# GLSL: Textures and Maps

Sunday, May 24, 2020 10:58 PM

## Samplers

Buffers of data: normals, depth, colors forming images, displacements, heights, …

Textures seem to have ***layers***.

***Immutable vs mutable storage*** textures. The size, format and layers of immutable storage textures can't change after creation.

***Samplers*** are the handler variables that point at textures in the application. Shaders typically access them using uniforms.

## Textures

The name "texture" is somewhat a misnomer. They are best described as arbitrary chunks of memory.

Diffuse textures are used to associate a vertex with a color. The color is accessed using the texture coordinate assigned to the vertex.

Some techniques use the texture color sample in the shading or reflection model.

***Sampler*** uniform variables are used to access ***texture units*** in GLSL. A ***texture unit*** is the buffer that stores a texture's data in the GPU. There's a number of them, GL_MAX_TEXTURE_UNITS.

***Objects***, ***targets***, and ***texture units***:

- glActiveTexture selects one of the ***texture units***, for example GL_TEXTURE0.
- glGenTextures creates an ***object***, referred to by the resulting descriptor;
- glBindTextures then binds the ***object*** to a ***target***, the textures target GL_TEXTURE_2D (other texture targets are GL_TEXTURE_1D, GL_TEXTURE_1D_ARRAY, GL_TEXTURE_2D, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_2D_MULTISAMPLE, GL_TEXTURE_2D_MULTISAMPLE_ARRAY, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_CUBE_MAP_ARRAY, or GL_TEXTURE_RECTANGLE);
- glTexStorage2D then sets the ***storage format*** (the pixel encoding, like RGBA, that specifies how the data is to be interpreted) and ***size*** of the data;
- glTexImage2D or glTexSubimage2D then load the data from the ***CPU*** to the texture unit in the ***GPU***.

When creating the uniform variable for the texture, you specify the texture unit number.

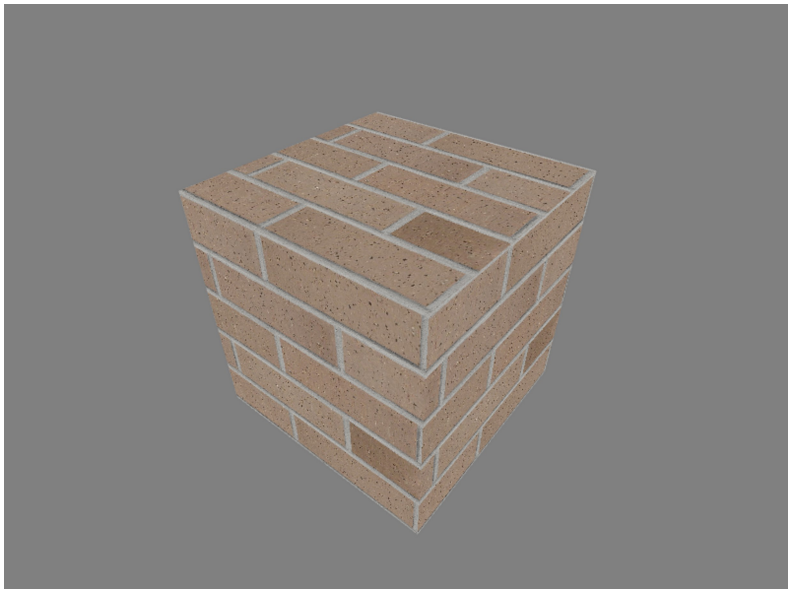
When the polygon is larger than the texture, OpenGL ***magnifies*** it using the ***resampling filter*** that you specify. The magnification filter produces more sample points between every 4 neighboring texels of the original texture. The extra samples are the result of a linear interpolation between the 4 texels:

