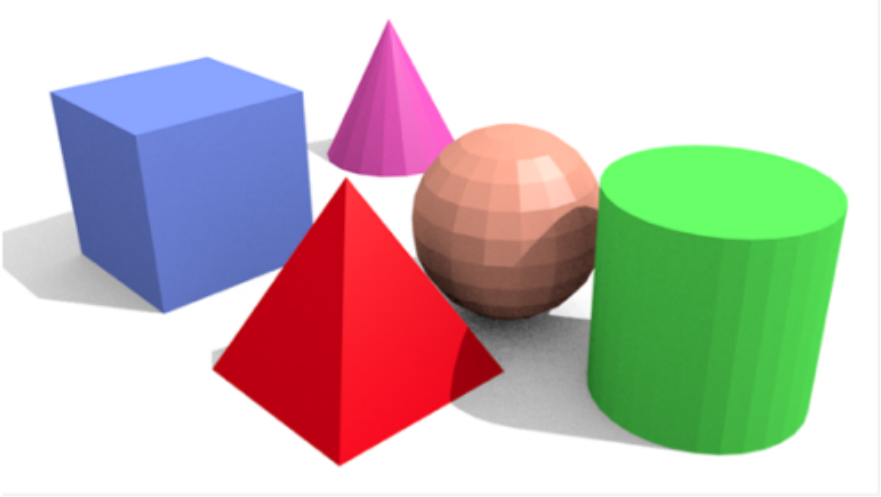
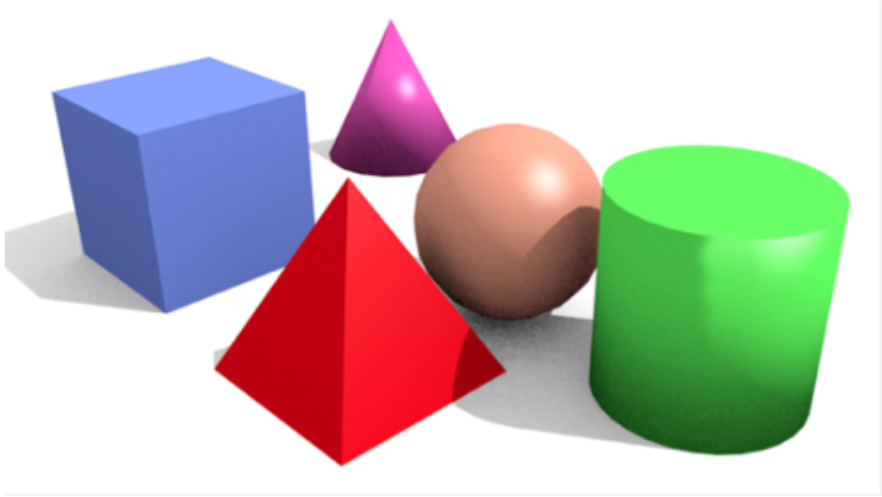
Smooth shading and Shader loading

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.



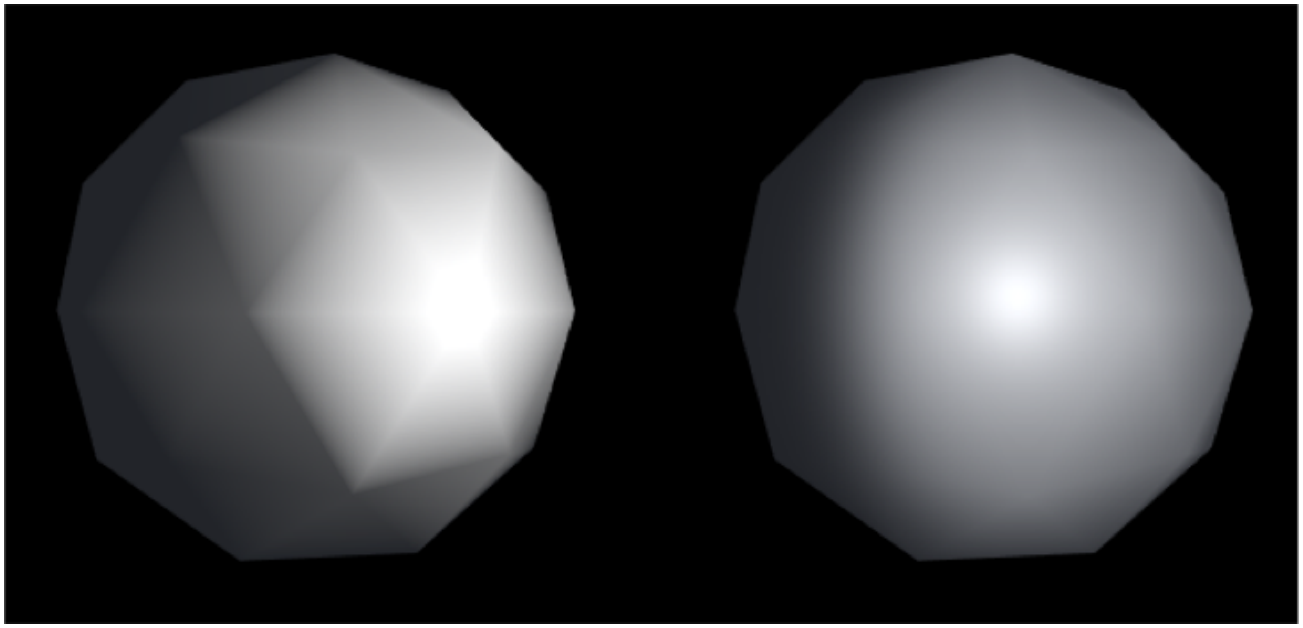


Smooth shading of polygonal objects

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 even 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 one order of magnitude.

Smooth shading of polygonal objects

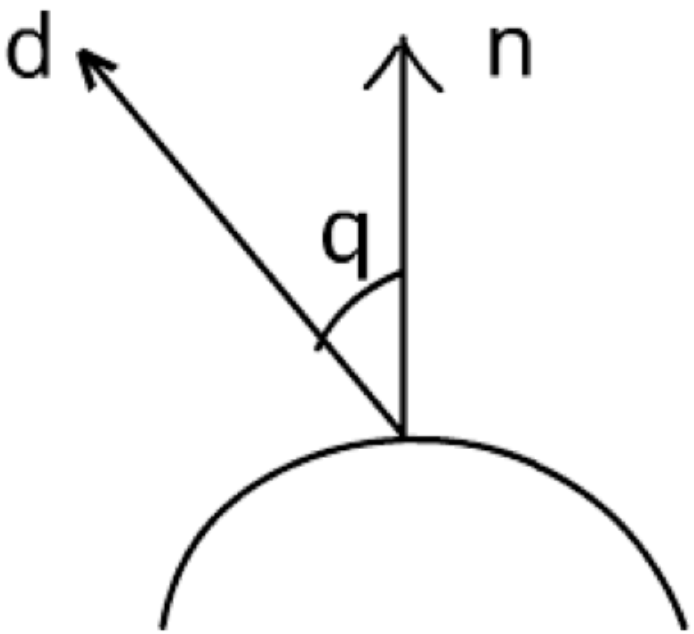
The two most common smooth shading techniques are:

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

Smoothing occurs by blending the colors generated, or the parameters passed to the shaders involved in approximating the rendering equations.

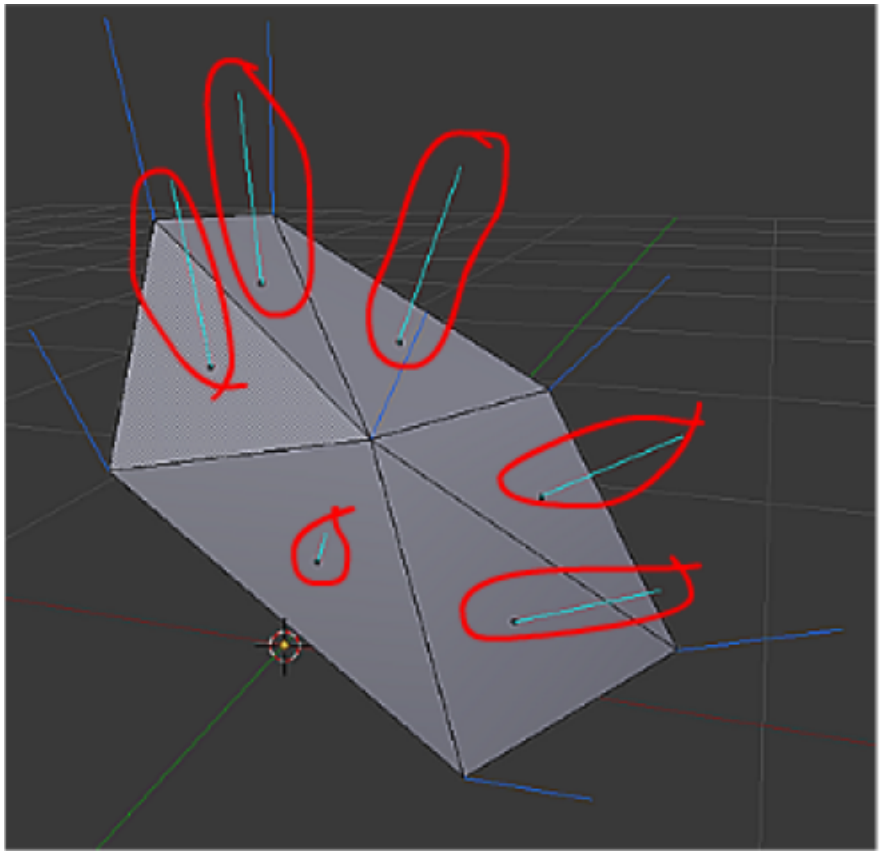
Vertex normal vectors

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.



