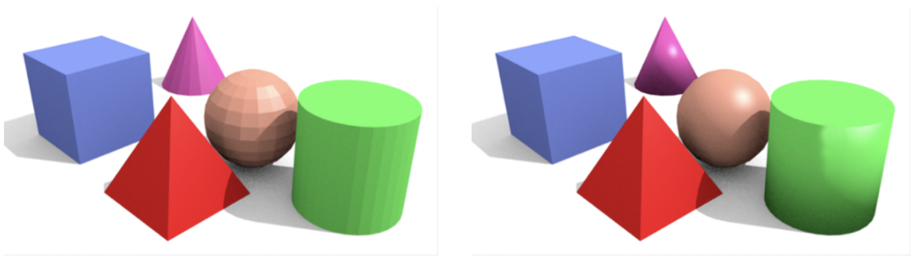
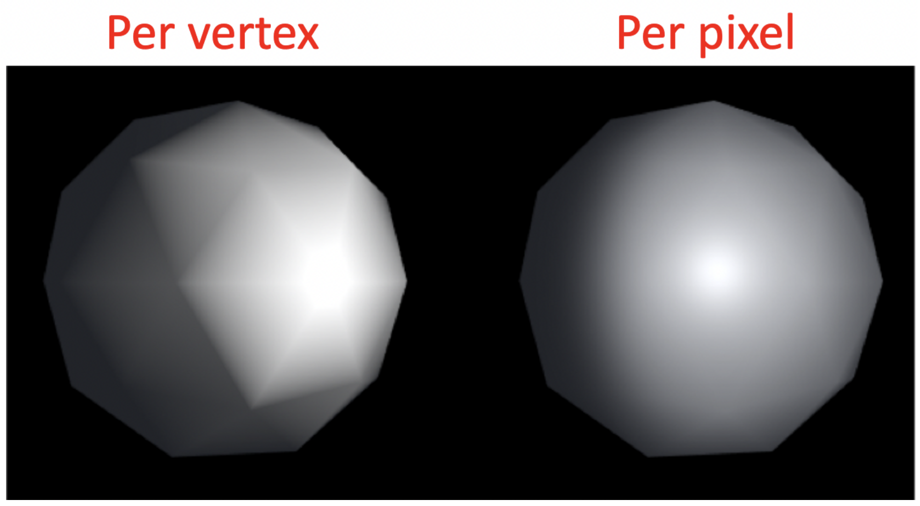
Assignment 18 (Smooth Shading)

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



The solution can be computed per-vertex or per-pixel:



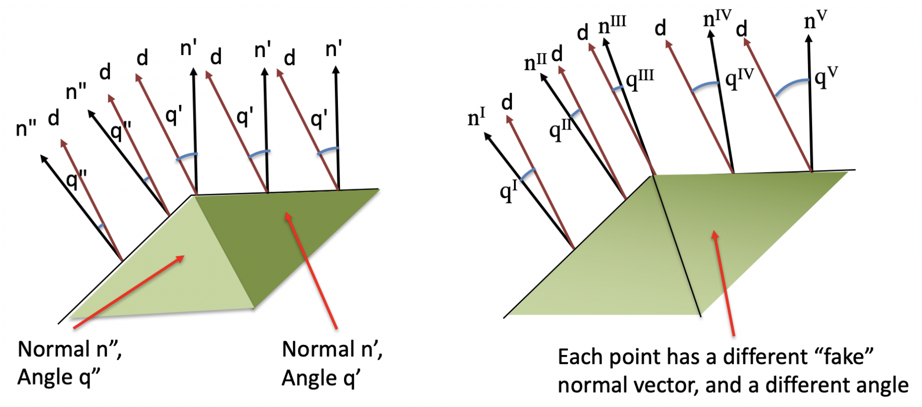
To create smoother images, the rendering equation can be even solved several times per pixel to avoid the aliasing effect. The two most common smooth shading techniques are:

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

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

In a curved surface, the normal changes continuously, creating a different color in every point that makes it appear smooth. The basis of smooth rendering is to “fake” the geometrical normal vector, and use an artificial one that changes smoothly over the surface.



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:

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

The new elements specify the direction of the normal vector in that specific point. The normal vector associated with a vertex is usually oriented to account for the curvature of the considered surface in the corresponding point.

When considering a mesh surface, the rendering equation can determine the colors for the pixels of the object. Up to now we have seen the normal vector direction as the main parameter to determine the color. We can also compute the solution of the rendering equation only for the points that corresponds to the vertices, in which we have the direction of the normal. What do we do for the internal part of the triangles? We can interpolate the exact color computed for the vertices of the triangles.

Goraud shading

The Gouraud shading technique uses the lights and reflections models to compute the color of each vertex. Then, the colors of the inner pixels of a triangle are determined by interpolation from the vertex colors.

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:

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

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.

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.

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

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.

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.

Considerations about 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.
- 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.
- Note: do not confuse the Phong specular model with the Phong shading technique.

More on vertex normal vectors

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

Loading the Shaders

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

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(); // tellg is used to
  std::vector<char> buffer(fileSize);            // determine the file size
  file.seekg(0);
  file.read(buffer.data(), fileSize);
  file.close();
  return buffer;
}
```

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;
    // pointer to the binary data and
  // an integer specifying its size.
  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;
}
```

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

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);
    // code
  // here's the important part
    vkDestroyShaderModule(device, fragShaderModule, nullptr);
    vkDestroyShaderModule(device, vertShaderModule, nullptr);
}
```

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

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

```
// look here
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";
// here
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";
// and also here
VkPipelineShaderStageCreateInfo shaderStages[] =
    {vertShaderStageInfo, fragShaderStageInfo};
```