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

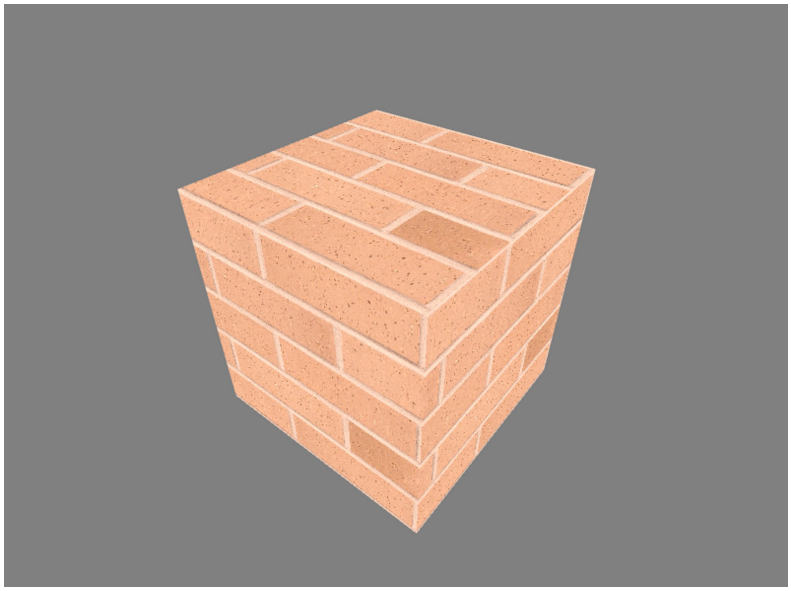
When the polygon is smaller than the texture, OpenGL ***minifies*** it using the resampling filter that you specify.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