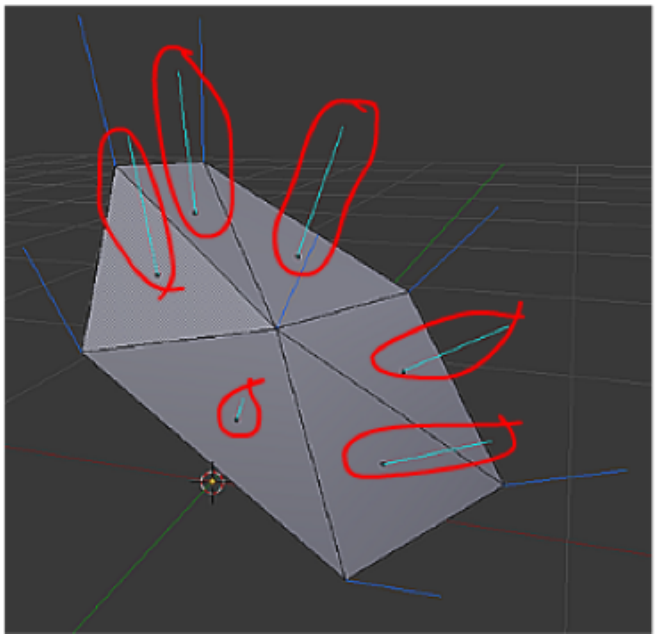
Vertex normal vectors

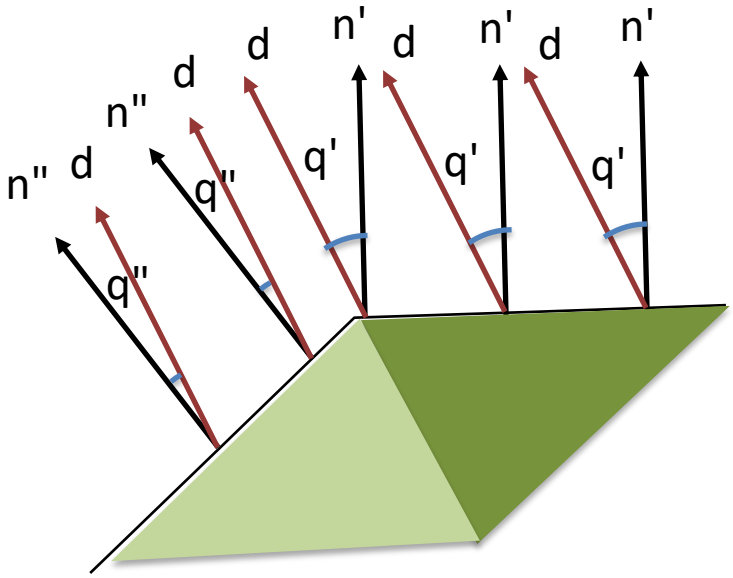
Meshes are composed by triangles and each a triangle has a single normal vector that is identical in all of its points.



Vertex normal vectors

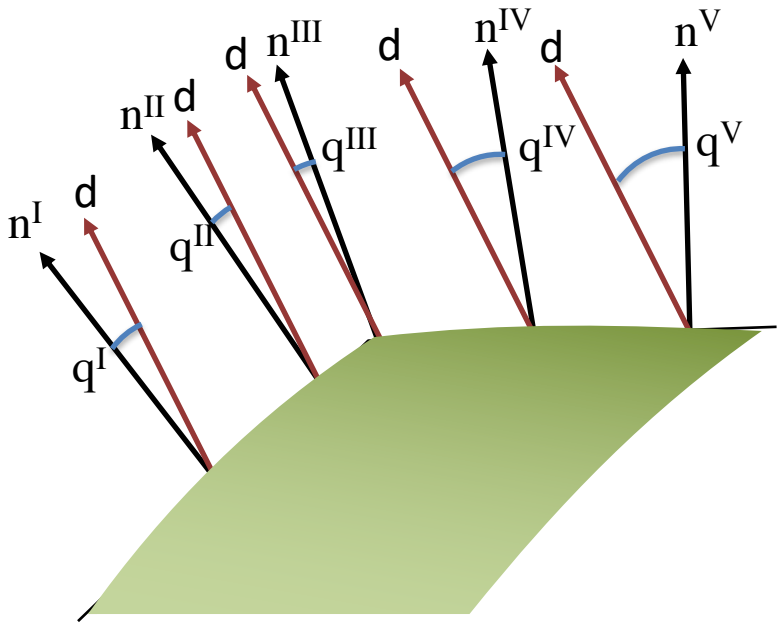
The rendering equation is very sensitive to the direction of the normal vector: the abrupt changes in the colors of the pixels across the edges of two adjacent triangles are caused by the discontinuities in the direction of such vectors.





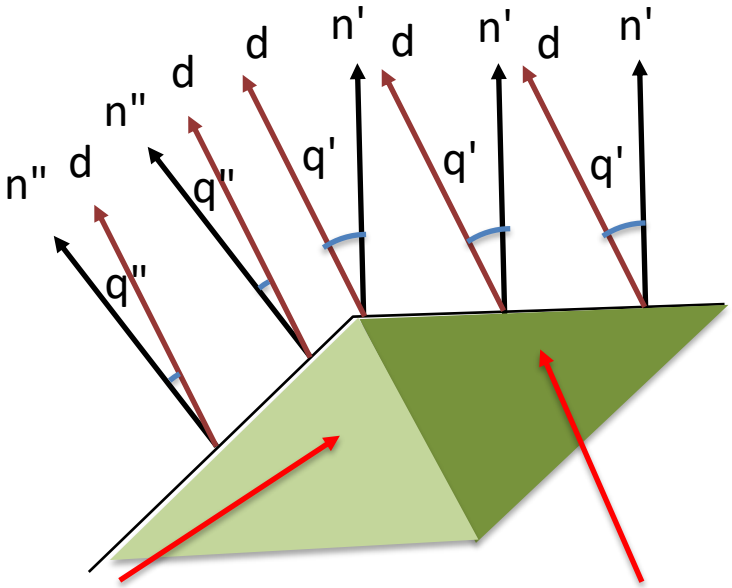
Vertex normal vectors

In a curved surface, the normal changes continuously, creating a different color in every point that makes it appear smooth.



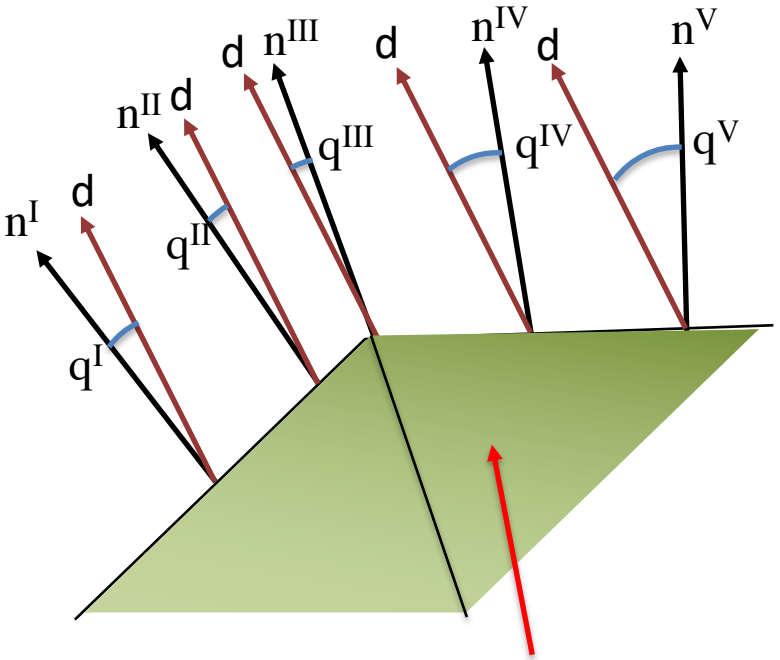
Vertex normal vectors

The basis of smooth rendering is to “fake” the geometrical normal vector, and use an artificial one that changes smoothly over the surface.



Normal n”,
Angle q”

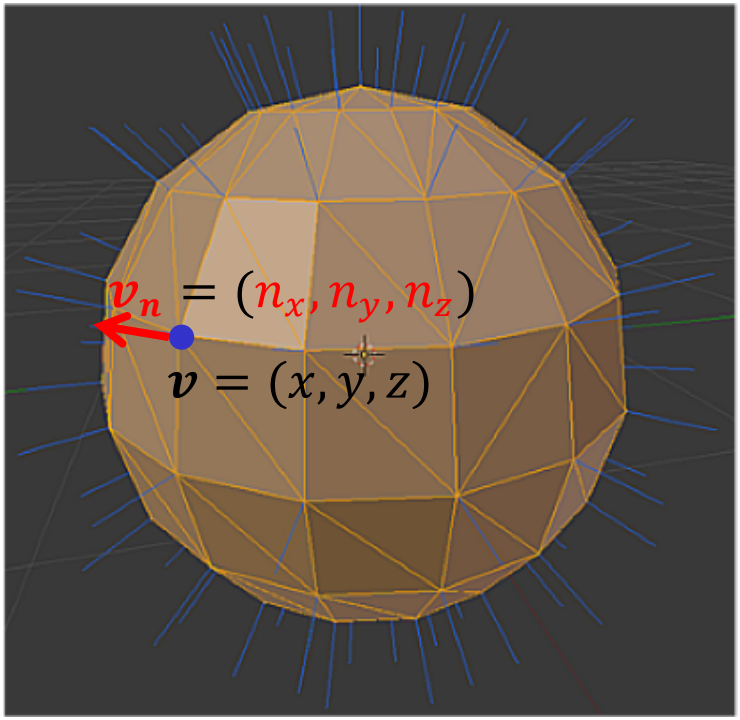
Normal n’,
Angle q’



Each point has a different “fake” normal vector, and a different angle

Vertex normal vectors

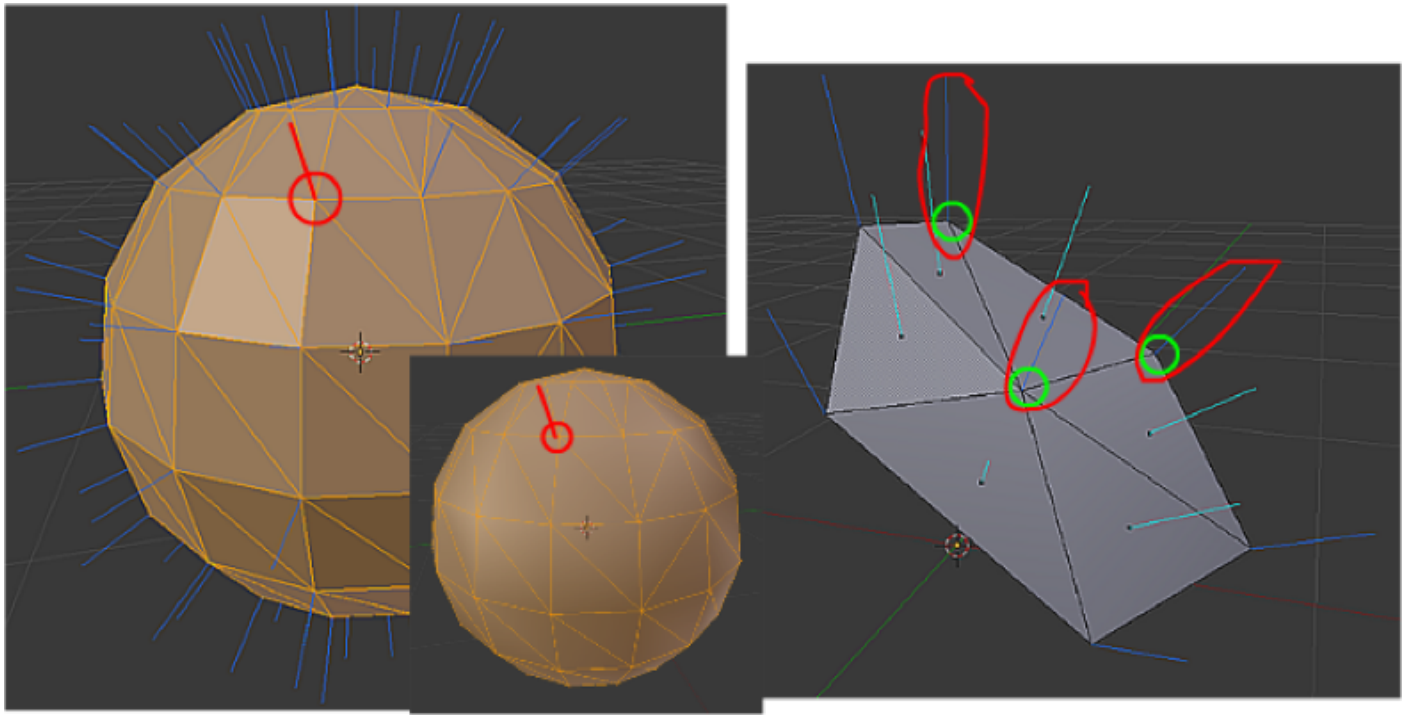
To support fake normal vectors, the encoding of a vertex is extended to 6 values, which define both the position and the direction of the normal vector to the surface for each *x*, *y* and *z* direction.



