POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

D E
I B

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

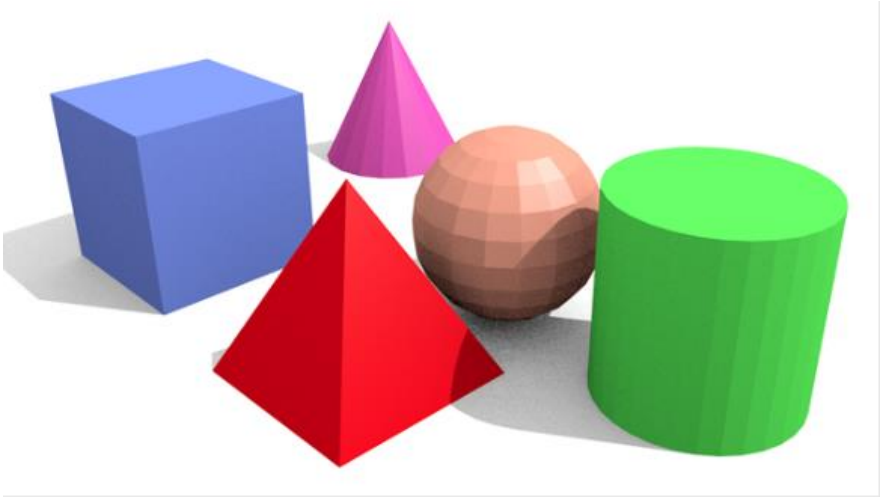
Computer Graphics

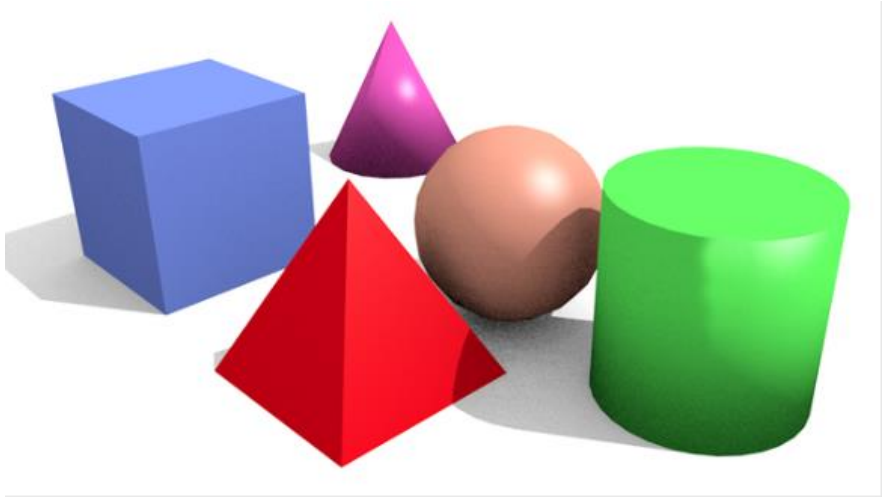
- Smooth shading

Smooth shading of polygonal objects

As introduced, meshes are polygonal objects with sharp edges that, with special rendering techniques, can appear smooth.

The smoothing effect is also connected with performance issues, which decide how many times the rendering equation is solved.



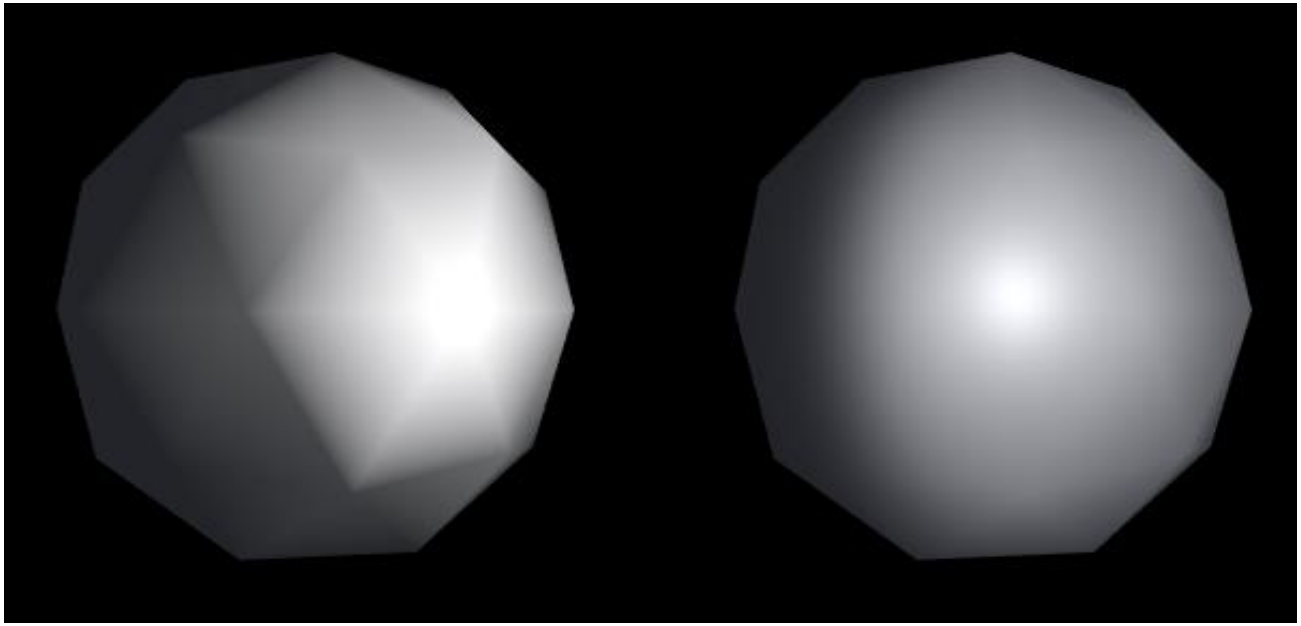


Smooth shading of polygonal objects

The solution can be computed *per-vertex* or *per-pixel*.

Per vertex

Per pixel



Smooth shading of polygonal objects

Today's applications are characterized by 3D models composed of around 100'000 vertices and occupying at least 2'000'000 pixels on the screen. Moreover, to create smoother images, the rendering equation can be solved several times per pixel to avoid the aliasing effect.

Solving the rendering equation per pixel provides more visually appealing images, at the expense of performance reductions of around one order of magnitude.

Smooth shading of polygonal objects

The two most common smooth shading techniques are:

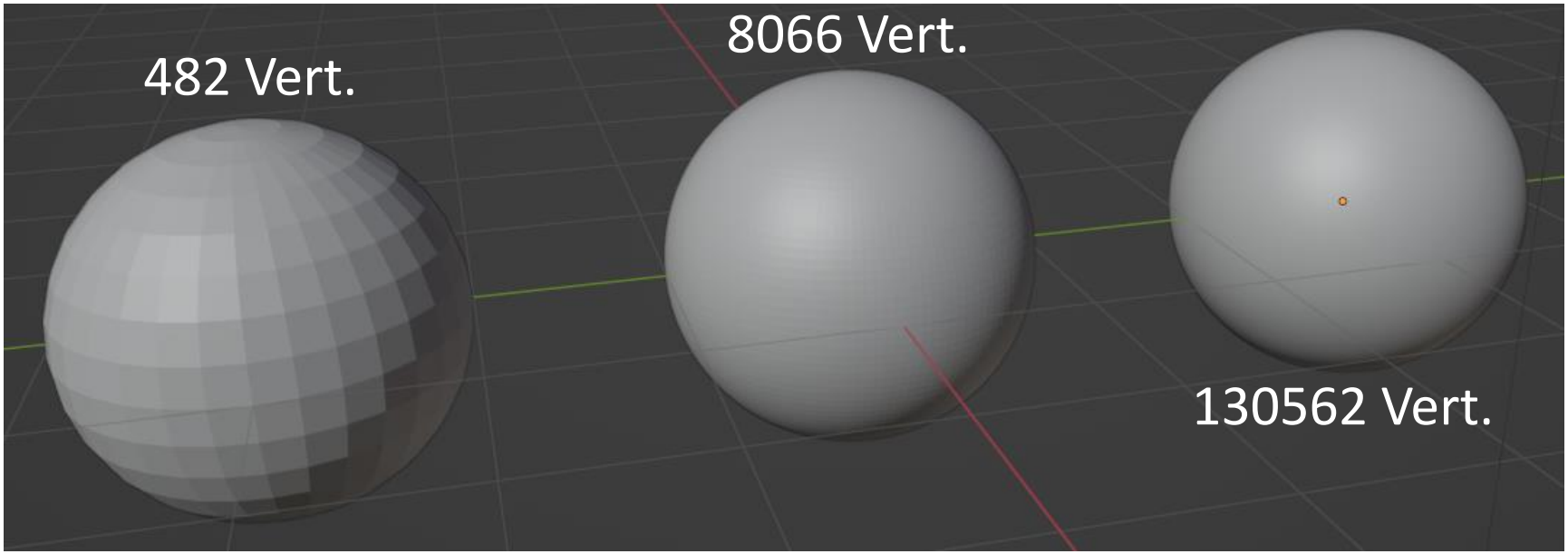
- *Gouraud shading* (per-vertex)
- *Phong shading* (per-pixel)

In the first case, smoothing occurs by blending the colors generated, and in the second by interpolating the parameters passed to the shaders involved in the solution of the rendering equations.

Vertex normal vectors

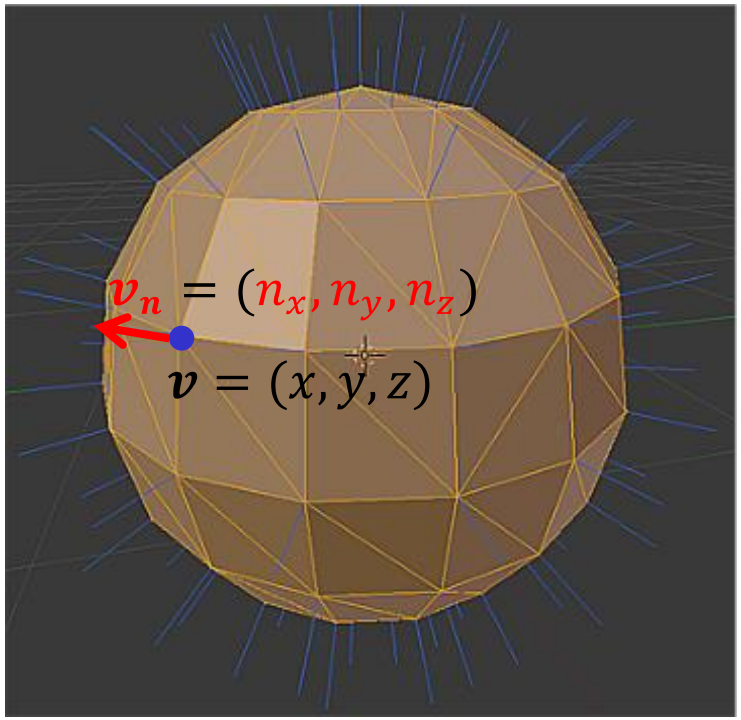
Let us consider a curved surface. A mesh can approximate the surface by sampling it in a finite set of points.