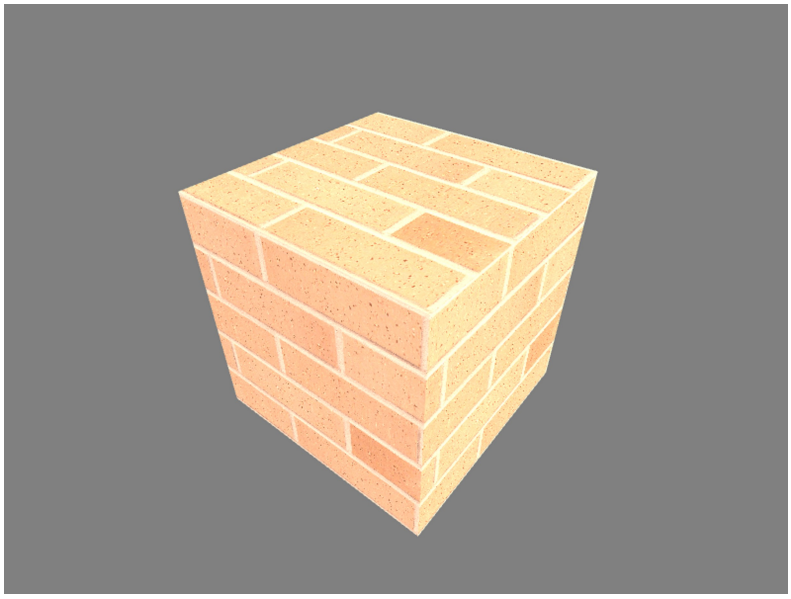
Base color



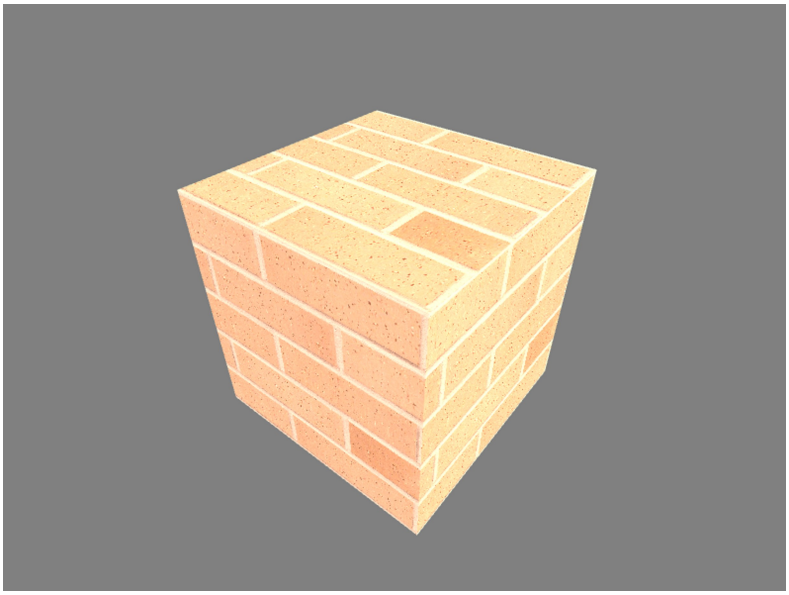
Base color + ambient



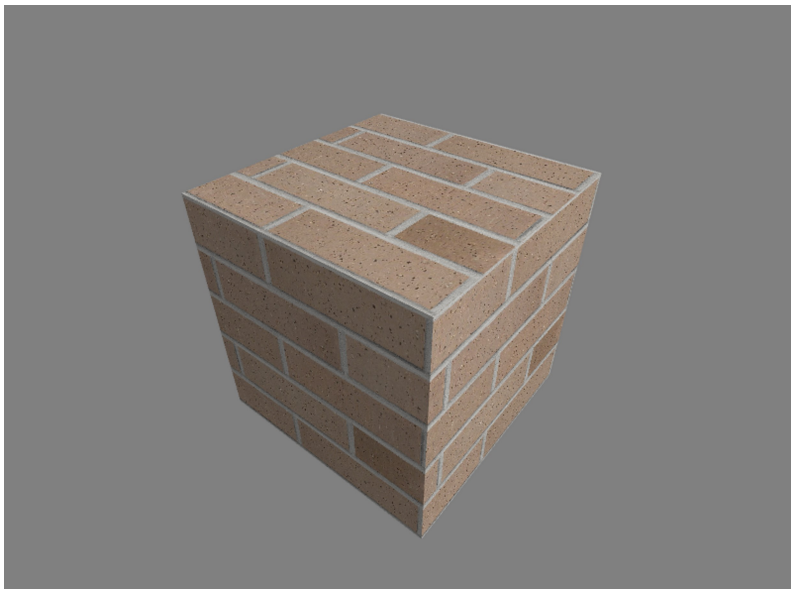
Base color + ambient + diffuse



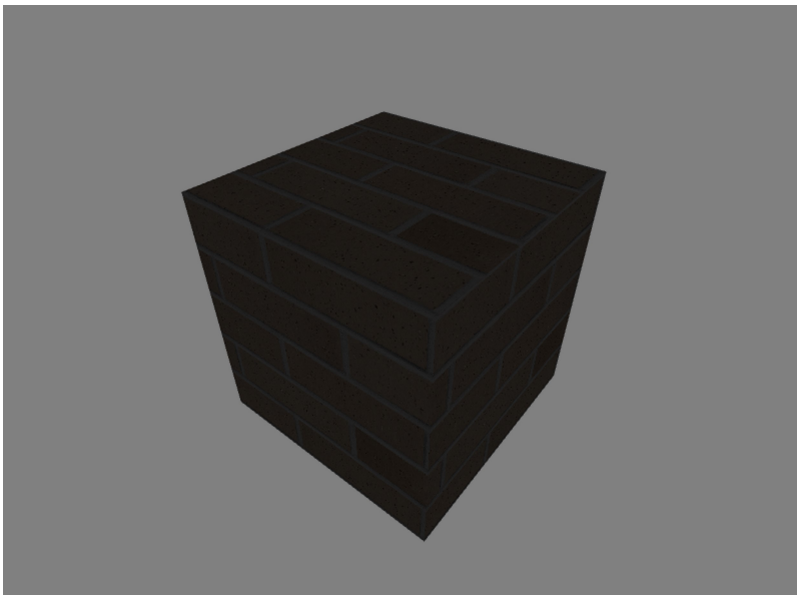
Base color + ambient + diffuse + specular