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

Vertex normal vectors

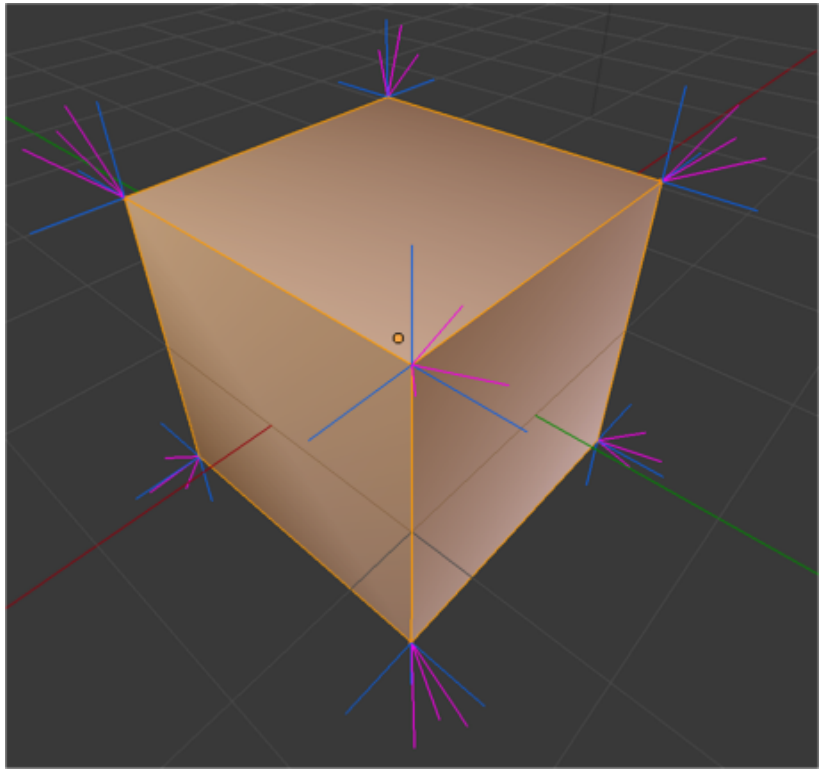
Storing the normal vector direction, together with the vertex position, allows the user to freely define it, and to control whether there is an abrupt change between adjacent triangles or not.



Vertex normal vectors

Note that the stored normal vector might be different from the geometric one determined by the triangle to which that vertex 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.

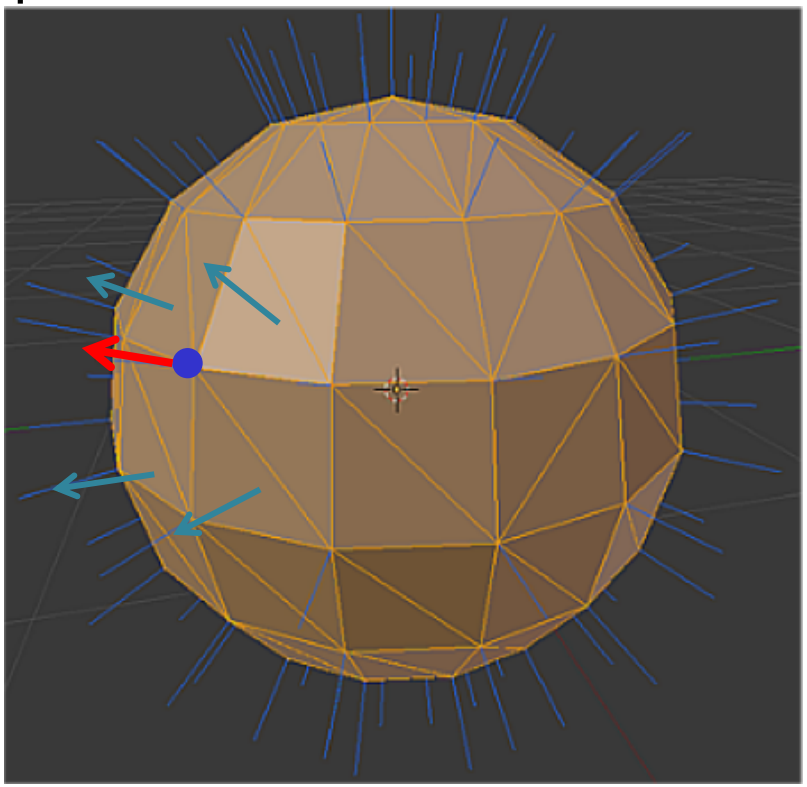


Blue: real normals

Purple: fake normals

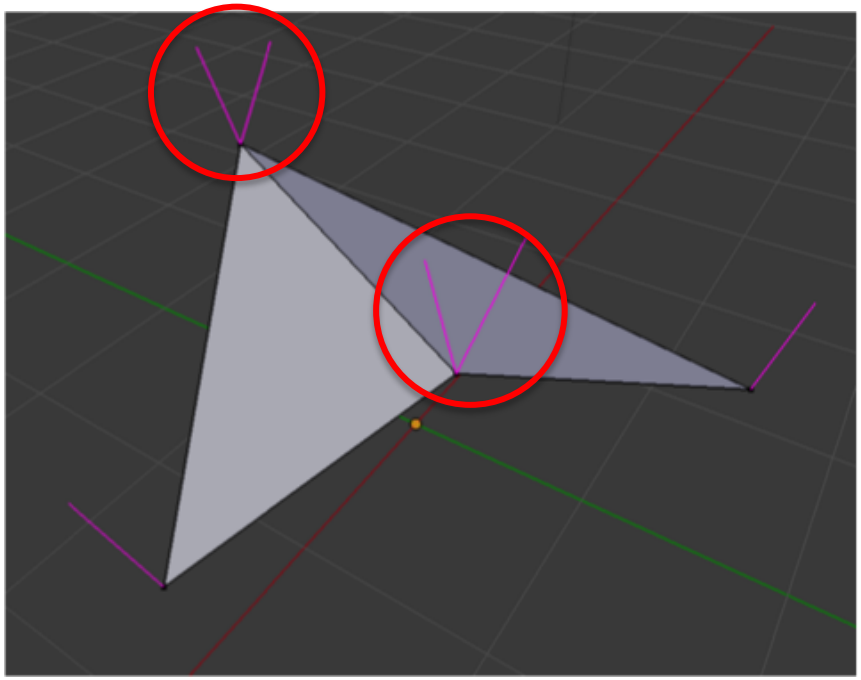
Vertex normal vectors

The normal vector associated with a vertex is usually oriented to account for the curvature of the considered surface in the corresponding point.



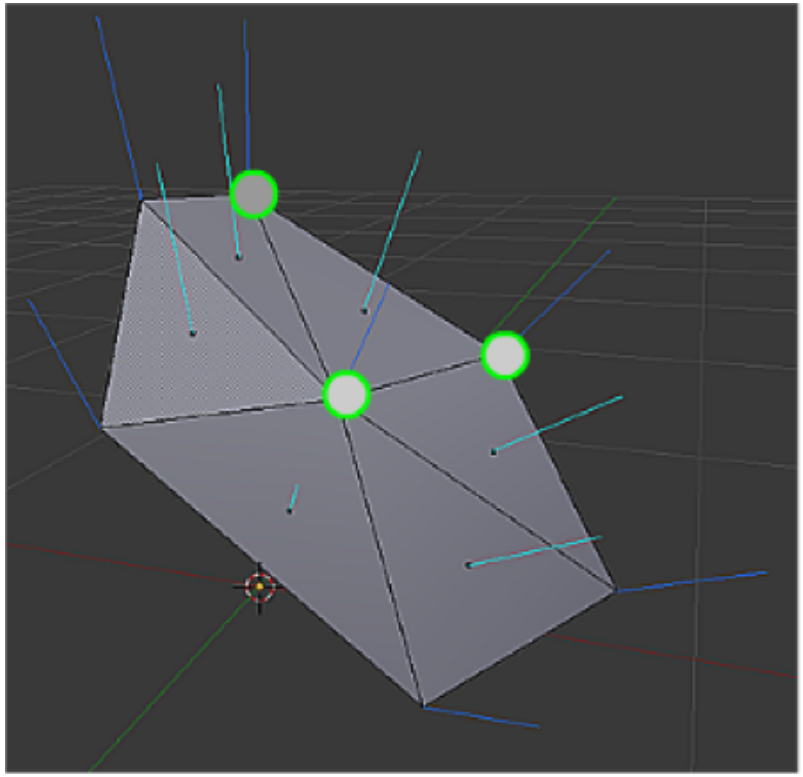
Vertex normal vectors

Note also that the normal vector direction is a property of a triangle, and not of a vertex: two triangles might have a vertex in the same spatial position, but characterized by a different normal vector direction.



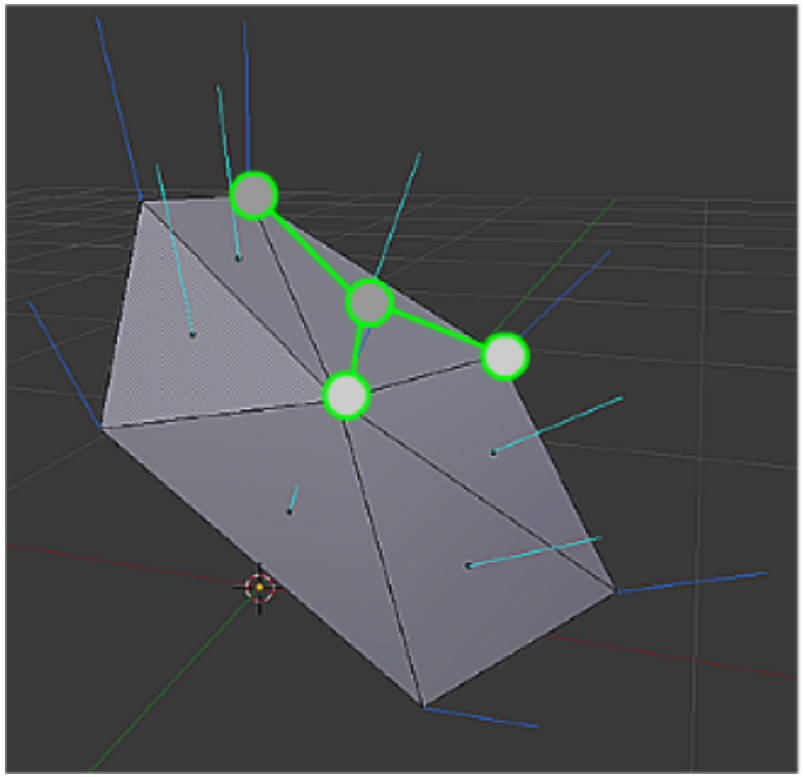
Vertex normal vectors

When considering a mesh surface, the rendering equation can determine the colors for the pixels of the object in the three vertices of a triangle, according to the associated normal vectors.



