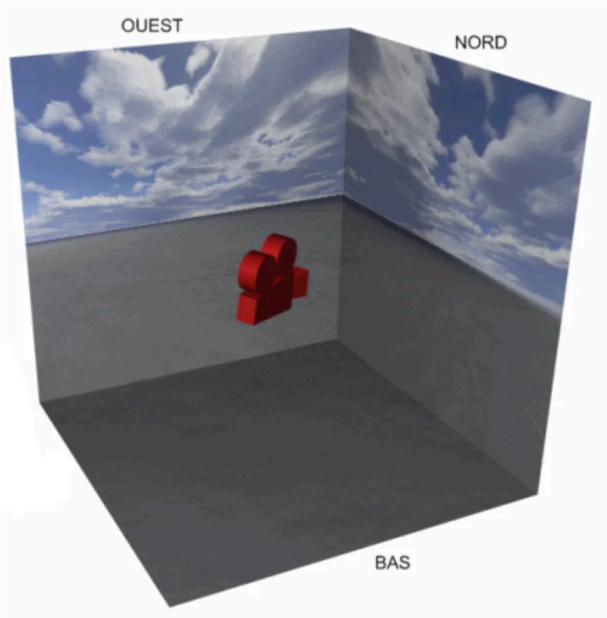


Figure 47: Skybox : boîte texturée simulant un environnement infini.

Le principe d'implémentation est le suivant :

1. Afficher la skybox **en premier**, en désactivant l'écriture dans le depth buffer.
2. Afficher ensuite tous les autres objets de la scène (qui apparaissent toujours devant la skybox).

```
void display() {
    glDepthMask(GL_FALSE);   // désactiver l'écriture dans le depth buffer
    draw(skybox, environment);

    glDepthMask(GL_TRUE);    // réactiver l'écriture
    // dessiner les autres objets ...
}
```

Environment mapping

L'**environment mapping** simule la réflexion de l'environnement (skybox) sur une surface brillante. Cela donne l'impression que l'objet reflète le monde autour de lui.

Le principe, schématisé ci-dessous, est de calculer, pour chaque fragment, la direction de réflexion du vecteur vue par rapport à la normale, puis de lire la couleur correspondante dans la texture de la skybox.

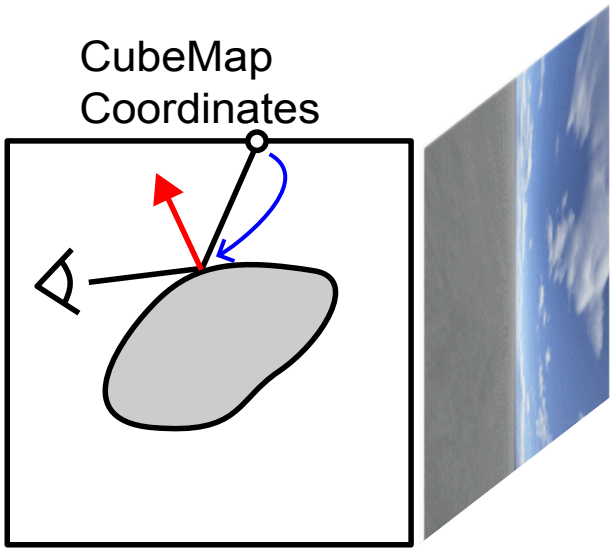


Figure 48: Environment mapping : réflexion de la skybox sur une surface.

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

Normal mapping

Le **normal mapping** consiste à modifier les normales de la surface à l'aide d'une image (la **normal map**) pour simuler des détails géométriques plus fins que les triangles du maillage. La géométrie réelle du triangle ne change pas : seule la normale utilisée dans le calcul d'illumination est modifiée, ce qui donne l'illusion de relief visible sur la comparaison suivante.

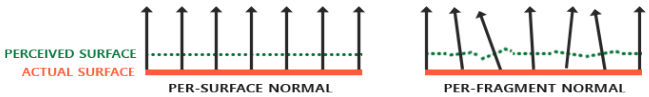


Figure 49: Normal mapping : la géométrie reste plane (gauche) mais l'illumination simule du relief (droite).

Espace tangent

Les normales d'une normal map ne sont pas stockées en coordonnées monde (qui dépendraient de l'orientation de l'objet), mais dans un repère local au triangle appelé **espace tangent** (T, B, N) :

- T (**tangente**) : direction alignée avec l'axe u de la texture sur la surface.
- B (**binormale** ou bitangente) : direction alignée avec l'axe v de la texture.
- N (**normale**) : normale géométrique de la surface.

Ce repère est défini par les coordonnées de texture (u, v) et les positions des sommets du triangle :

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

Ce système de deux équations vectorielles (soit 6 équations scalaires) permet de résoudre T et B (6 inconnues). En notation matricielle, pour les composantes x par exemple :

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

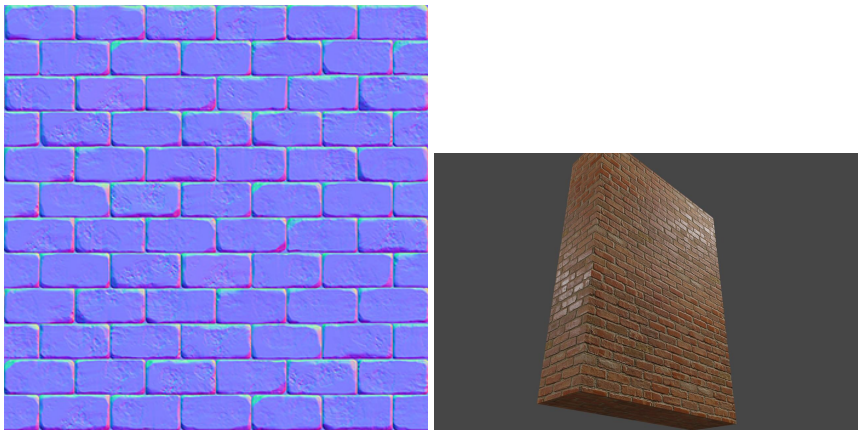
et de même pour les composantes y et z. Les vecteurs T et B sont ensuite normalisés.