Using base color as Ka, Kd, and Ks; the result is closer to the book's



The light was inside the cube



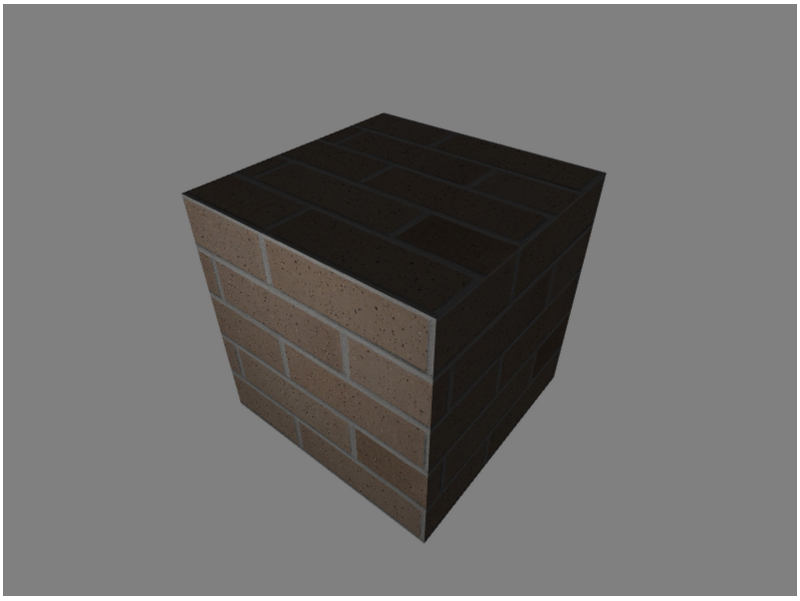
I liked this:

**IMPORTANT:** I used the diffuse coefficient Kd, but when you are sampling the color from a diffuse texture, that is your diffuse reflectivity.

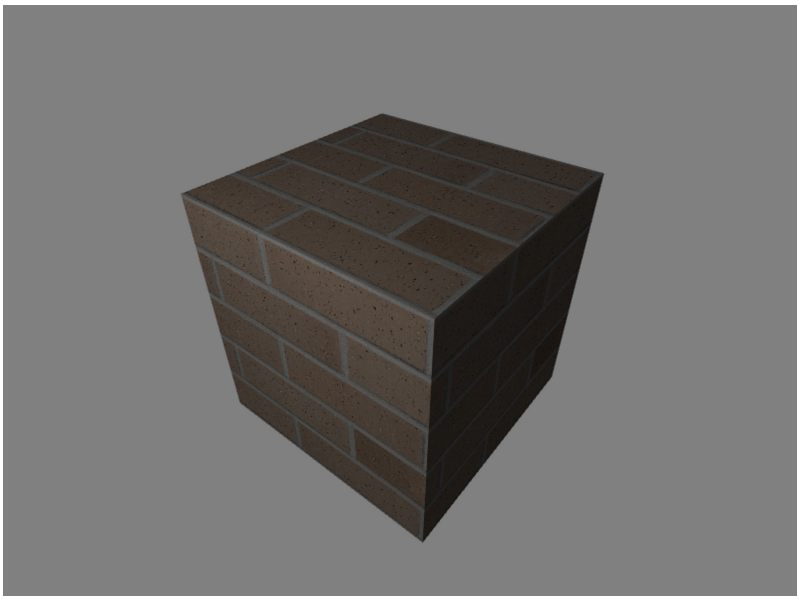
Kd = 0.5f, 0.5f, 0.5f
Ka = 0.5f, 0.5f, 0.5f
Ks = 1.0f, 1.0f, 1.0f