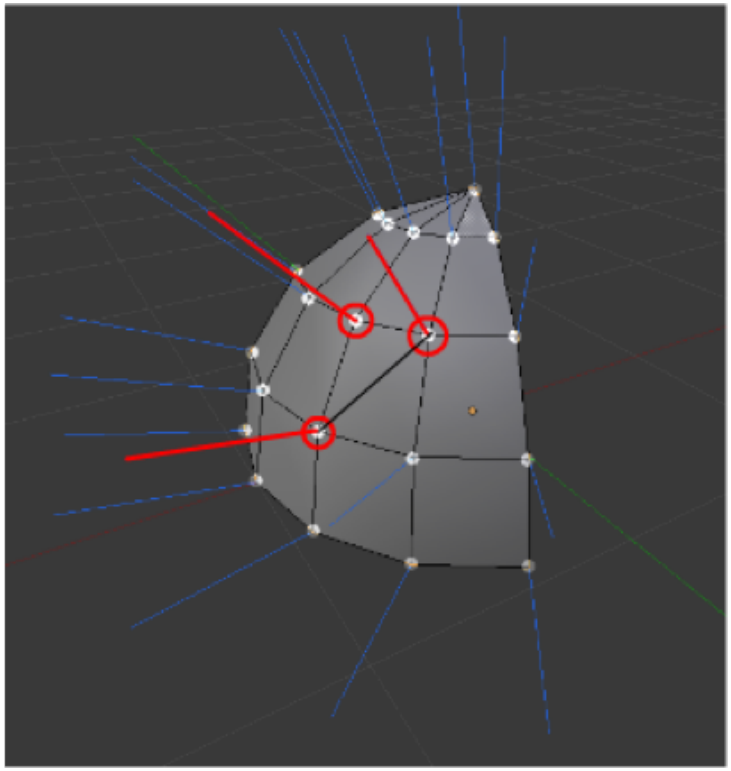
Vertex normal vectors

The colors of the internal pixels of the triangles can then be computed using different interpolation techniques.



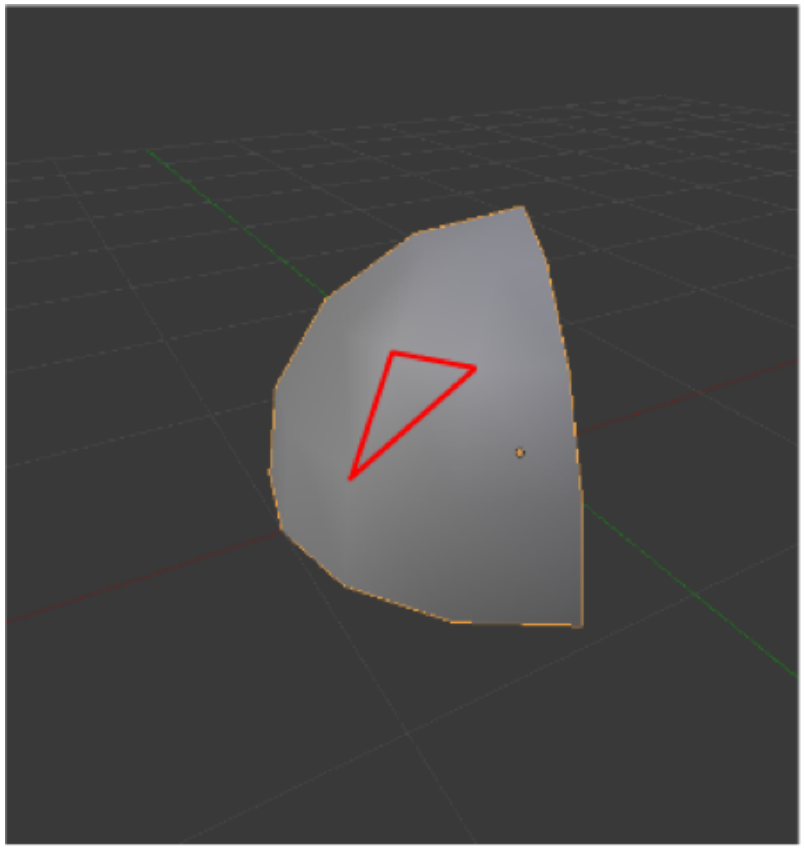
Gouraud shading

The Gouraud (per vertex) shading technique uses the lights and reflections models to compute the color of each vertex.



Gouraud shading

Then, the colors of the inner pixels of a triangle are determined by interpolation from the vertex colors.

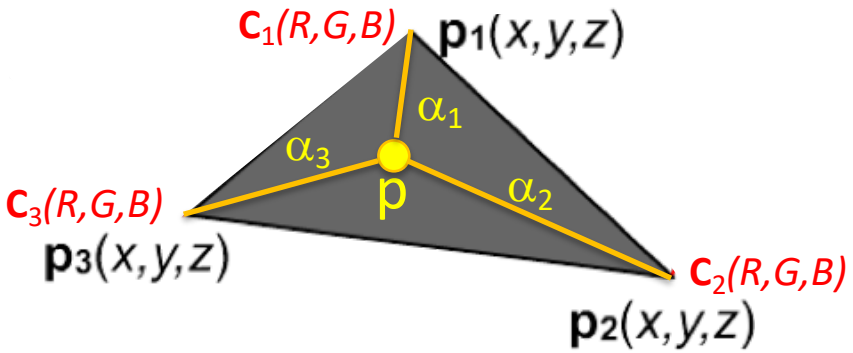


Gouraud shading

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 for the position, 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.



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

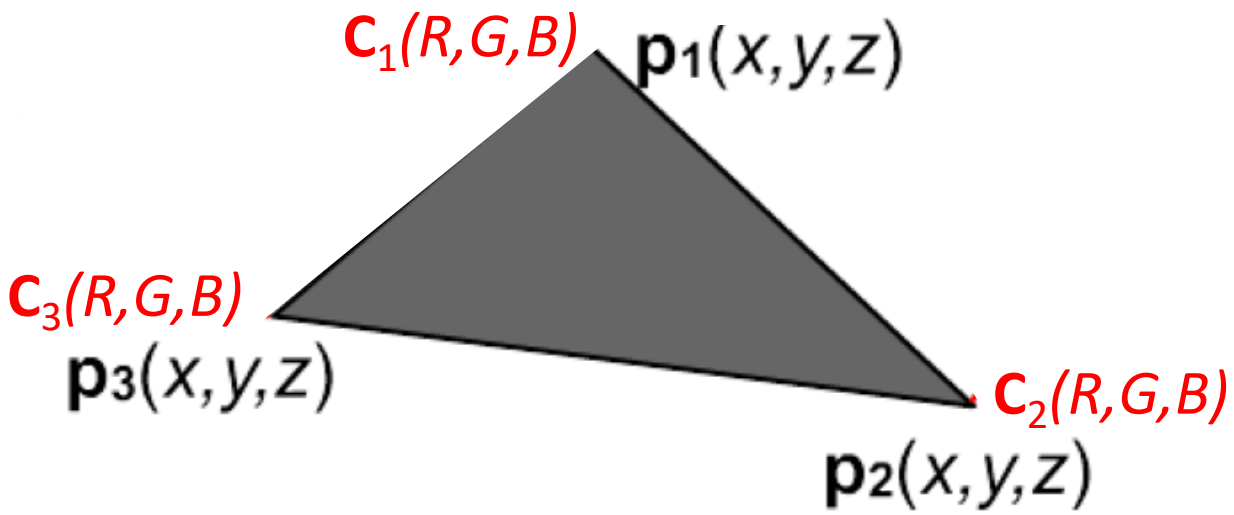
Gouraud shading

For objects that are static 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

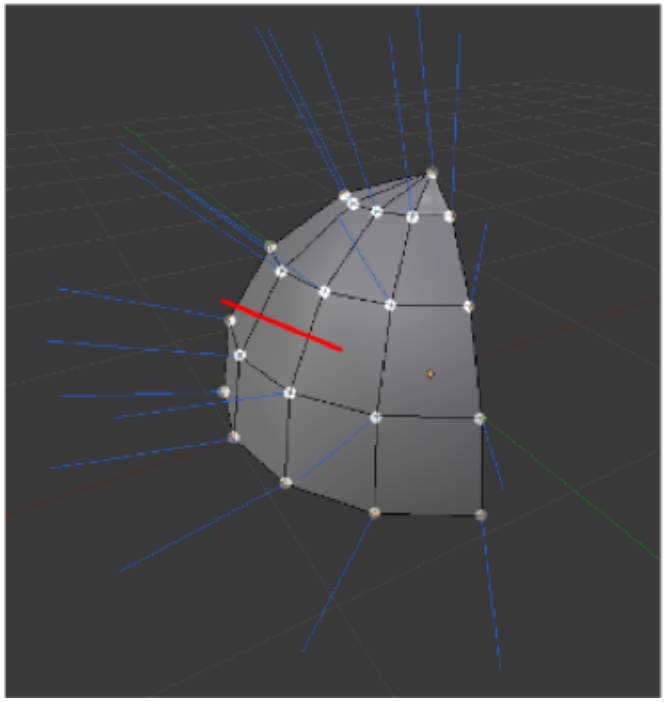
Color pre-computation is usually done in 3D authoring software such as Blender. Moreover, vertex color usually occupy less memory than normal vector directions (i.e. *4 vs. 12 bytes*).



Phong shading

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

The *vertex normal vectors* are interpolated to approximate the normal vector direction to the actual surface in the internal points of a triangle.