The higher is the sampling, the closer will be the approximation to the real surface.



Vertex normal vectors

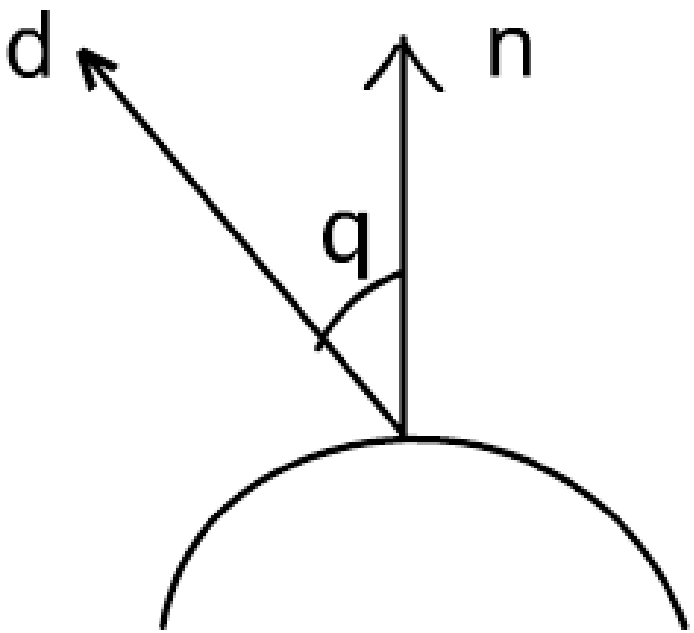
To improve rendering of curved surfaces, the encoding of a vertex is extended to 6 values, which define both the position of the vertex and the direction of the normal vector to the surface in that 3D point.



$$v = (x, y, z, n_x, n_y, n_z)$$

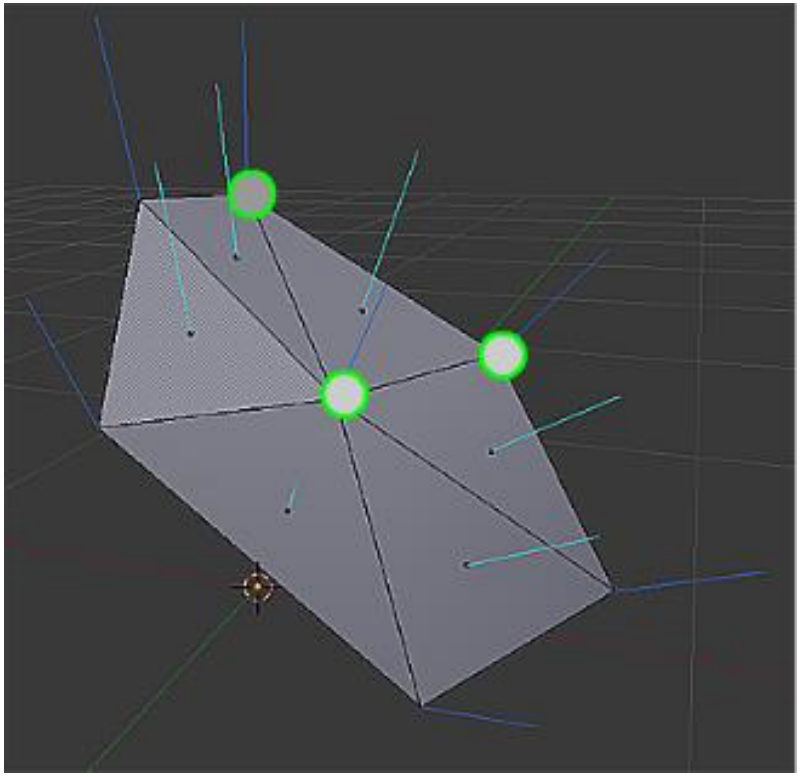
Vertex normal vectors

As we have seen in the previous lessons, the approximation of the rendering equation uses the normal vector direction to compute the color of a surface.



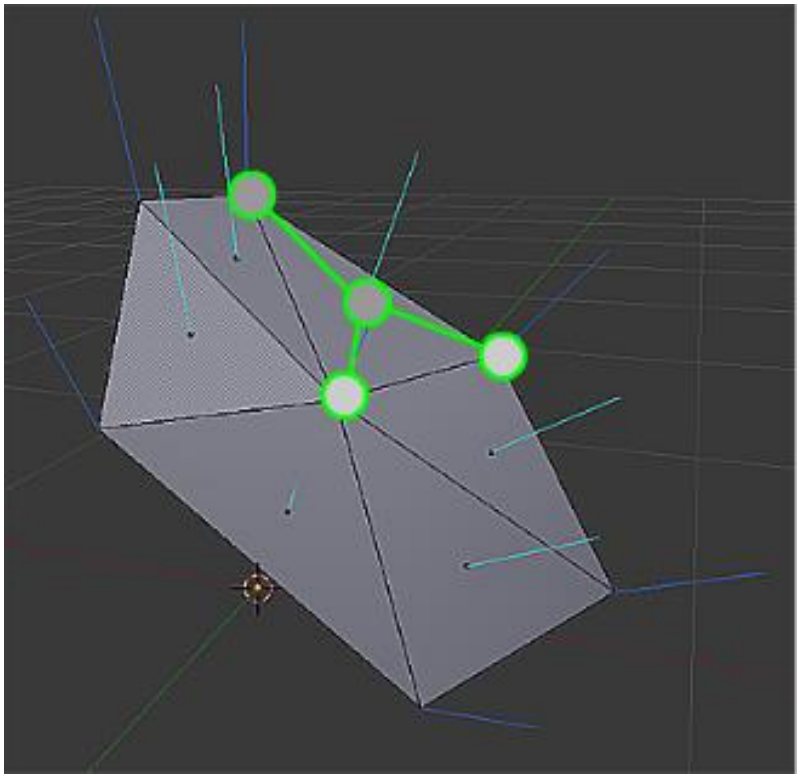
Vertex normal vectors

When considering a mesh, the rendering equation determines the colors of the pixels in the three vertices of each triangle, according to the associated normal vectors.



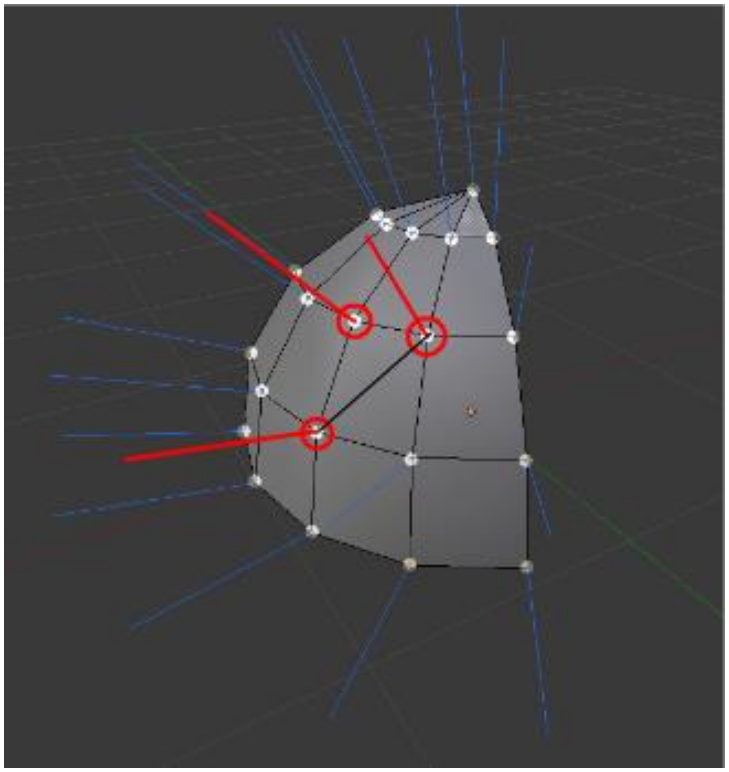
Vertex normal vectors

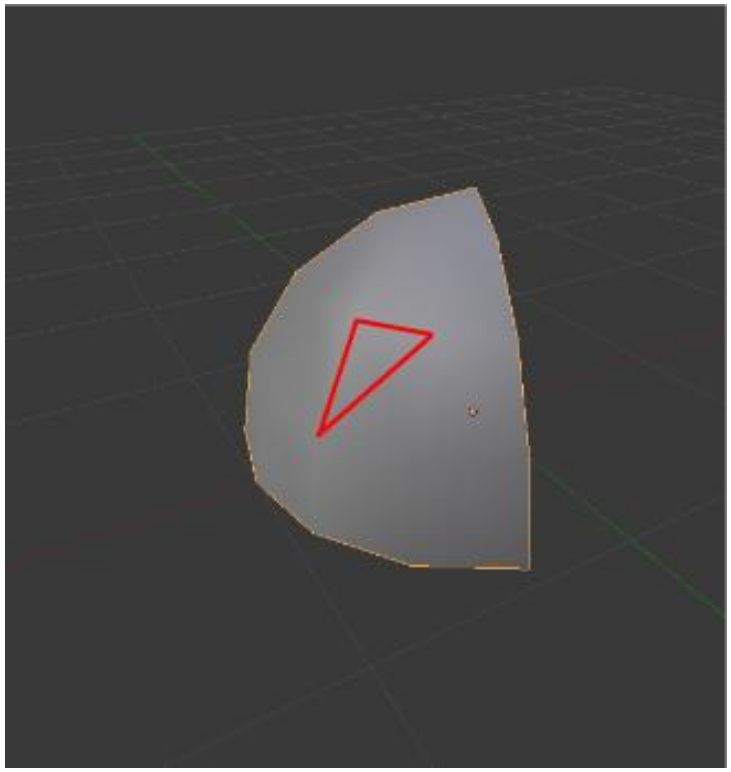
The colors of the internal pixels of the triangles are then computed using an interpolation technique.



Gouraud shading

The Gouraud (per vertex) shading technique computes the colors of the inner pixels of a triangle by interpolating the vertex colors.