Ld = 1.0f, 1.0f, 1.0f
La = 0.4f, 0.4f, 0.4f
Ls = 1.0f, 1.0f, 1.0f

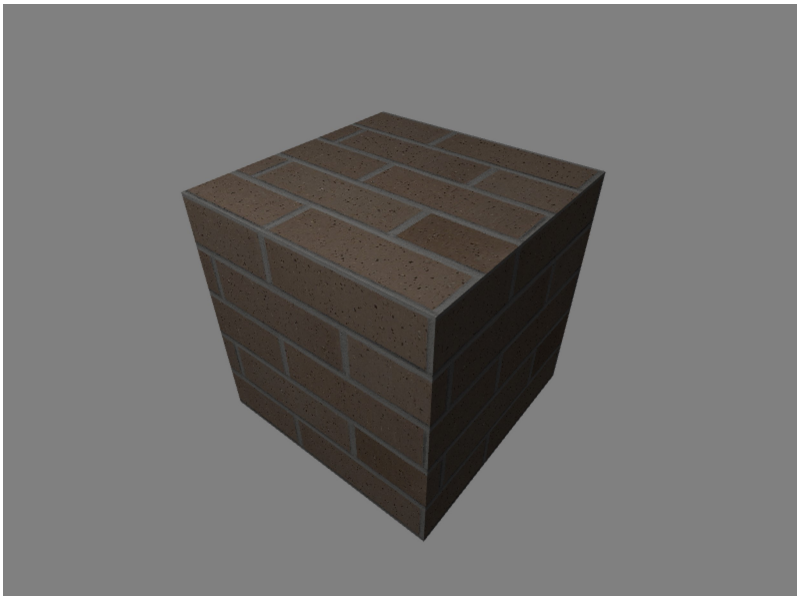
Material.Shininess = 50.0f
Light.Position = 0.0f, 0.5f, 1.2f