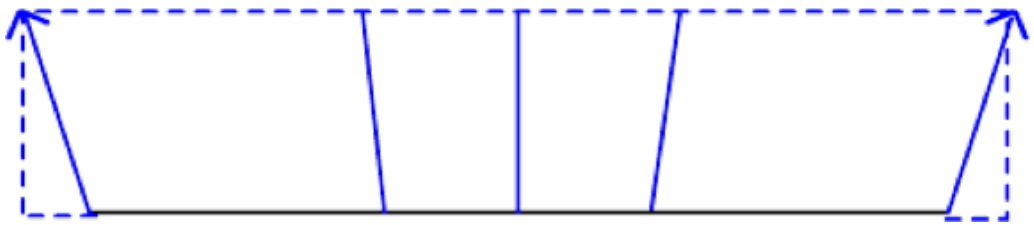


$$\alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 = p$$
$$n = \alpha_1 n_1 + \alpha_2 n_2 + \alpha_3 n_3$$

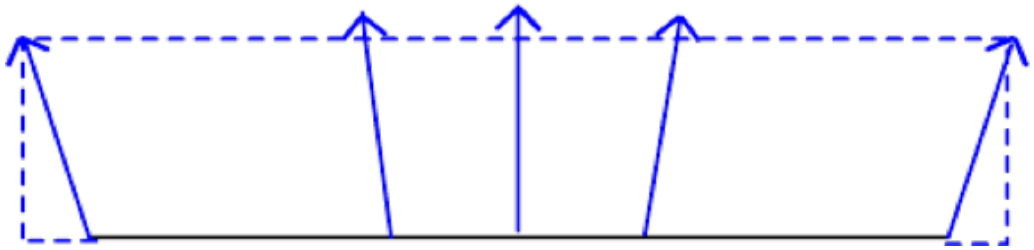
Phong shading

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

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

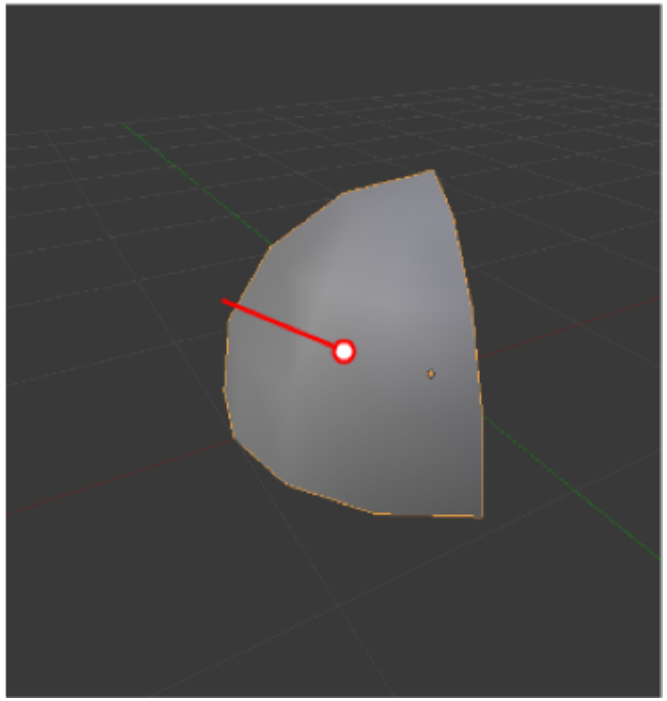


For this reason normal vectors should be normalized at every step to restore their unitary size.



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 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.

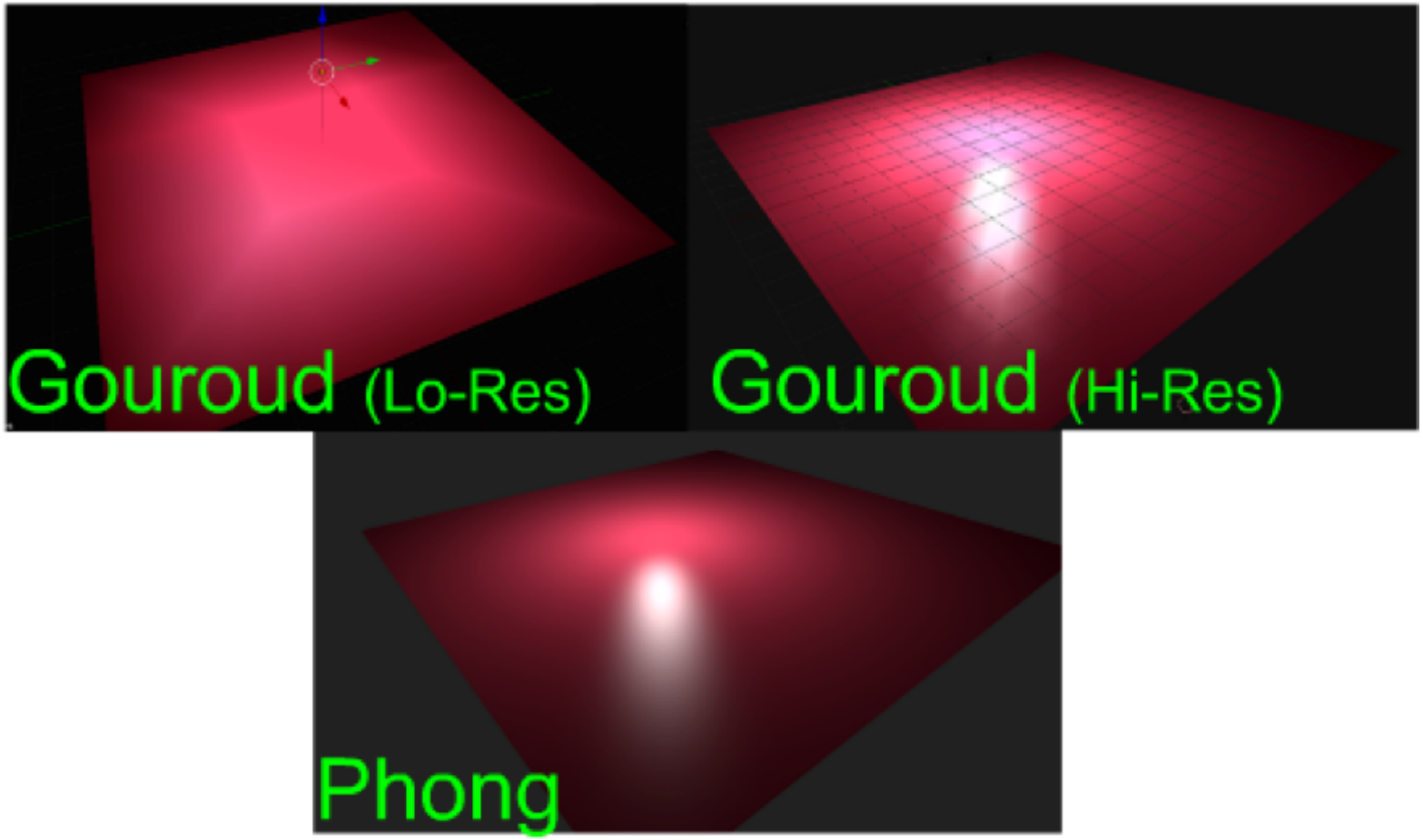
Phong shading

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.

In this case however, the area of each triangle is just a few pixel wide, making the number of vertex shown on screen comparable to the total number of pixels.