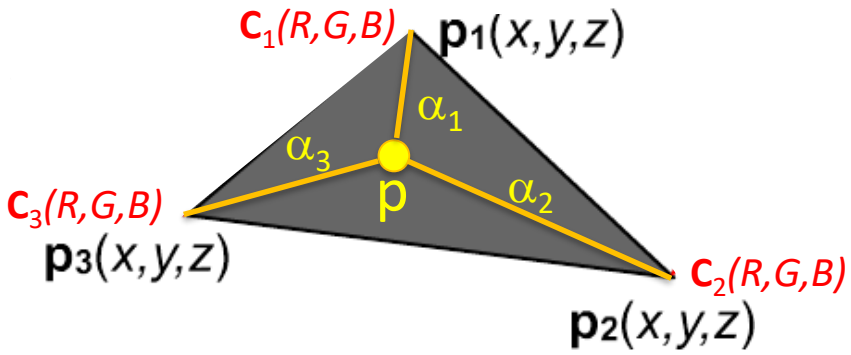


Gouraud shading

From geometric consideration, the position of a point inside a triangle can be computed with a convex linear combination of its vertices.

The same interpolation coefficients used can be used for the colors:

- For an internal point p of a triangle, its coefficients are determined (with a linear system of equations).
- Such coefficients are used for interpolating the colors.



$$\alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 = p$$
$$c = \alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3$$

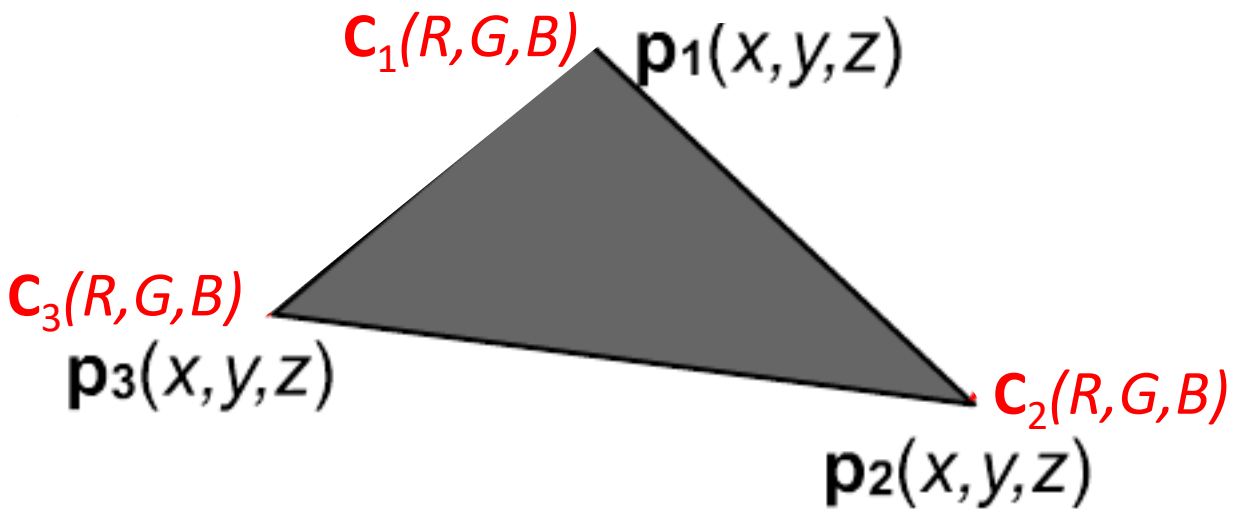
Gouraud shading

For objects that do not move in the scene, which are illuminated by static lights, and whose material BRDF does not depend on the direction of the observer (i.e. just diffuse component following the Lambert model), vertex colors can be pre-computed and stored with the geometry.

Computation of the light model can then be disabled during real-time rendering, and vertex normal vectors can be replaced by vertex colors.

Gouraud shading

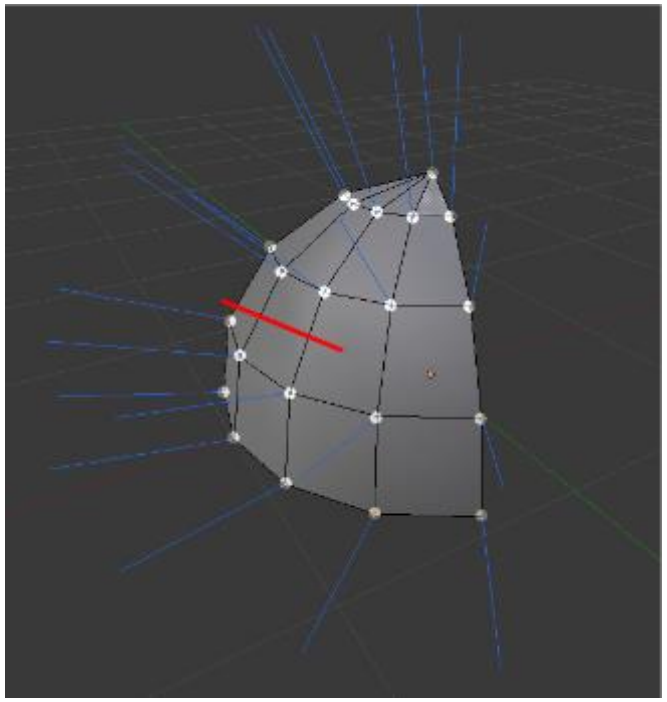
Color pre-computation is usually done in 3D authoring software such as Blender. Moreover, vertex color usually occupy less memory than normal vector directions (i.e. *4 vs. 12 bytes*).



Phong shading

The *Phong shading alogirthm* computes the color of each internal pixel separately. This is thus a *per-pixel* shading algorithm.

In this case, the *vertex normal vectors* are interpolated to approximate the normal vector direction to the actual surface in the internal points of the triangle.

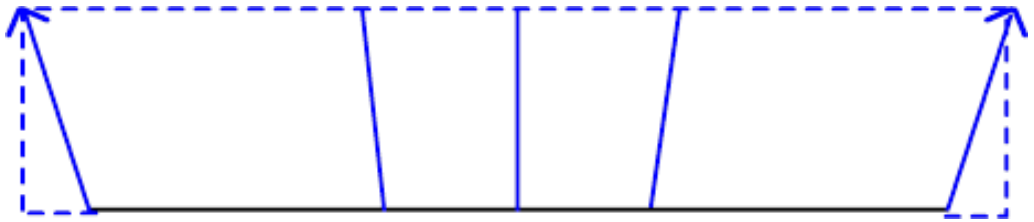


$$\alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 = p$$
$$n = \alpha_1 n_1 + \alpha_2 n_2 + \alpha_3 n_3$$

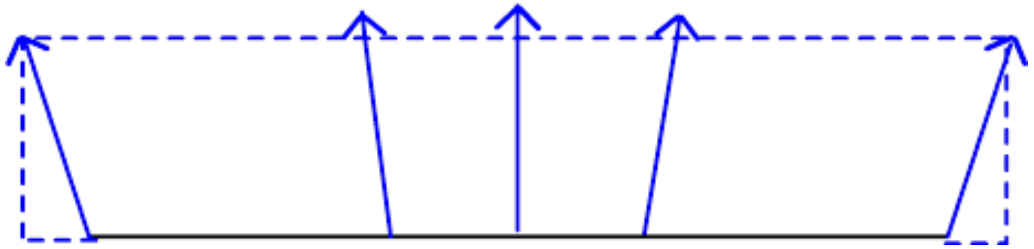
Phong shading

Interpolation is performed by considering the x, y, z components of the normal vectors separately.

However, this may lead to interpolated vectors that are no longer unitary (even if the normal vectors associated to the vertices of the triangle are so).

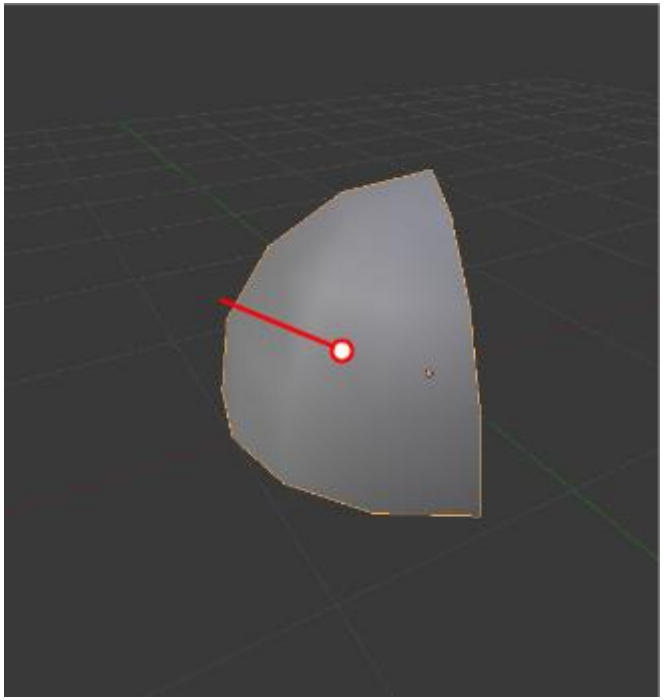


For this reason, interpolated normal vectors must be *normalized* *(made unitary again)* at every step to restore the required size.



Phong shading

The illumination model is then computed for every pixel, using the interpolated normal vectors together with the other constants required by the light model and the BRDF in the rendering equation.



Phong shading

The Phong shading method is much more expensive than the Gouraud method because it requires the solution of the rendering equation for every pixel.

This can reduce a lot the performances, especially when considering several light sources (since the illumination model has linear complexity with respect to the number of lights).