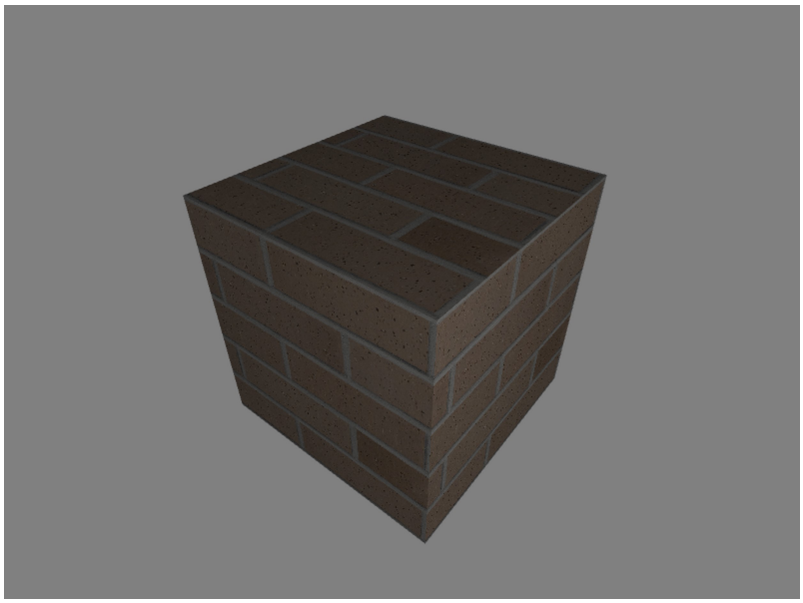
Light.Position = 0.0f, 1.0f, 1.2f



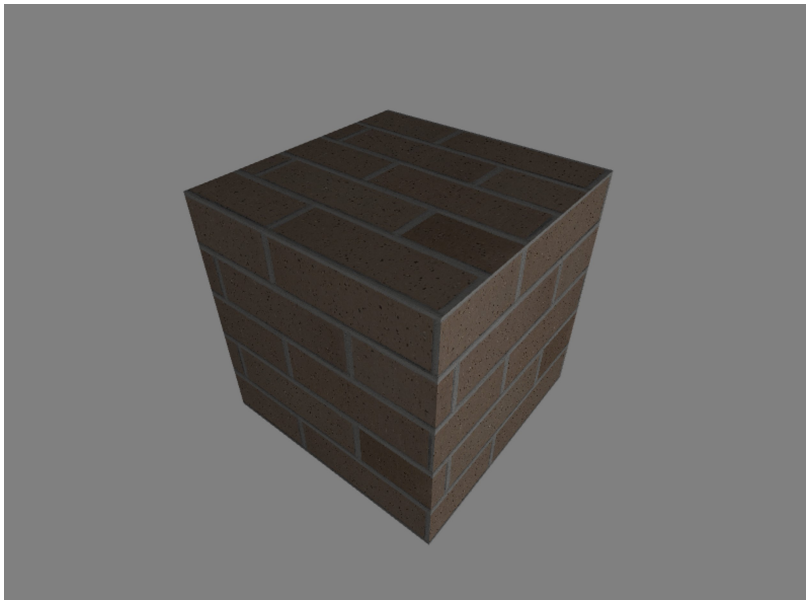
Light.Position = 0.0f, 1.5f, 1.2f



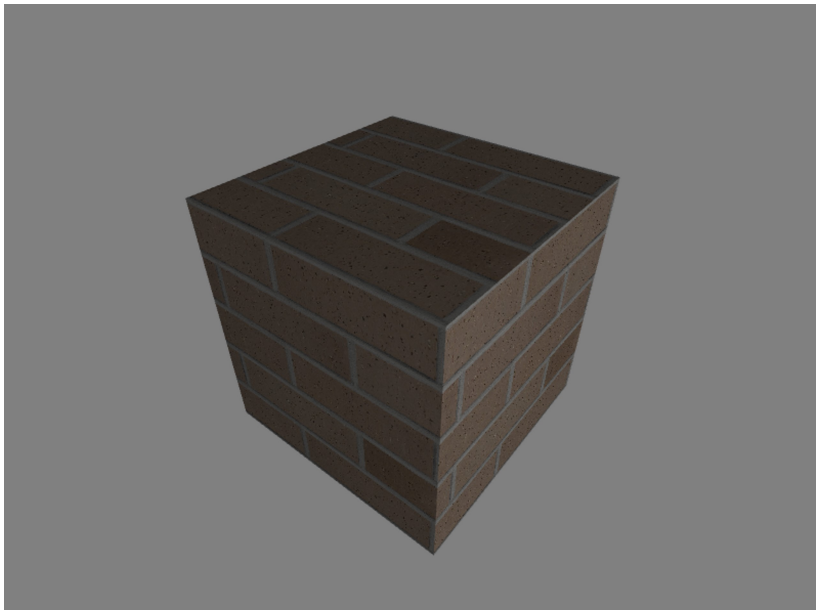
Light.Position = 1.5f, 1.0f, 1.2f



Light.Position = 2.5f, 1.0f, 1.2f



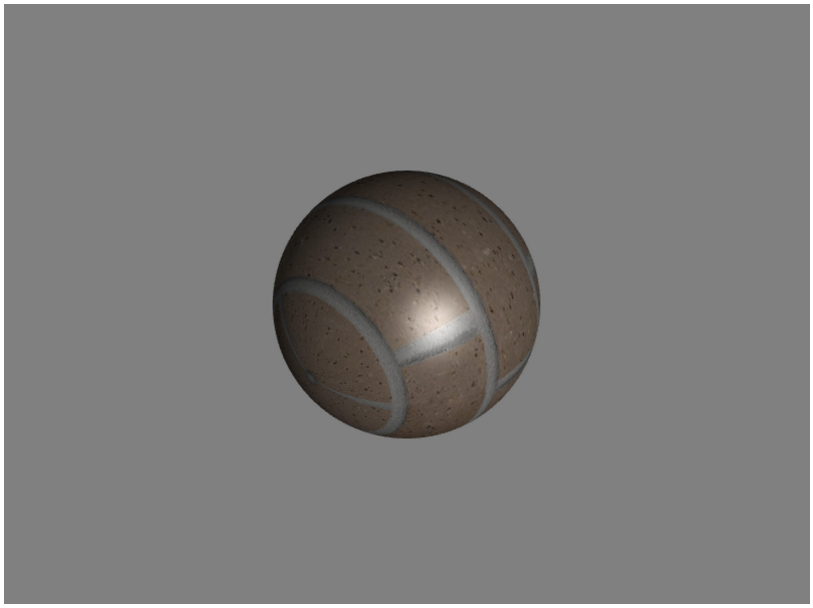
Light.Position = 3.0f, 1.0f, 1.2f



The specular component can't be noticed easily on a geometry like this. Note that the specularity effect relies on gradual changes