Phong shading

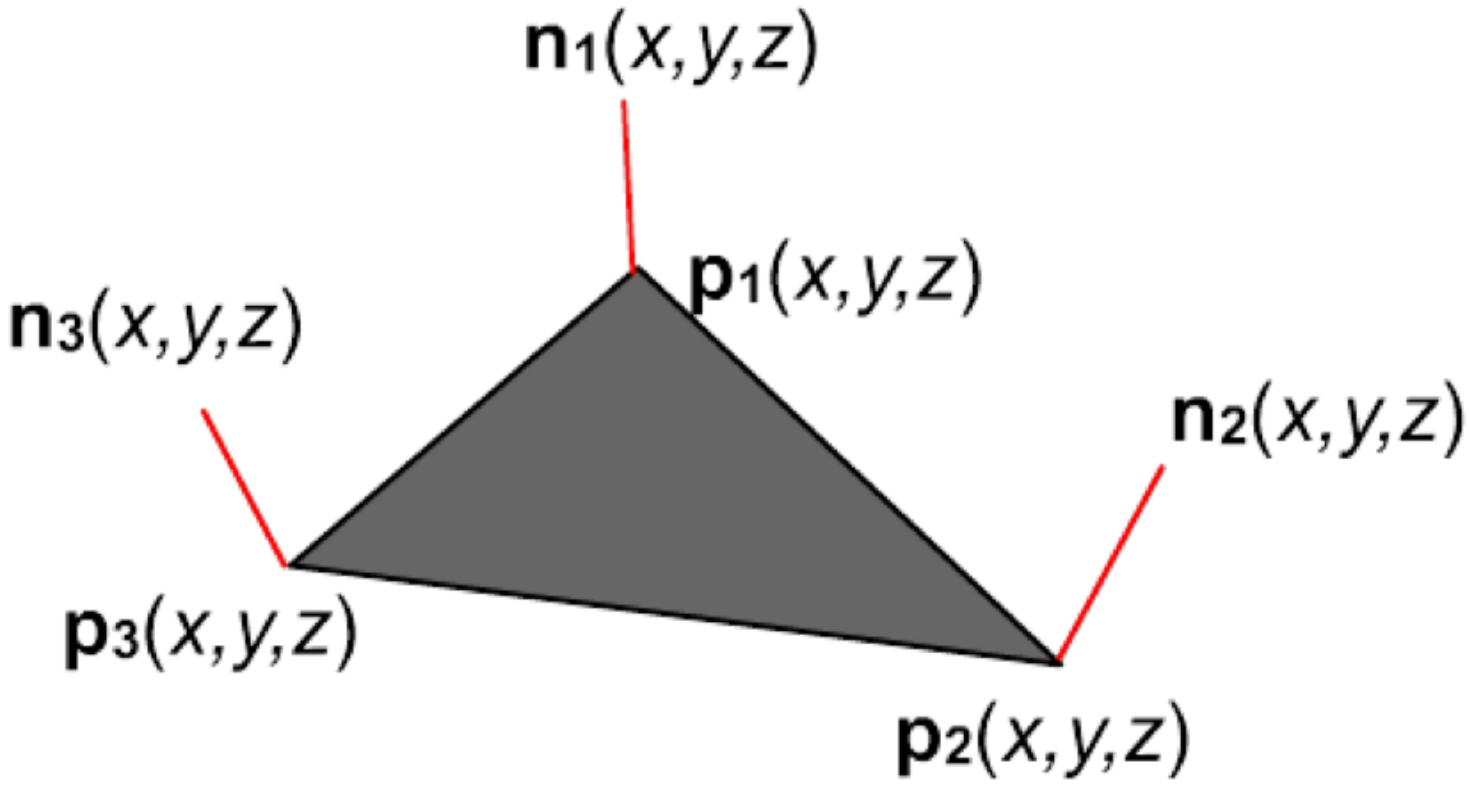


Phong shading

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

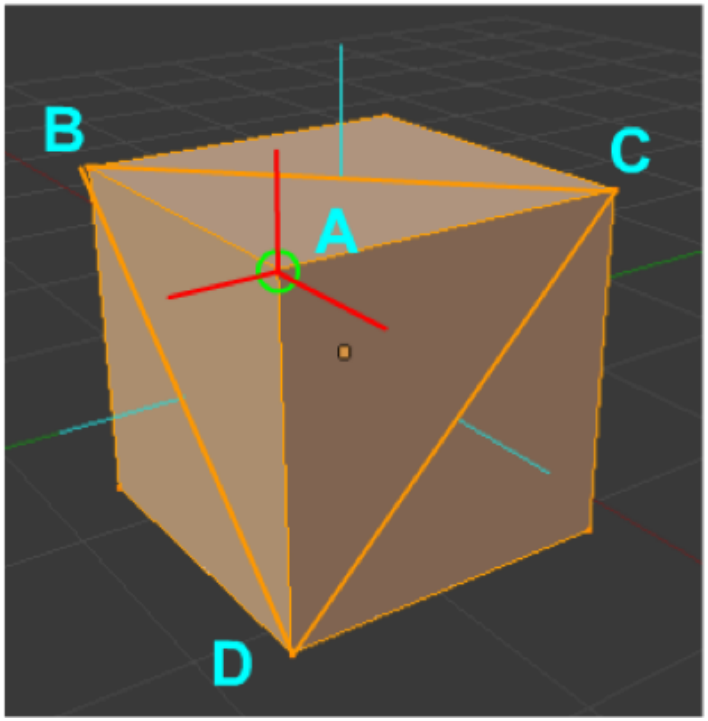
Vertex normal vectors: considerations

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



Vertex normal vectors: considerations

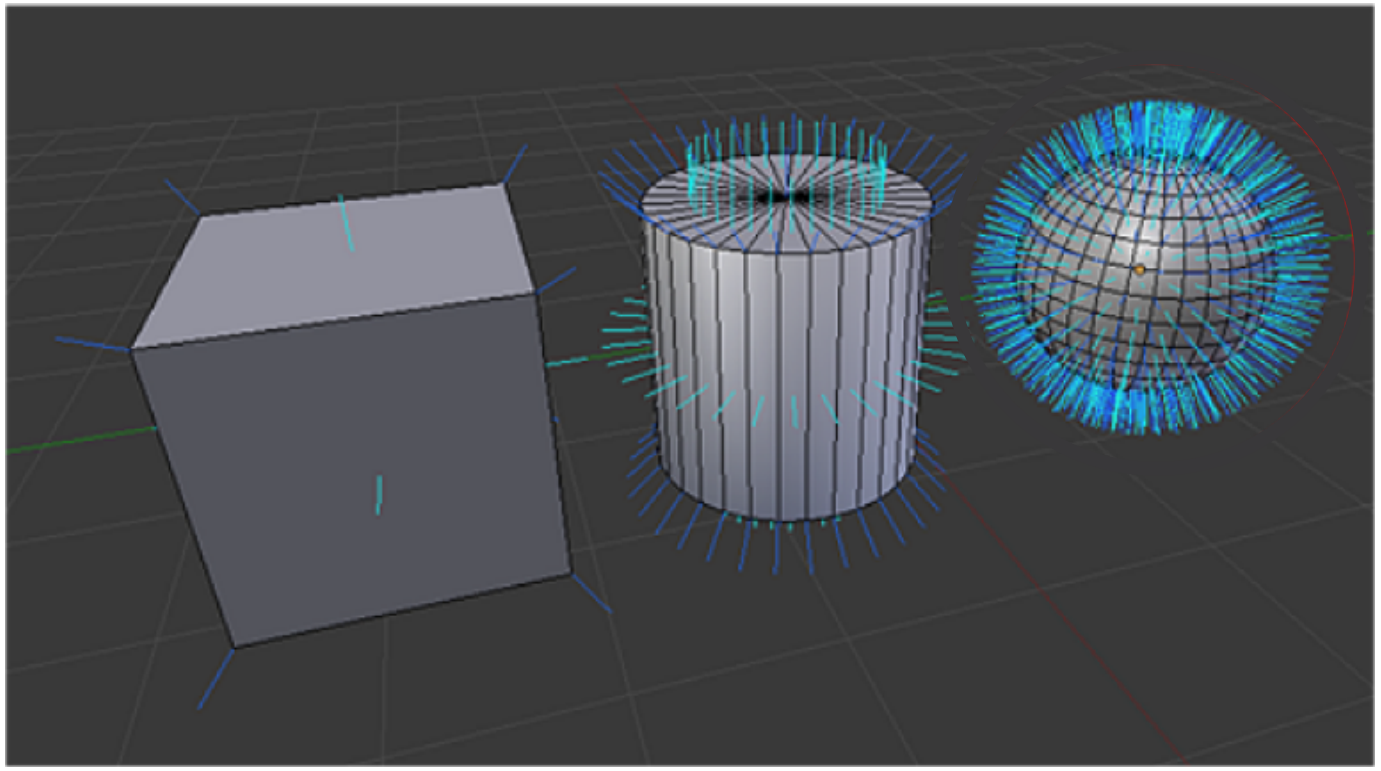
Even in simple figures such as a cube, there could be several different 6-tuples that have identical position, but different normal vectors.



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

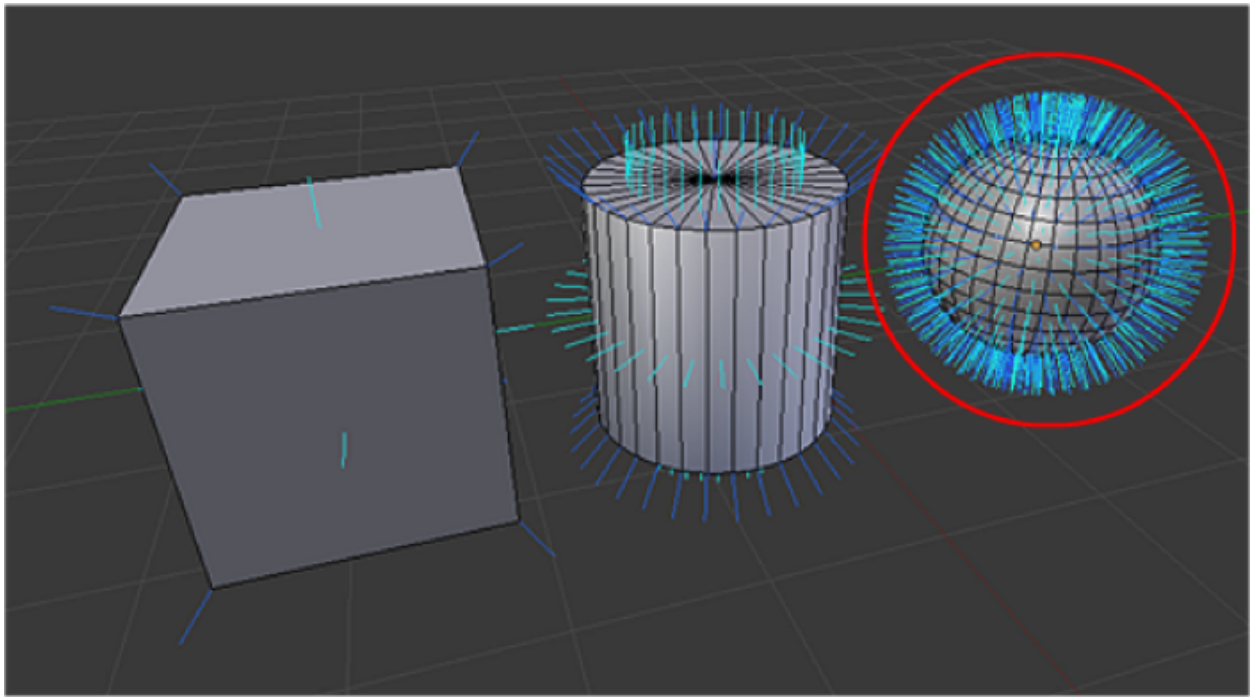
Vertex normal vectors: considerations

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



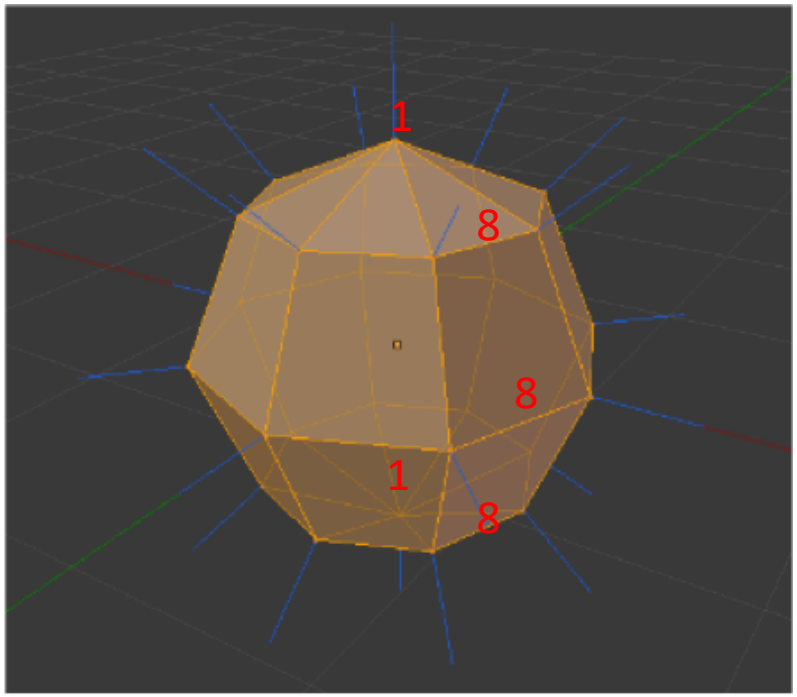
Vertex normal vectors: considerations

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.



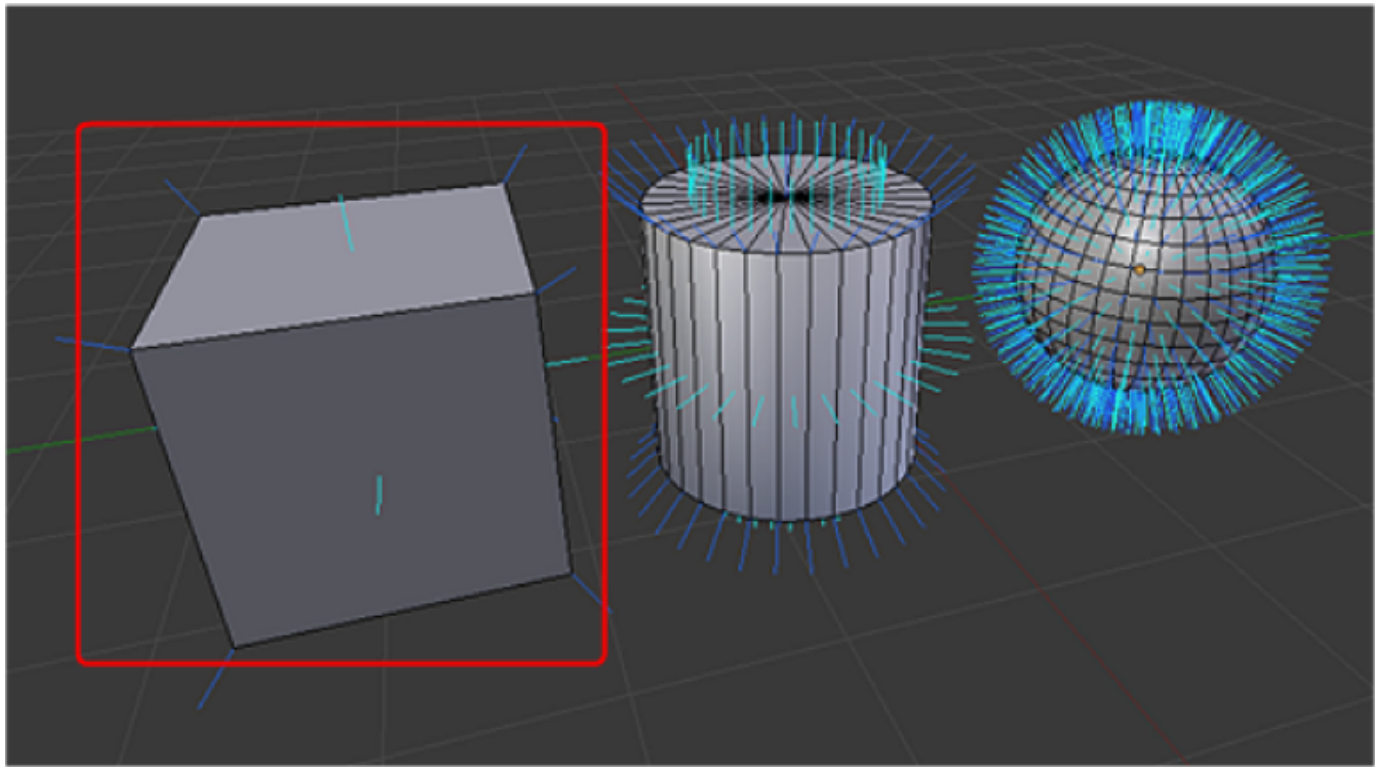
Vertex normal vectors: considerations

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.