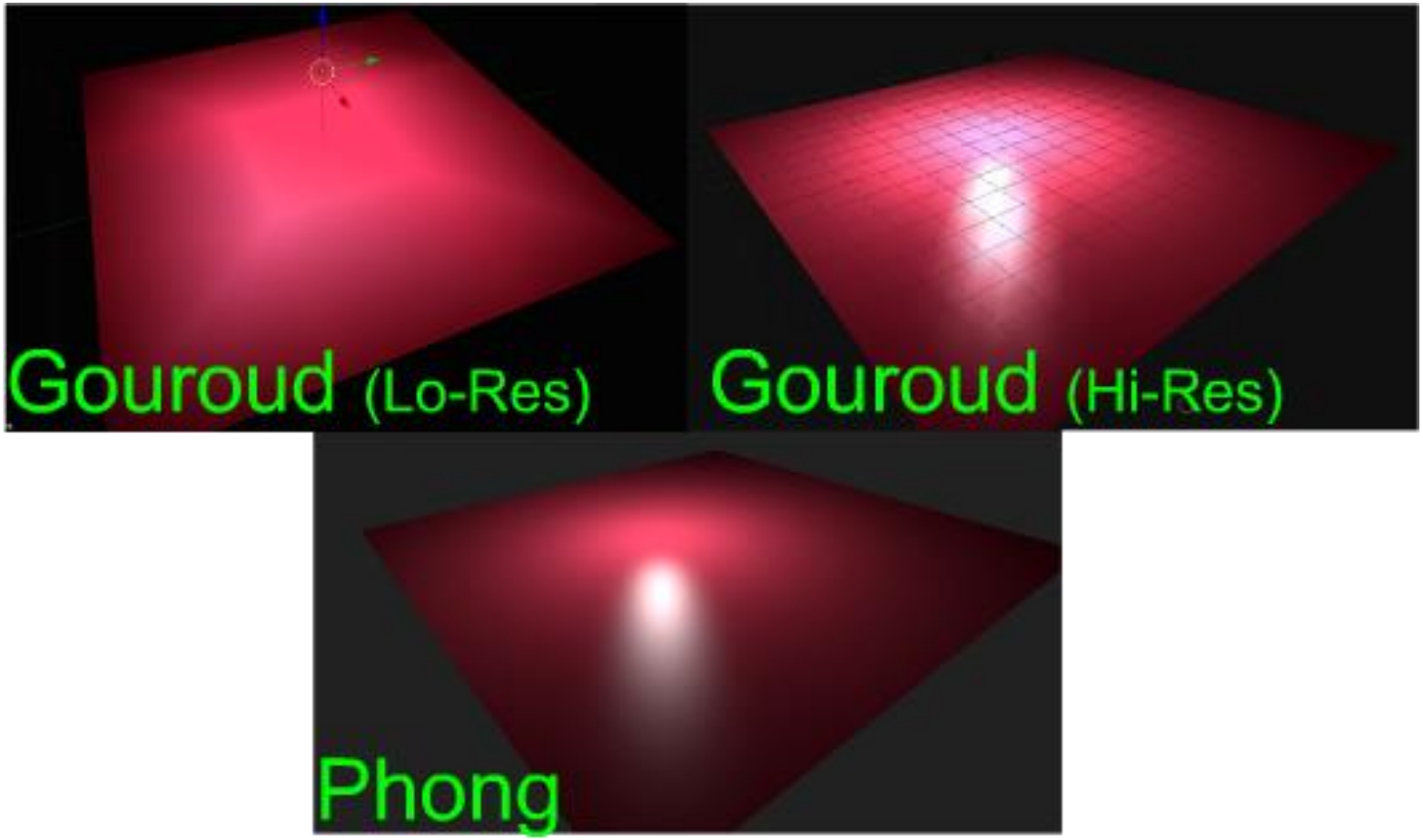
However it can produce high quality rendering even for models composed by few vertices.

Phong shading

The Gouraud technique may produce artifacts on the image, and cannot capture specific lighting conditions: this happens because interpolation might miss some of the details that appear between the vertices.

The two methods however tends to give the same result when considering *geometries composed by many vertices*, since in this case, the area of each triangle is just a few pixel wide.

Phong shading



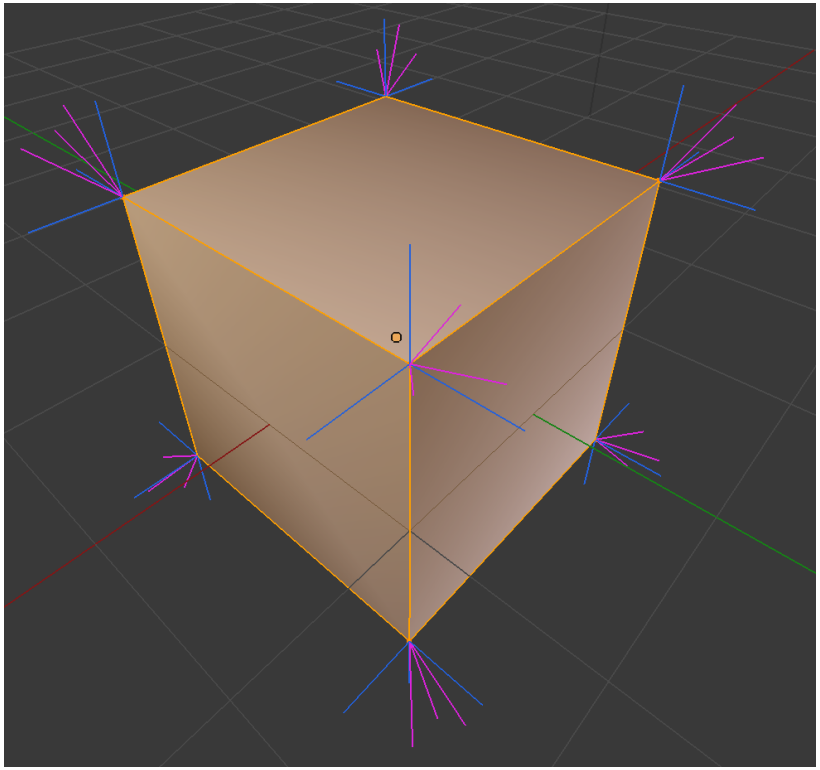
Phong shading

Note: do not confuse the *Phong specular model* with the *Phong shading technique*.

Vertex normal vectors directions

The normal vector associated to a vertex might be different from the geometric one determined by the triangle to which it belongs.

In general, it can be an arbitrary vector, that might have nothing to do with the associated surfaces. However, normal vectors completely uncorrelated with the geometry are rarely useful.



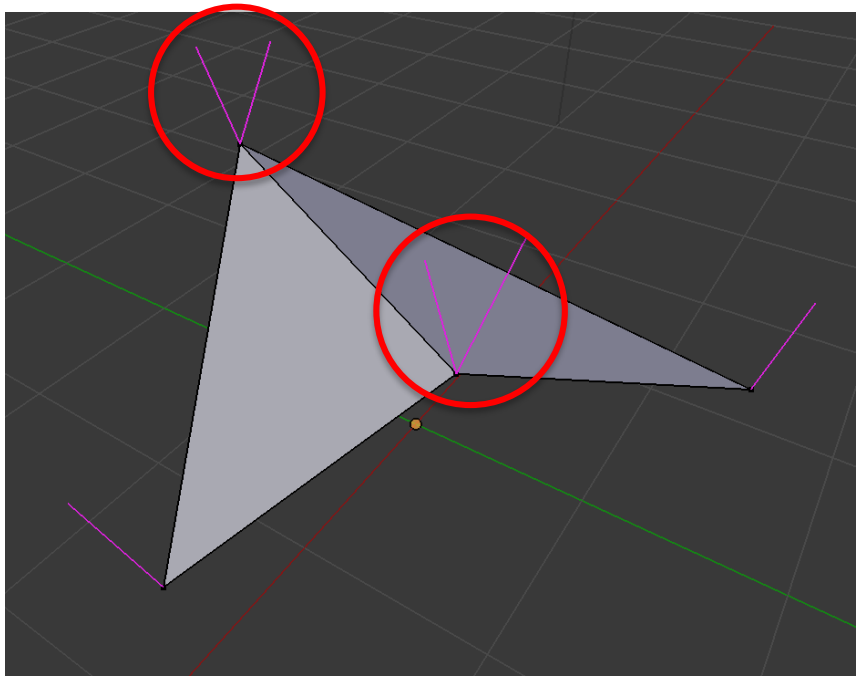
Blue: real normals

Purple: fake normals

Note the strange and unnatural shading of the surfaces of this cube.

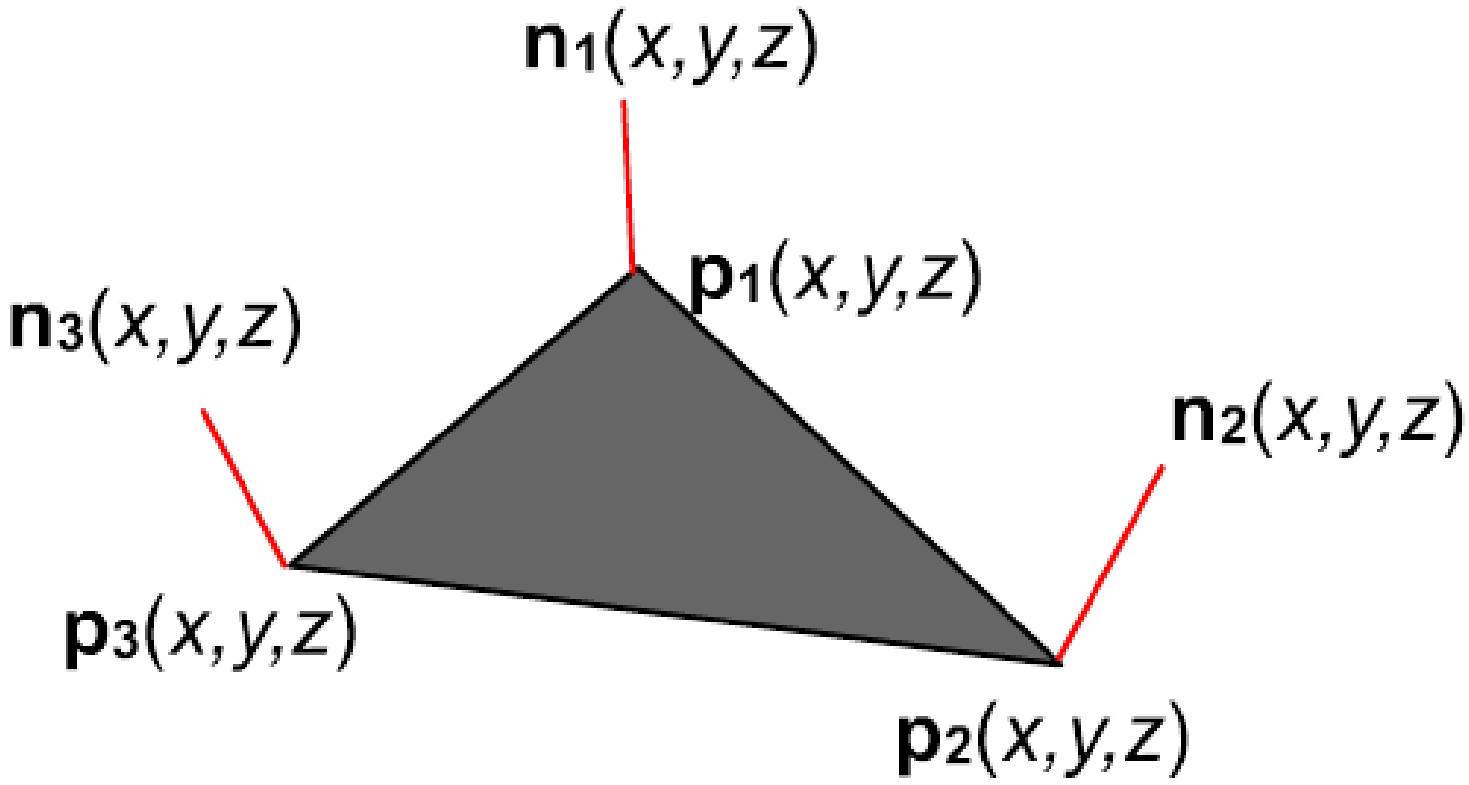
Vertex normal vectors directions

With this definition, the normal vector direction is a property of a (*vertex, triangle) couple*, and not of a *vertex alone*: two triangles might have a vertex in the same spatial position, but characterized by two different normal vector direction.