of curvature, computed by the cosine/dot-product of the camera vector and the reflection vector.

```
vec3 specularComponent
  = Light.Ls
     * Material.Ks
     * pow(clamp(dot(viewCameraDirection, reflectedLightVector), 0.0, 1.0), Material.Shininess);
```

Specularity looks good on spherical geometry

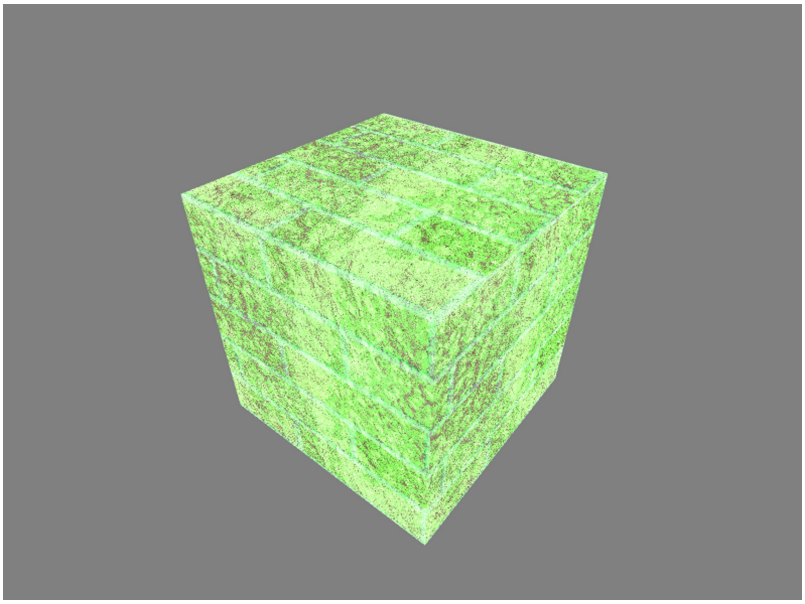


No specularity

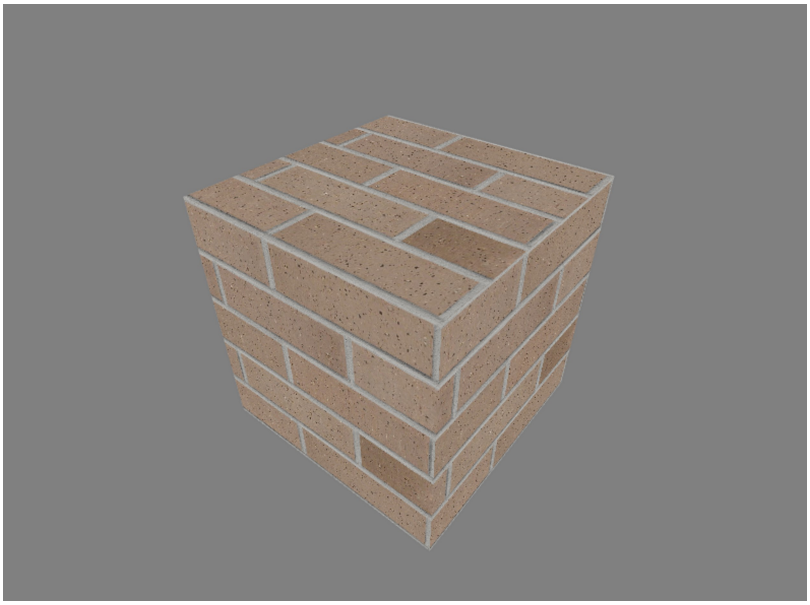
It's interesting that the texture doesn't look deformed on the sphere.