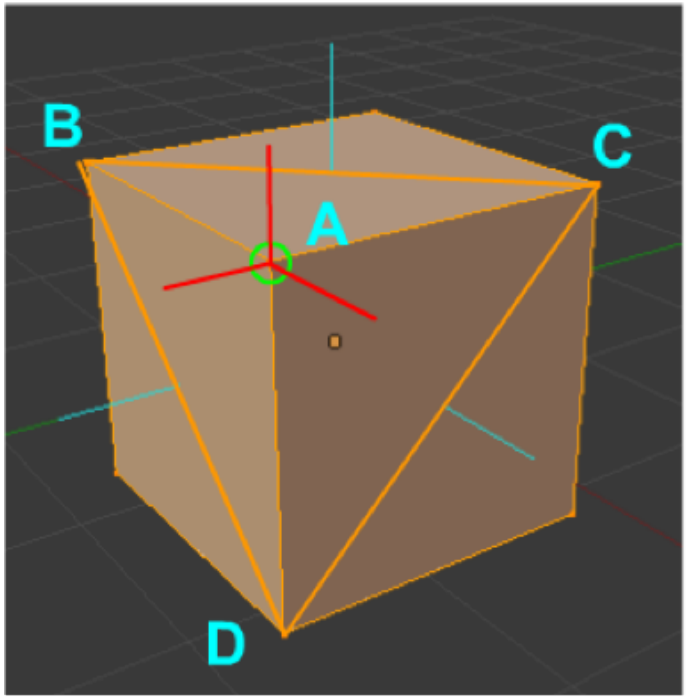
Vertex normal vectors: considerations

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



Vertex normal vectors: considerations

In this case, however, 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, vertex in position A corresponds to three different vertices: A, A' and A".



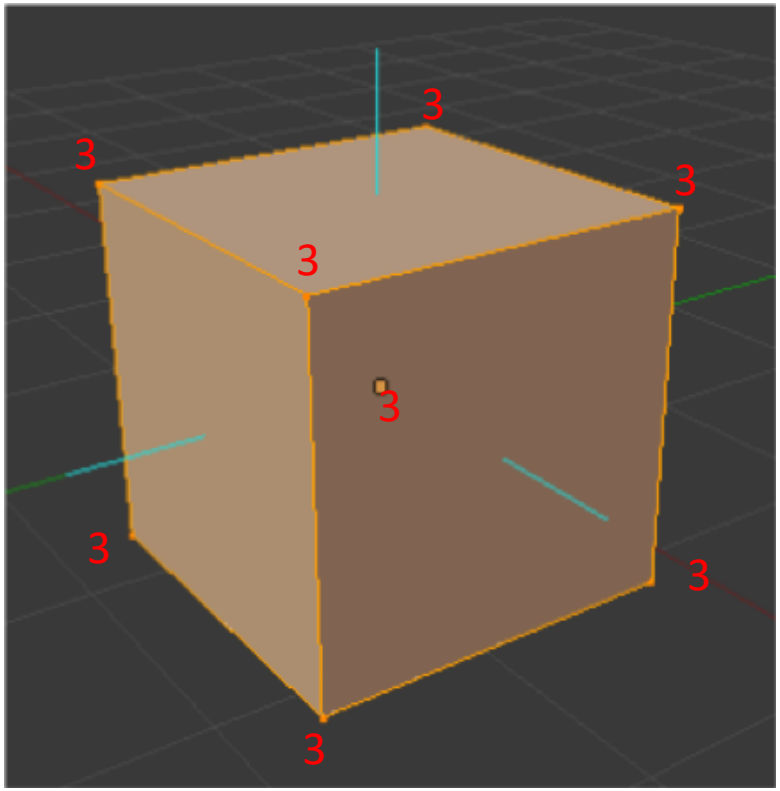
Vertex **A** in *ABC*:
$\mathbf{p}_A(1,1,1)$ $\mathbf{n}_A(0,1,0)$