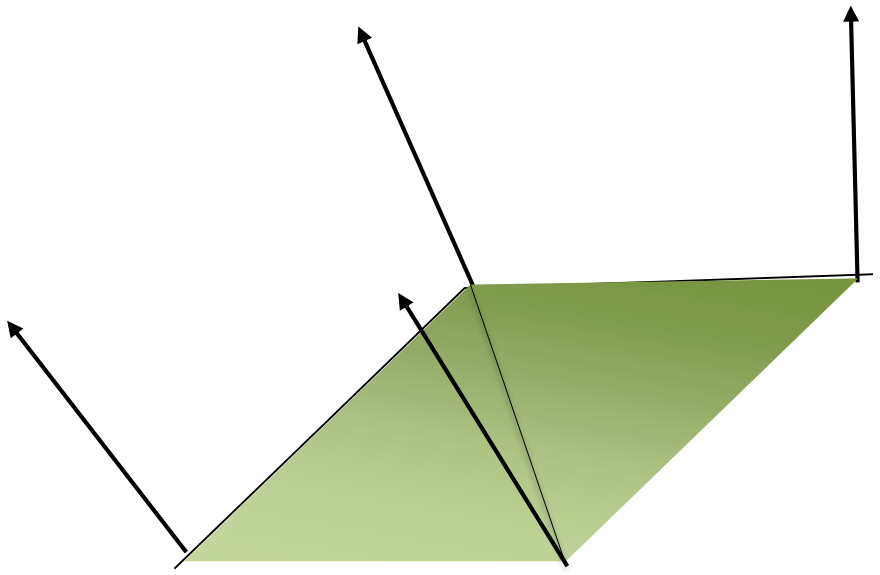
Vertex normal vectors directions

One triangle is thus defined by *18* values (3 vertices x (3 vertex coordinates + 3 normal vector direction components)) :



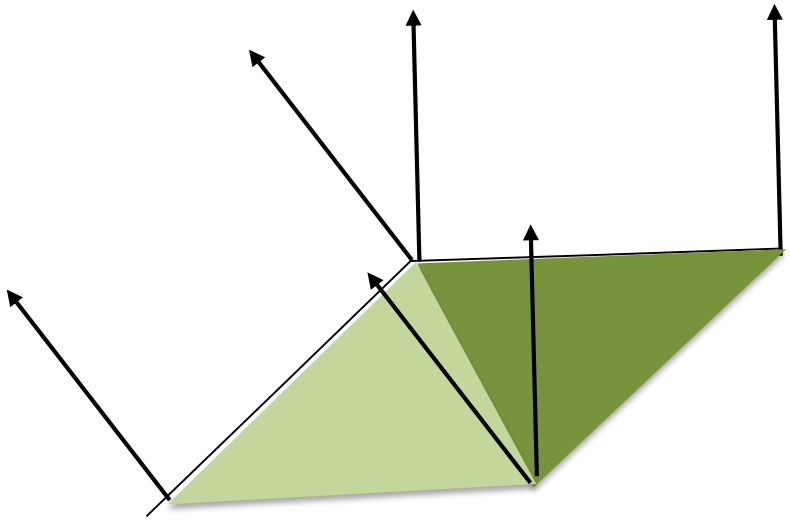
Vertex normal vectors directions

Modeler can exploit these properties to encode both smooth and sharp surfaces. The vertices in the same position of two adjacent triangles can have the *same normal vector direction* to produce a *smooth surface*.



Vertex normal vectors directions

Two vertices in two triangles, with *the same position but different normal vector direction*, can be used encode *sharp surfaces*.



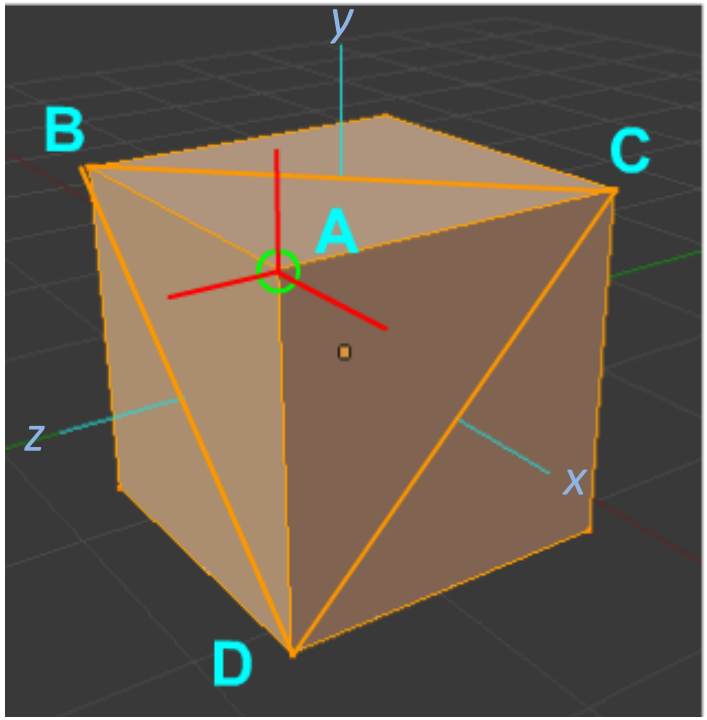
Vertex normal vectors: considerations

Even in simple figures such as a cube, there could be several different 6-tuples that have identical position, but different normal vector direction.

A in triangle *ABC*:
(1,1,1,0,1,0)

A in triangle *ACD*:
(1,1,1,1,0,0)

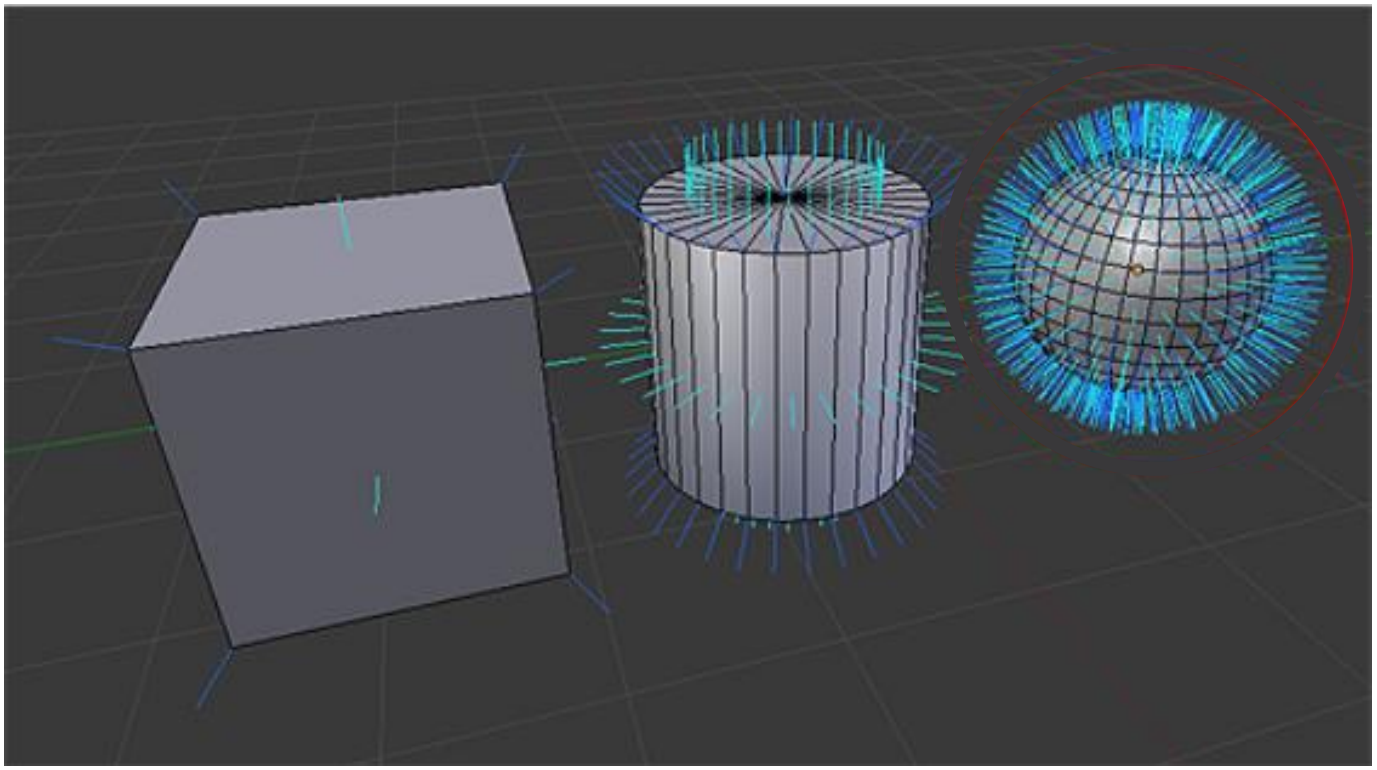
A in triangle *ADB*:
(1,1,1,0,0,1)



Here, vertex A, belongs to three triangles, and it has a different normal direction, on for each side.