Encodage de la normal map

Chaque pixel de la normal map stocke une normale perturbée en composantes $(r, g, b) \in [0, 1]^3$, encodant les composantes dans l'espace tangent sur $[-1, 1]$:

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

La composante b (bleue) correspond à la direction de la normale géométrique N. C'est pourquoi les normal maps apparaissent **bleutées** : la plupart des normales sont proches de $(0, 0, 1)$ dans l'espace tangent, ce qui donne $(0.5, 0.5, 1.0)$ en RGB. On peut observer cet encodage sur les images ci-dessous.



Normal map : les normales sont encodées en couleur dans l'espace tangent (gauche). Application sur un mur en briques (droite).

La normale finale en coordonnées monde est obtenue par changement de base :

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

où $\text{TBN} = (T \mid B \mid N)$ est la matrice 3×3 dont les colonnes sont les vecteurs du repère tangent.

Implémentation GLSL

Vertex shader : calcul du repère tangent et passage au 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 : lecture de la normal map et calcul de l'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() {
    // Lire la normale dans la normal map et la convertir de [0,1] à [-1,1]
    vec3 n_tangent = texture(normal_map, fragment.uv).rgb * 2.0 - 1.0;

    // Transformer vers l'espace monde
    vec3 N = normalize(fragment.TBN * n_tangent);

    // Illumination avec la normale perturbée
    vec3 L = normalize(light_position - fragment.position);
    float diffuse = max(dot(N, L), 0.0);

    // ...
}
```

L'avantage de l'espace tangent est que la même normal map peut être réutilisée sur différents objets ou différentes parties d'un même objet, indépendamment de leur orientation dans le monde.

Parallax mapping

Le **parallax mapping** va plus loin que le normal mapping en déformant les **coordonnées de texture** elles-mêmes en fonction d'une **carte de hauteur** (height map). Alors que le normal mapping ne modifie que l'illumination (la silhouette reste plate), le parallax mapping crée une impression de **décalage géométrique** : les zones hautes semblent avancer et les zones basses reculer, en particulier lorsque l'on observe la surface sous un angle rasant.

Principe géométrique

Considérons un fragment sur une surface plane. Sans parallax mapping, on lirait la texture aux coordonnées (u, v) de ce fragment. Mais si la surface avait réellement du relief, le rayon de vue aurait intersecté la surface à un point différent, décalé horizontalement. Le parallax mapping approxime ce décalage, comme le montre le schéma ci-dessous.

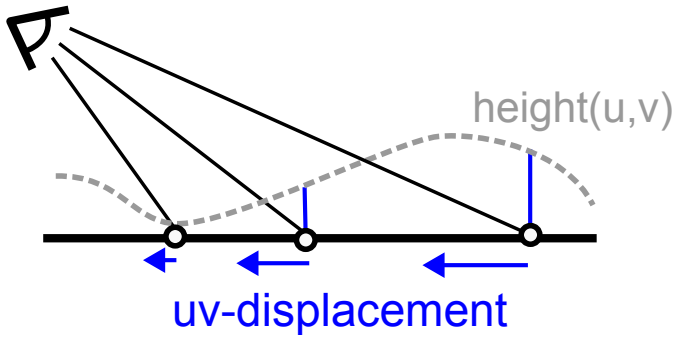


Figure 50: Parallax mapping : le rayon de vue (en rouge) devrait intersecter la surface en relief au point B, mais le fragment est en A. On décale les coordonnées de texture pour lire au point B.

Soit V le vecteur de vue exprimé dans l'espace tangent et h la hauteur lue dans la height map au point courant. Le décalage des coordonnées de texture est :

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

La division par V_z amplifie le décalage aux angles rasants (quand V_z est petit), ce qui correspond au comportement physique attendu : la parallaxe est plus visible sous un angle oblique.

Implémentation GLSL

Fragment shader avec 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;  // contrôle l'amplitude du relief (~0.05-0.1)

out vec4 FragColor;

void main() {
    // Vecteur de vue dans l'espace tangent
    vec3 V_world = normalize(camera_position - fragment.position);
    vec3 V_tangent = normalize(transpose(fragment.TBN) * V_world);

    // Lire la hauteur et calculer le décalage
    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;

    // Utiliser les coordonnées décalées pour la texture et la 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 avec la normale et la couleur décalées
    // ...
}
```

Steep Parallax Mapping

Le parallax mapping simple est une approximation du premier ordre qui fonctionne bien pour des reliefs faibles. Pour des reliefs plus prononcés, il produit des artefacts (glissements, distorsions). Le **Steep Parallax Mapping** (ou Parallax Occlusion Mapping) améliore la qualité en parcourant le rayon de vue **pas à pas** à travers la carte de hauteur :

1. Discrétiser le rayon de vue en N étapes dans l'espace tangent.
2. À chaque étape, comparer la hauteur courante du rayon avec la hauteur lue dans la height map.
3. S'arrêter lorsque le rayon passe en dessous de la surface (la hauteur du rayon devient inférieure à la hauteur de la height map).
4. Interpoler entre les deux dernières étapes pour obtenir un point d'intersection précis.

```
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;
}
```

Cette méthode est plus coûteuse (une lecture de texture par étape), mais produit des résultats visuellement convaincants même pour des reliefs importants, avec une gestion correcte de l'auto-occultation des détails du relief.