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

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

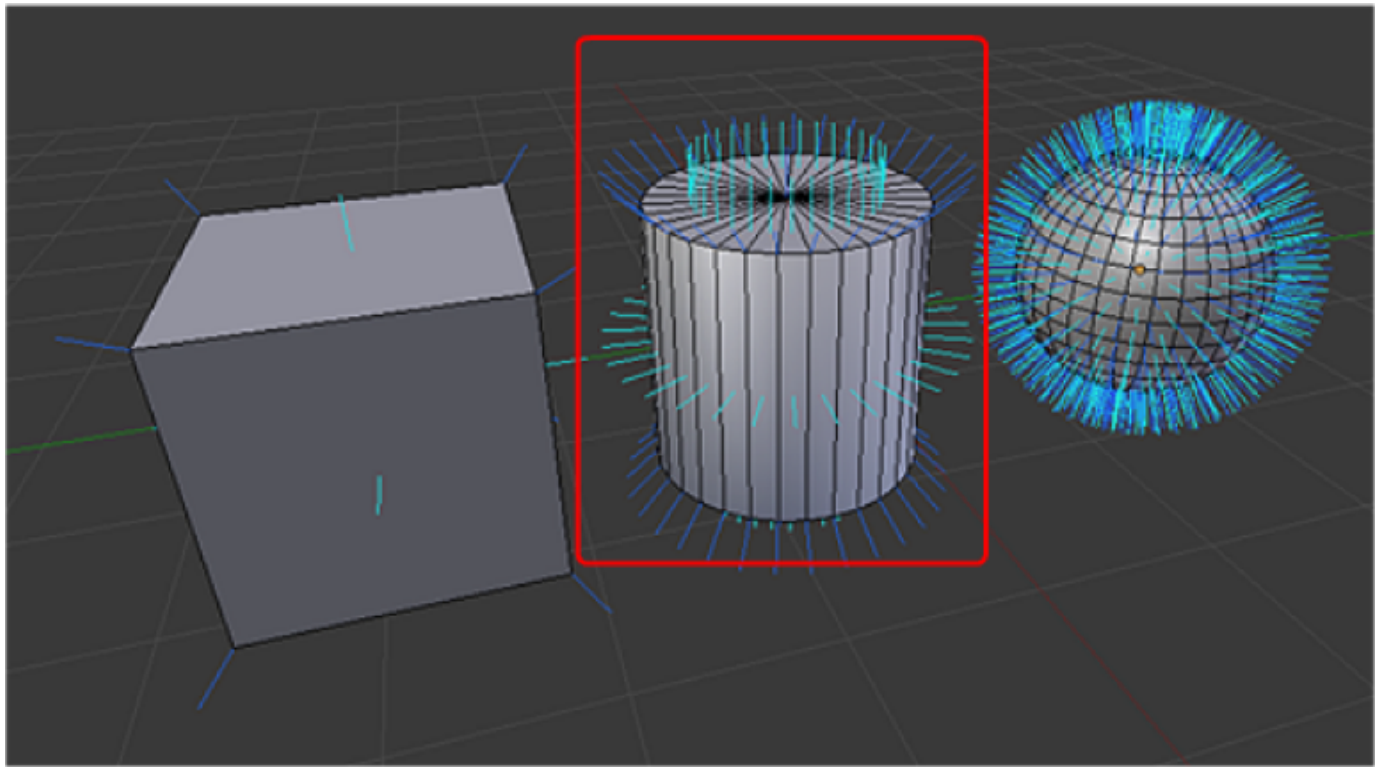
Vertex normal vectors: considerations

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



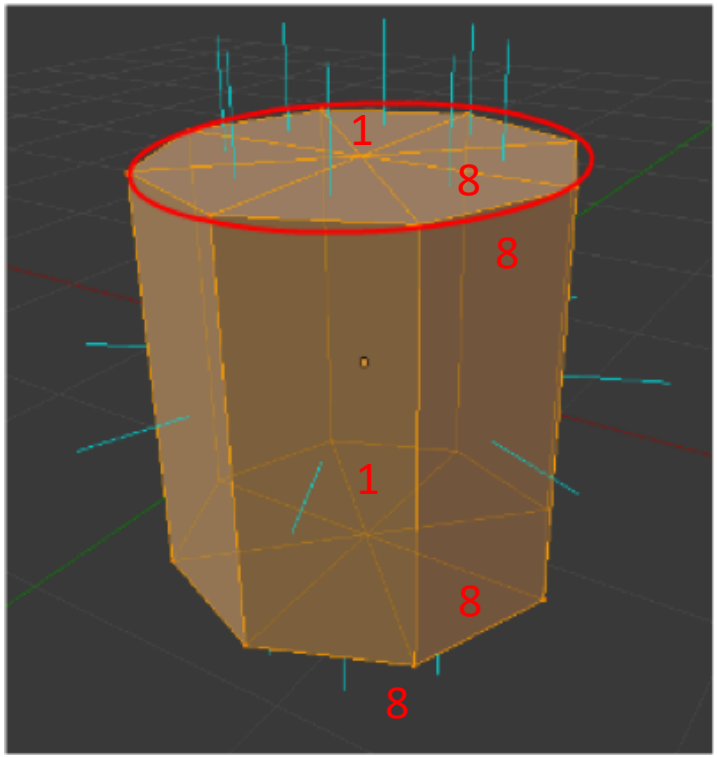
Vertex normal vectors: considerations

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

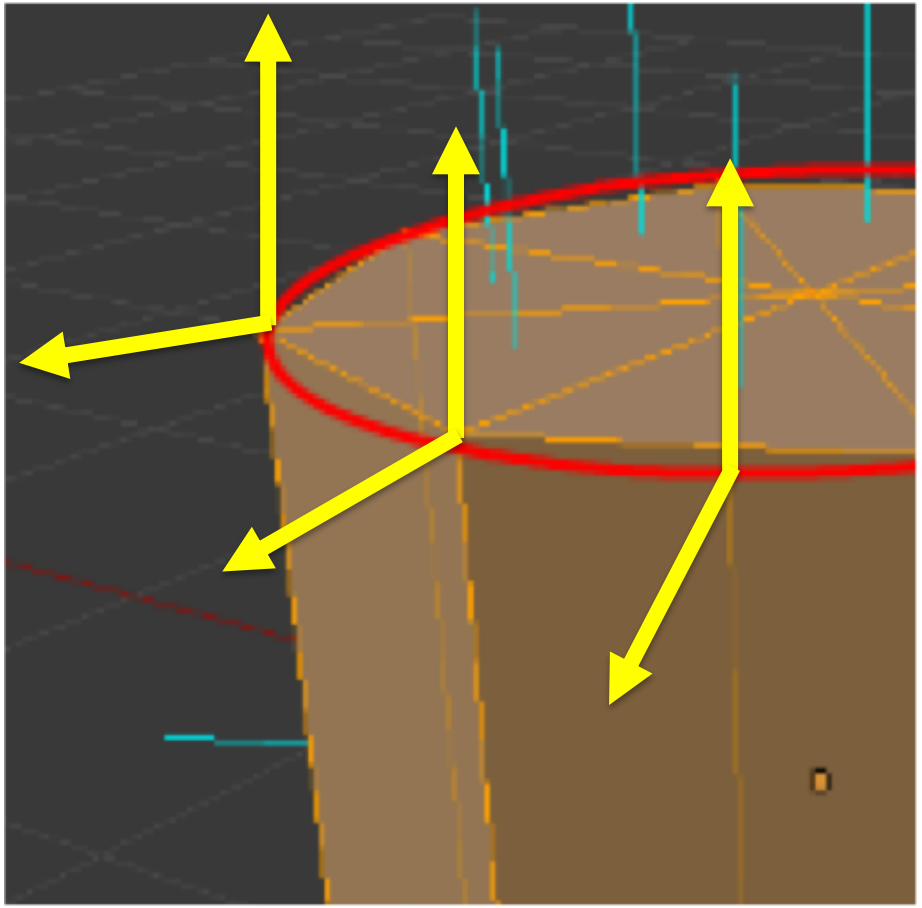
A cylinder approximated by *8* slices is defined by *32* triangles (*8* *(top cap)*+*16* *(side)*+8 (bottom cap)) and *34* vertices (*1* *(top cap center)*+*8* *(top cap ring)* + *8* *(upper side ring)*+*8* *(bottom side ring)* +*8* *(bottom cap ring)*+*1* *(bottom cap center)*) since the vertices on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



Vertex normal vectors: considerations

Note that in this case, for the vertices on the ring:

- The one belonging to the cap, will have normal oriented along the *y-axis*.
- The one belonging to the side, will be directed outside, along the radius of the cap, with they *y-axis* component equal to zero.



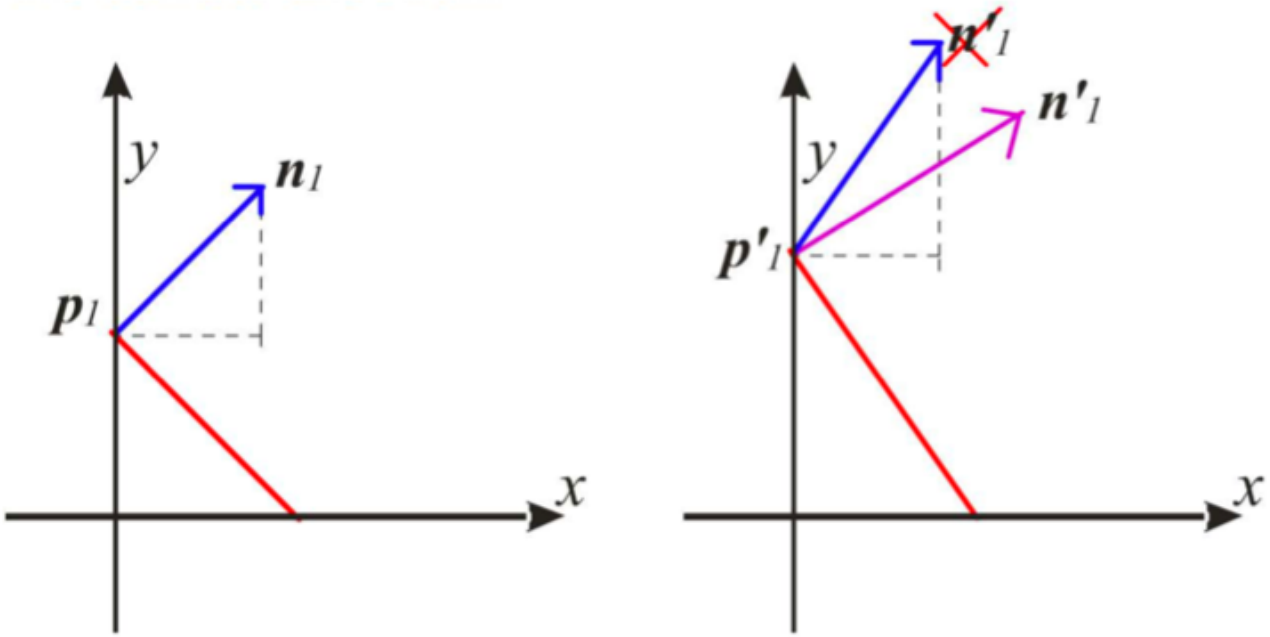
Transforming vertex normals

If the normal vectors are stored with the object, they should follow it along its World and View transform.

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

Transforming vertex normals

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.



Transforming vertex normals

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*.

$$M = \left|\begin{array}{ccc:c} & M_l & & \begin{matrix} d_x \\ d_y \\ d_z \end{matrix} \\ \hdashline 0 & 0 & 0 & 1 \end{array}\right| > M_n = \left|\begin{array}{c} M_l^{T-1} \\ \hdashline \end{array}\right|$$

Transforming vertex normals

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:

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

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:

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

Transforming vertex normals

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:

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

since:

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

We have:

$$a' = M_l \cdot a + \mathbf{d} \quad b' = M_l \cdot b + \mathbf{d}$$

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

Transforming vertex normals

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:

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

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 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 required for per-pixel shading), the previous procedure can also be used if uniform scaling is applied.

Loading the Shaders

As we have seem, Vulkan can load the shaders from SPIR-V binary files.

We have seen how to compile them outside the application.

Here we see how to load them in the application code, and prepare them for being linked to a pipeline.

Loading the Shaders

The SPIR-V can be loaded into a byte array using conventional C++ functions. In particular, we can create a `readFile()` function that receive as input the `filename` and returns a `char` array containing it.

```
static std::vector<char> readFile(const std::string& filename) {
      std::ifstream file(filename, std::ios::ate | std::ios::binary);
      if (!file.is_open()) {
            throw std::runtime_error("failed to open file!");
      }

      size_t fileSize = (size_t) file.tellg();
      std::vector<char> buffer(fileSize);

      file.seekg(0);
      file.read(buffer.data(), fileSize);

      file.close();

      return buffer;
}
```

Loading the Shaders

The `std::ios:ate` opening mode and the `tellg()` method can be used to determine the file size.

```
static std::vector<char> readFile(const std::string& filename) {
     std::ifstream file(filename, std::ios::ate | std::ios::binary);
     if (!file.is_open()) {
          throw std::runtime_error("failed to open file!");
     }

     size_t fileSize = (size_t) file.tellg();
     std::vector<char> buffer(fileSize);

     file.seekg(0);
     file.read(buffer.data(), fileSize);

     file.close();

     return buffer;
}
```

Loading the Shaders

In this way, a properly sized buffer can be created, and used to store the binary code of the shader read from the file.

```
static std::vector<char> readFile(const std::string& filename) {
     std::ifstream file(filename, std::ios::ate | std::ios::binary);
     if (!file.is_open()) {
          throw std::runtime_error("failed to open file!");
     }

     size_t fileSize = (size_t) file.tellg();
     std::vector<char> buffer(fileSize);

     file.seekg(0);
     file.read(buffer.data(), fileSize);

     file.close();

     return buffer;
}
```

Create the Shader Module

The binary code can be used to create a *Shader Module*, the data structure Vulkan uses to access the shaders. Since we need one module per Shader, it is handy to create a procedure to perform this task.

```
VkShaderModule createShaderModule(const std::vector<char>& code) {
    VkShaderModuleCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.codeSize = code.size();
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

    VkShaderModule shaderModule;

    VkResult result = vkCreateShaderModule(device, &createInfo, nullptr,
                      &shaderModule);
    if (result != VK_SUCCESS) {
        throw std::runtime_error("failed to create shader module!");
    }

    return shaderModule;
}
```

Create the Shader Module

Shader modules handles are stored into `VkShaderModule` objects, created with the `vkCreateShaderModule()` function, after filling a `VkShaderModuleCreateInfo` structure.

```
VkShaderModule createShaderModule(const std::vector<char>& code) {
    VkShaderModuleCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.codeSize = code.size();
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

    VkShaderModule shaderModule;

    VkResult result = vkCreateShaderModule(device, &createInfo, nullptr,
                      &shaderModule);
    if (result != VK_SUCCESS) {
        throw std::runtime_error("failed to create shader module!");
    }

    return shaderModule;
}
```

Create the Shader Module

The function requires only the pointer to the binary data, and an integer specifying its size.

```
VkShaderModule createShaderModule(const std::vector<char>& code) {
     VkShaderModuleCreateInfo createInfo{};
     createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
     createInfo.codeSize = code.size();
     createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

     VkShaderModule shaderModule;

     VkResult result = vkCreateShaderModule(device, &createInfo, nullptr,
                          &shaderModule);
     if (result != VK_SUCCESS) {
          throw std::runtime_error("failed to create shader module!");
     }

     return shaderModule;
}
```

Create the Shader Module

Since SPIR-V files contain just an intermediate binary representation of the Shader programs, they are no longer necessary at the end of the pipeline creation process. For this reason they should be destroyed once the pipeline has been created using the `vkDestroyShaderModule()` function.

```
void createPipeline(…) {
        auto vertShaderCode = readFile((SHADER_PATH + VertexShaderName).c_str());
        auto fragShaderCode = readFile((SHADER_PATH + FragShaderName).c_str());

        VkShaderModule vertShaderModule =
                createShaderModule(vertShaderCode);
        VkShaderModule fragShaderModule =
                createShaderModule(fragShaderCode);

    …

    vkDestroyShaderModule(device, fragShaderModule, nullptr);
    vkDestroyShaderModule(device, vertShaderModule, nullptr);
}
```

Configuring the Shader Stages

Shaders are then used in the pipeline creation process as an array of `VkPipelineShaderStageCreateInfo` objects.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] =
     {vertShaderStageInfo, fragShaderStageInfo};
```

Configuring the Shader Stages

The `stage` field of each element defines which type of shader is contained in the module: `VK_SHADER_STAGE_VERTEX_BIT` for the *Vertex*, and `VK_SHADER_STAGE_FRAGMENT_BIT` for the *Fragment*.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] =
     {vertShaderStageInfo, fragShaderStageInfo};
```

Configuring the Shader Stages

The `module` field contain a pointer to the corresponding *Shader Module* previously created.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] =
     {vertShaderStageInfo, fragShaderStageInfo};
```

Configuring the Shader Stages

Each shader might have several entry points. The `pName` field contains the name of the procedure that must be called to perform the corresponding function. Usually, this will be `"main"`.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] =
     {vertShaderStageInfo, fragShaderStageInfo};
```