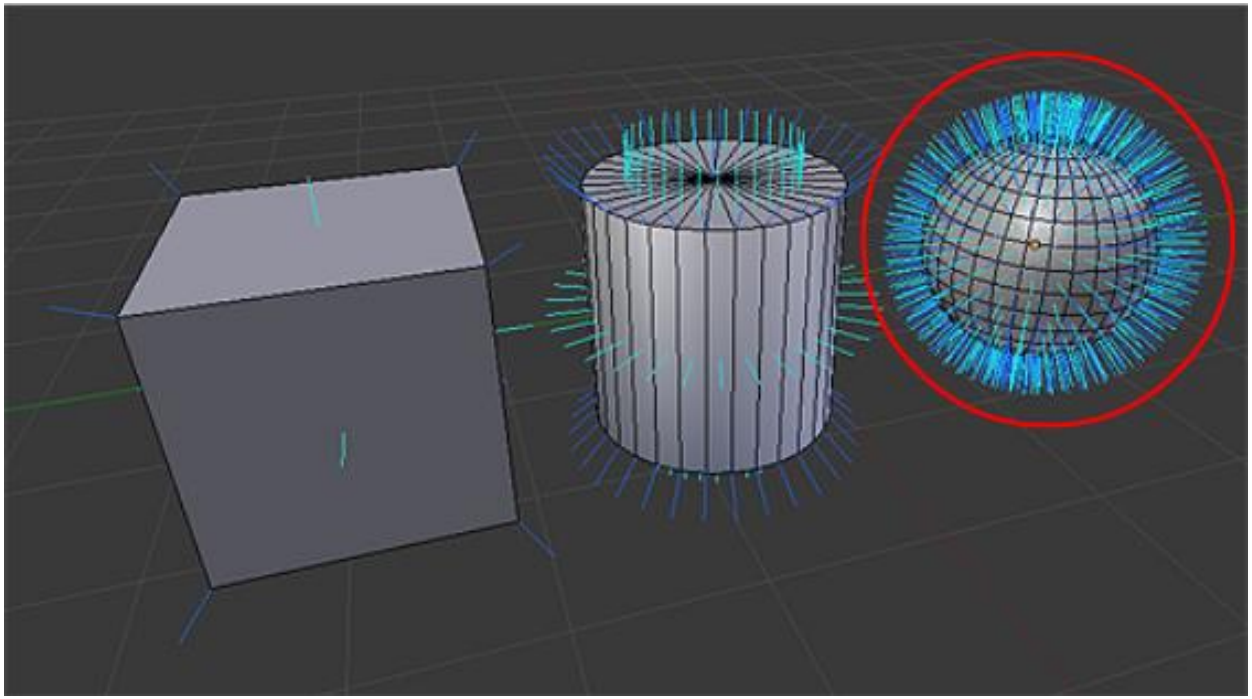
Vertex normal vectors: considerations

Let us consider how we can encode three basic solid: a cube, a cylinder, and a sphere.



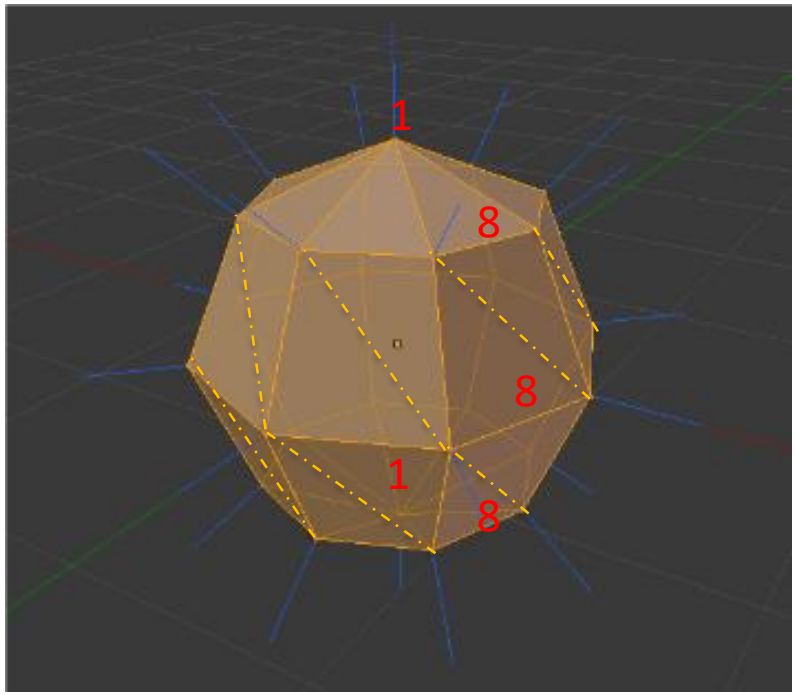
Vertex normal vectors: considerations

In a smooth solid such as a sphere, each vertex in a position has the same normal vector direction in all the triangles to which it belongs. This normal vector direction corresponds to the one of the sphere it is approximating.



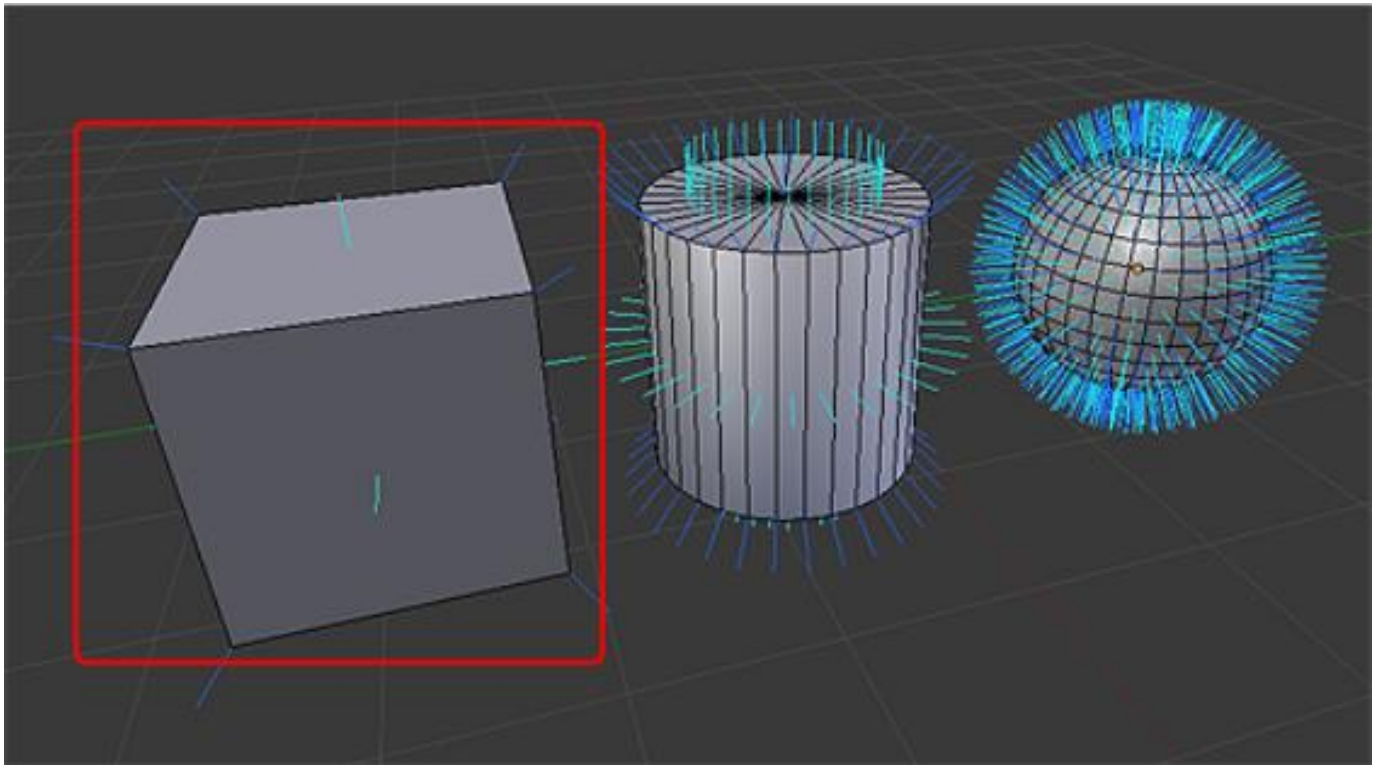
Vertex normal vectors: considerations

For instance, a sphere approximated by *4* rings and *8* slices is characterized by *48* triangles *(8 (top fan)+8*2 (upper stripe)+8*2 (lower stripe) +8 (bottom fan))* and 26 vertices *(1 (top tip)+3 x 8 (inner rings)+1 (bottom tip))*, since all vertices share both the position and the normal vector direction.



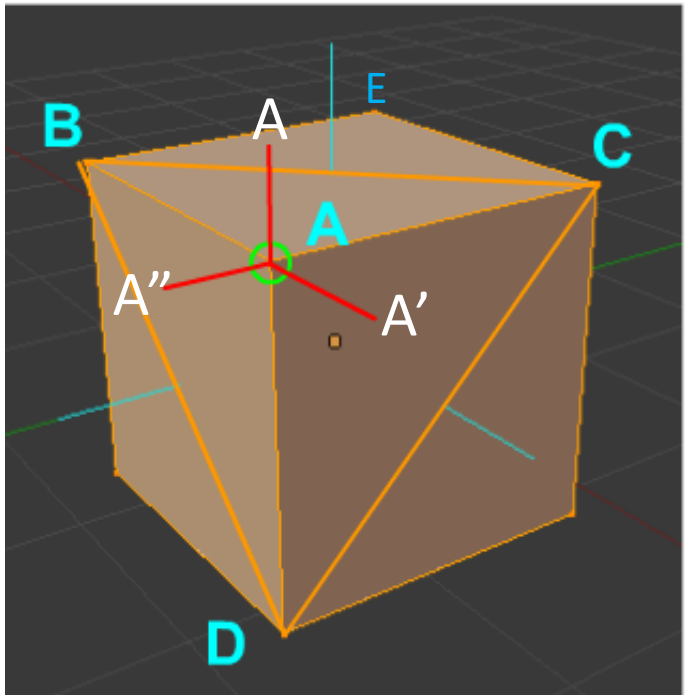
Vertex normal vectors: considerations

If all the faces that share the same vertex have a different normal vector direction, then the edges appear to be sharp. This is the case for example of the cube.



Vertex normal vectors: considerations

In encoding such as *indexed triangle lists*, the vertex must be repeated as many times as different faces: each one with the same position, but a different normal vector direction. For example, the vertex in position A of the cube corresponds to three different vertices in its mesh encoding: A, A’ and A”.



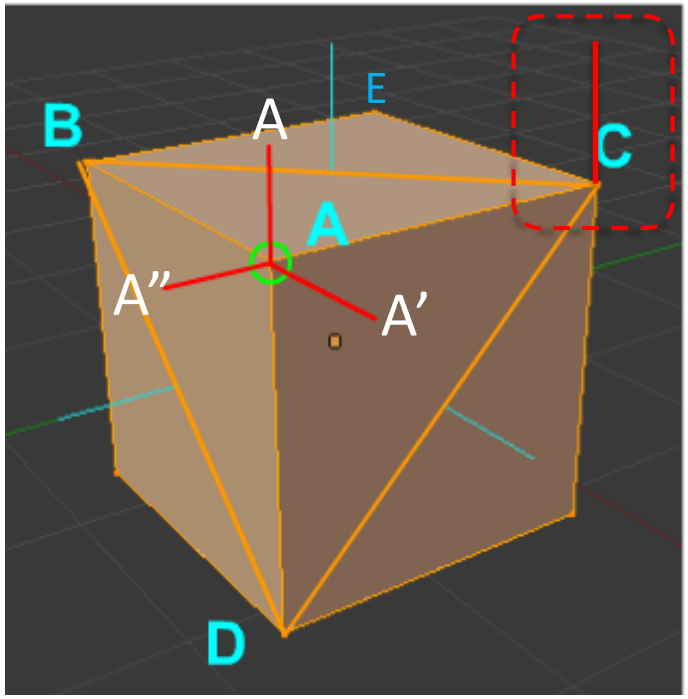
Vertex **A** in *ABC*:
$\mathbf{p}_A(1,1,1)$ $\mathbf{n}_A(0,1,0)$

Vertex **A'** in *ACD*:
$\mathbf{p}_{A'}(1,1,1)$ $\mathbf{n}_{A'}(1,0,0)$

Vertex **A"** in *ADB*:
$\mathbf{p}_{A''}(1,1,1)$ $\mathbf{n}_{A''}(0,0,1)$

Vertex normal vectors: considerations

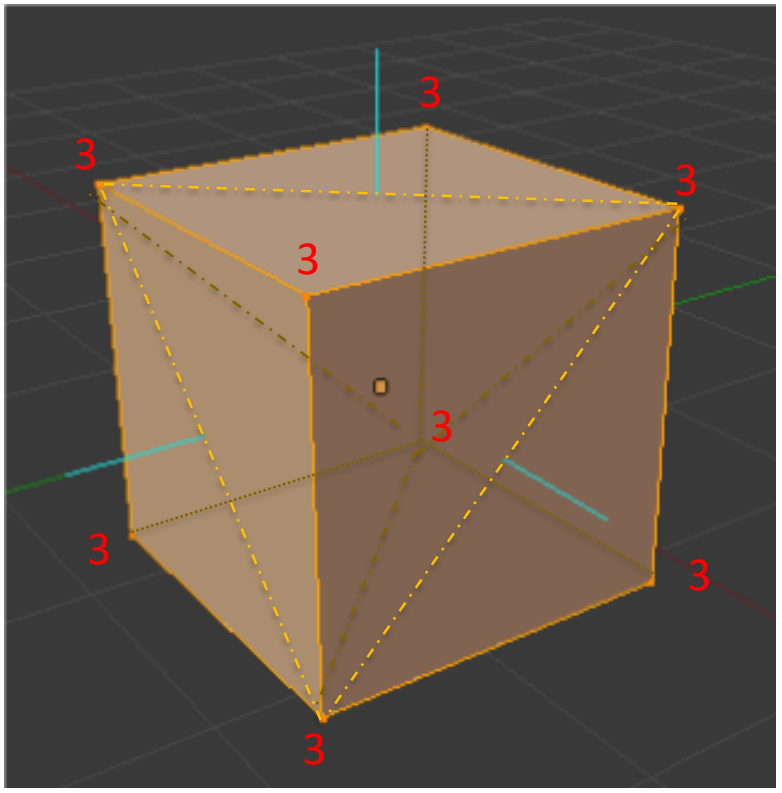
Note however that some vertex might still be reused: for example, in the cube below, vertex C with the normal pointing up, is share by both triangle ABC and BCE.



$\mathbf{p}_C(1,1,-1)$ $\mathbf{n}_C(0,1,0)$

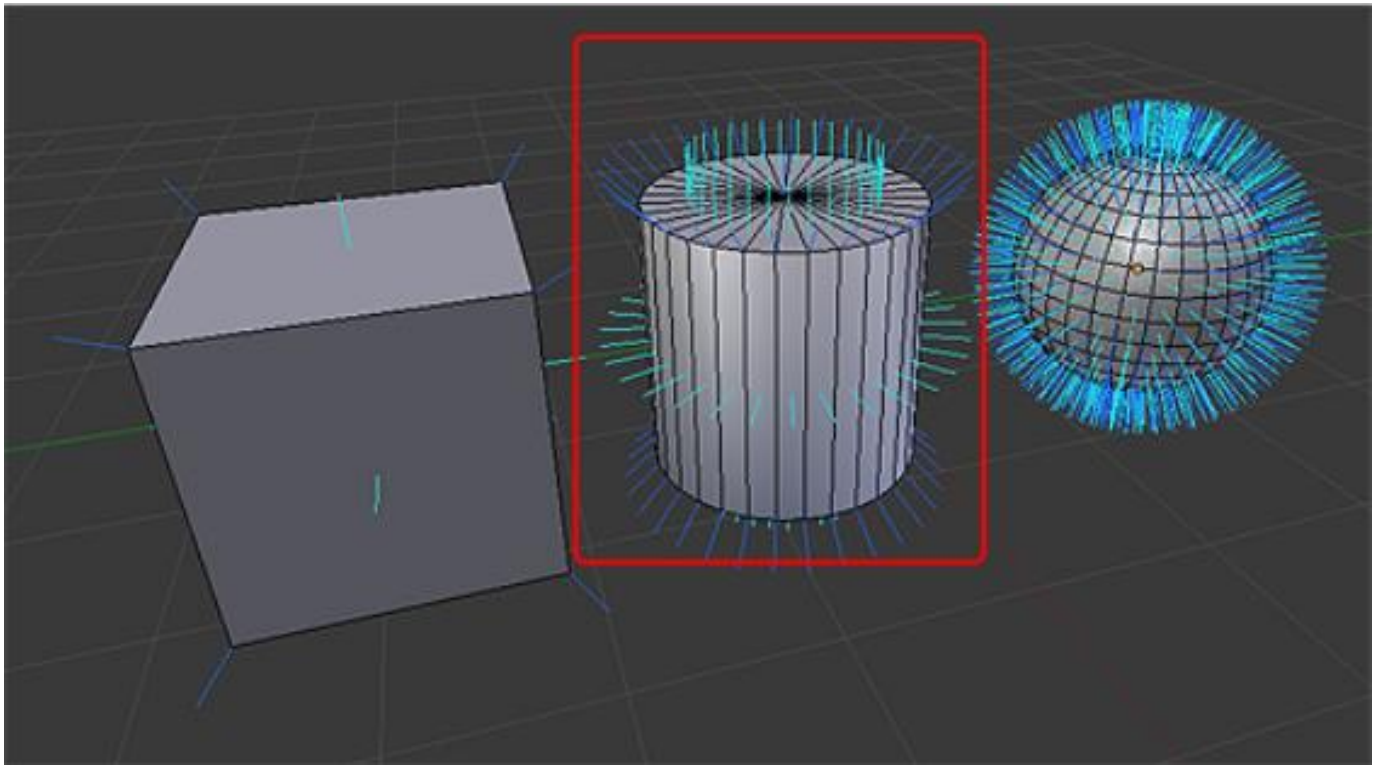
Vertex normal vectors: considerations

A cube is then characterized by *12* triangles (*2* per face), identified by *24* vertices (*8* positions with *3* different normal vectors each).



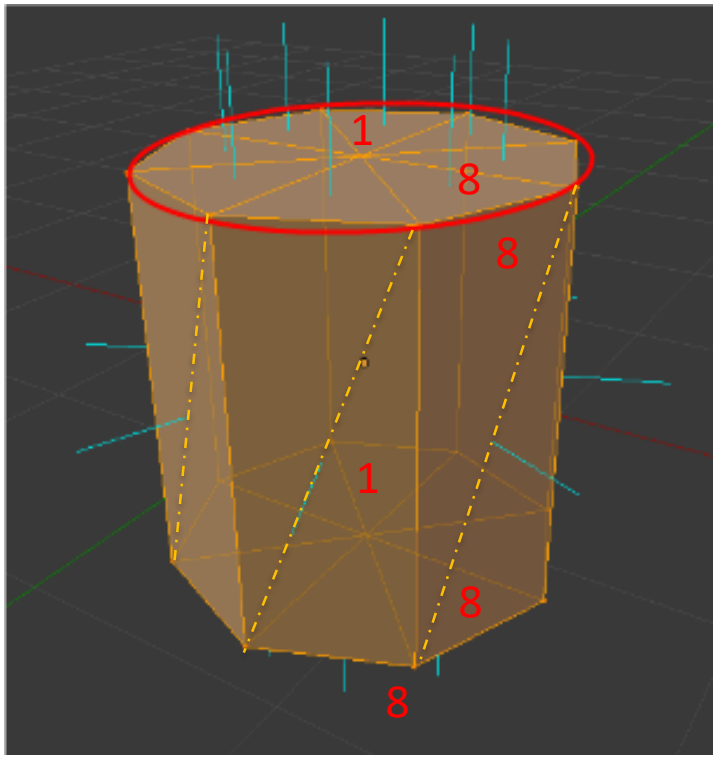
Vertex normal vectors: considerations

If some triangles have the same normal vector, but some other have different ones, the object has a sharp rounded border. This is the case of the cap of a cylinder.



Vertex normal vectors: considerations

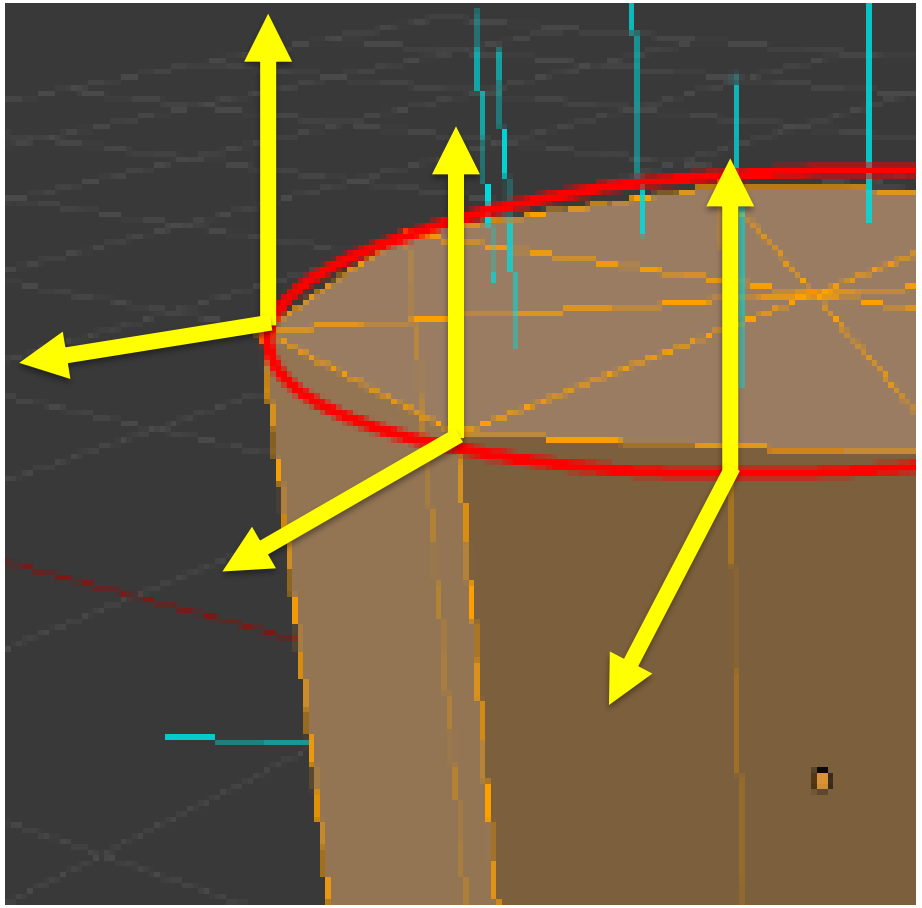
A cylinder, approximated by an *8* sides prism, is defined by *32* triangles (*8* *(top cap)*+*16* *(side)*+*8* (bottom cap)) and *34* vertices (*1* *(top cap center)*+*8* *(top cap ring)* + *8* *(upper side ring)*+*8* *(bottom side ring)* +*8* *(bottom cap ring)*+*1* *(bottom cap center)*) , since the vertices on the cap exist with two different normal vector directions.



Vertex normal vectors: considerations

In particular, for the vertices on the ring:

- The one belonging to the cap, have the normal vector oriented along the *y-axis*.
- The one belonging to the side, are directed outside, along the radius of the cap, with they *y-axis* component equal to zero.
- Each vertex will is shared by at least two triangles.



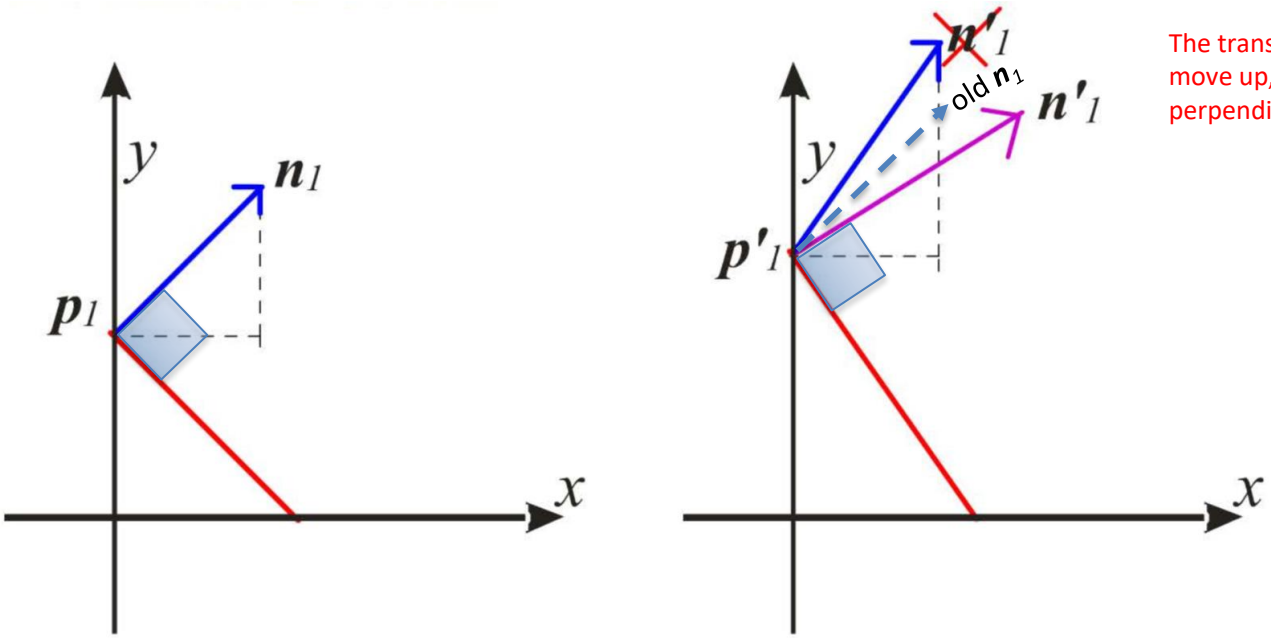
Transforming vertex normal

If the normal vectors are stored with an object that is moved with a transform encoded in a *World Matrix, they must be transformed as well.*

But the transformations of the normal vectors cannot be performed in the same way as the positions of vertices.

Transforming vertex normal

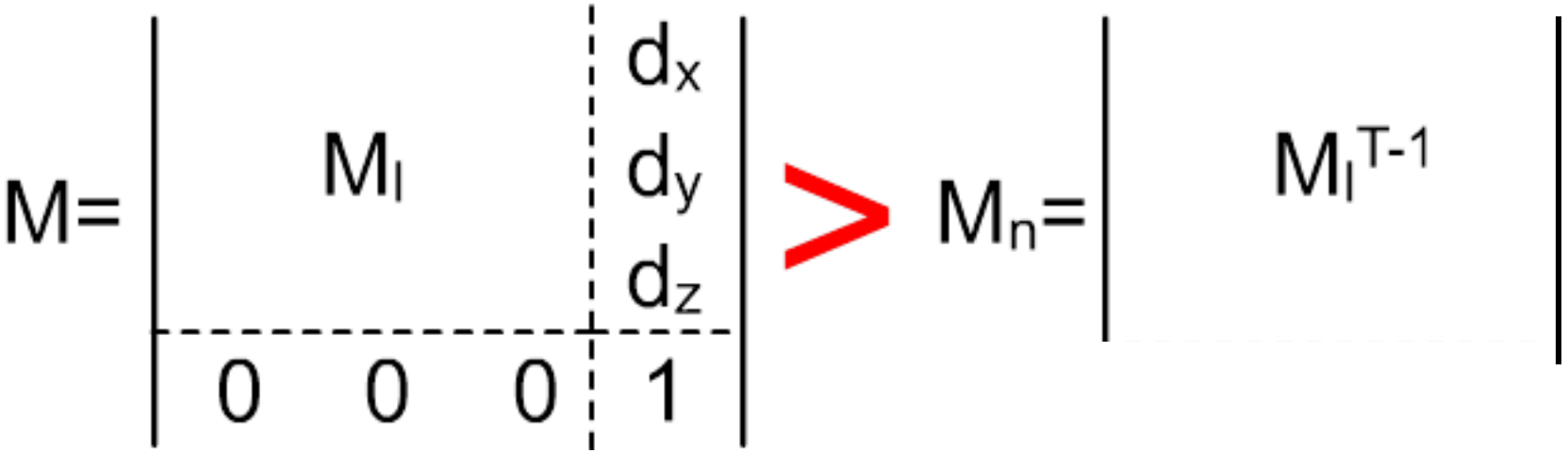
The main reason is that normal vectors are 3D *directions* (three-components vectors) and not homogeneous coordinates (four-components vectors). For example, it can be immediately seen that a scaling along the *y* axis, would produce a wrong result.



The transformed vector would move up, while the actual perpendicular should move down.

Transforming vertex normal

The transform matrix for the normal vectors can be derived from the 4x4 matrix that transforms the corresponding homogenous coordinates, by computing the *inverse-transpose* of its *3x3 upper-left sub-matrix*.



Transforming vertex normal – proof [just for completeness]

The proof is quite simple, and starts from the defition of normal vector: a vector that is *orthogonal to any vector of the considered surface* - that is whose scalar product is zero. In matrix notation, the scalar product can be written as:

$$(v_x \quad v_y \quad v_z) \cdot \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = v^T \cdot n = 0$$

Let us call M the transformation matrix, and N the one we are looking for to transform the normal. We then have that:

$$v' = M_l \cdot v$$

$$n' = N \cdot n$$

We want to prove that:

$$N = (M_l^T)^{-1}$$

The proof is here just for completeness, and it does not need to be studied!

Transforming vertex normal – proof [just for completeness]

The reason for which $v' = M_l \cdot v$ is the following. Vector v is a vector on the considered surface. Let us consider the two end-points a and b of the vector on the surface. Then we have:

$$\begin{pmatrix} v_x & v_y & v_z \end{pmatrix} = \begin{pmatrix} b_x & b_y & b_z \end{pmatrix} - \begin{pmatrix} a_x & a_y & a_z \end{pmatrix} \qquad v = b - a$$

and:

$$\begin{pmatrix} v'_x & v'_y & v'_z \end{pmatrix} = \begin{pmatrix} b'_x & b'_y & b'_z \end{pmatrix} - \begin{pmatrix} a'_x & a'_y & a'_z \end{pmatrix} \qquad v' = b' - a'$$

with:

$$\begin{pmatrix} a'_x & a'_y & a'_z & 1 \end{pmatrix}^T = M \times \begin{pmatrix} a_x & a_y & a_z & 1 \end{pmatrix}^T \quad \begin{pmatrix} b'_x & b'_y & b'_z & 1 \end{pmatrix}^T = M \times \begin{pmatrix} b_x & b_y & b_z & 1 \end{pmatrix}^T$$

since:

$$M = \begin{vmatrix} M_l & \mathbf{d} \\ \mathbf{0} & 1 \end{vmatrix}$$

The proof is here just for completeness, and it does not need to be studied!

We have:

$$a' = M_l \times a + \mathbf{d} \qquad b' = M_l \times b + \mathbf{d}$$

$$v' = b' - a' = M_l \times b + \mathbf{d} - (M_l \times a + \mathbf{d}) = M_l \times (b - a) = M_l \times v$$

Transforming vertex normal – proof [just for completeness]

If N is the correct matrix, the transformed normal vector should be orthogonal to the transformed vectors:

$$v'^T \cdot n' = 0$$

Let us insert the definition of the two transformed vectors:

$$= (M_l \cdot v)^T \cdot (N \cdot n) =$$

Then we apply the propoerty of the transpose of the product of two matrices:

$$= v^T \cdot M_l^T \cdot N \cdot n =$$

We then insert the target definition of N in the expression:

$$= v^T \cdot M_l^T \cdot (M_l^T)^{-1} \cdot n =$$

With a few passages, we obtain:

$$= v^T \cdot I \cdot n = v^T \cdot n = 0$$

The proof is here just for completeness, and it does not need to be studied!

Transforming vertex normal: special case

In the very special case in which the transpose of the 3x3 rotation matrix is identical to its inverse, the proposed procedure will have no effect.

This happens when ***no scaling or shear transformations are performed***.

In this very special case, then we can transform the normal vector directly with the world matrix M with a simple trick:

$$n' = (M \cdot |n_x \quad n_y \quad n_z \quad 0|^T).xyz$$

If the normal vectors are normalized before being used (which is anyway required for per-pixel shading), the previous procedure would work also when uniform scaling is applied.

Marco Gribaudo

Associate Professor

CONTACTS

Tel. +39 02 2399 3568

marco.gribaudo@polimi.it

https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)