***Multiple textures***

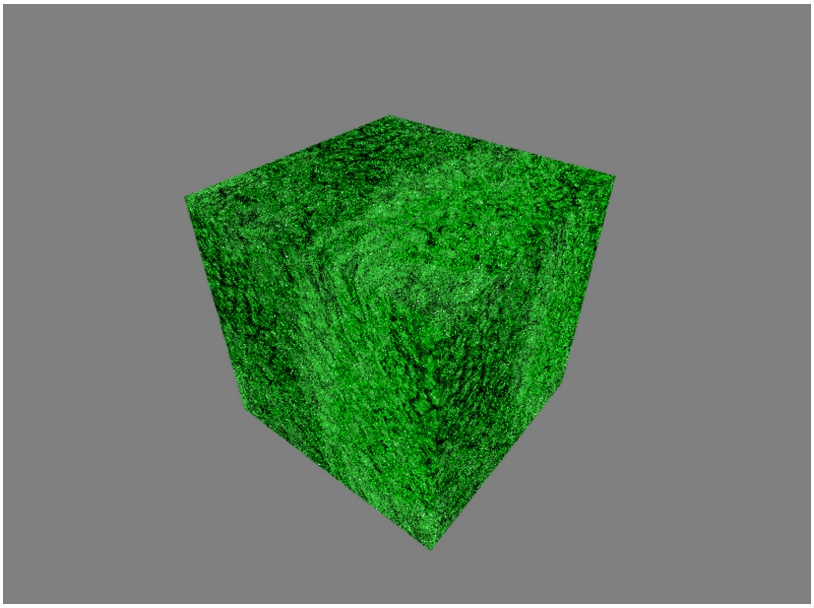
Sum of 2 textures:



=



+



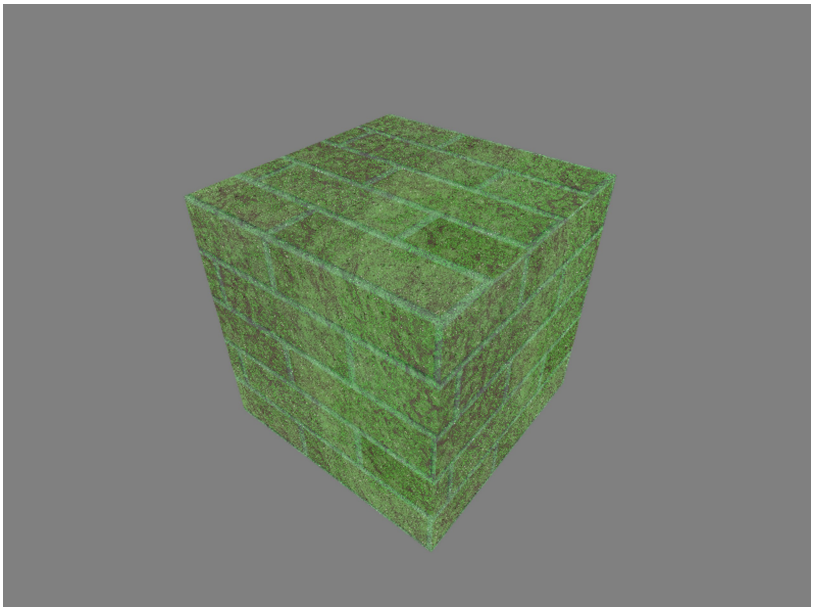
***Alpha***

The `mix(firstColor, secondColor, interpolator)` built-in function can be used to combine 2 textures together. It ***linearly interpolates*** between the first color and the second color, using the 3rd parameter as interpolation parameter:

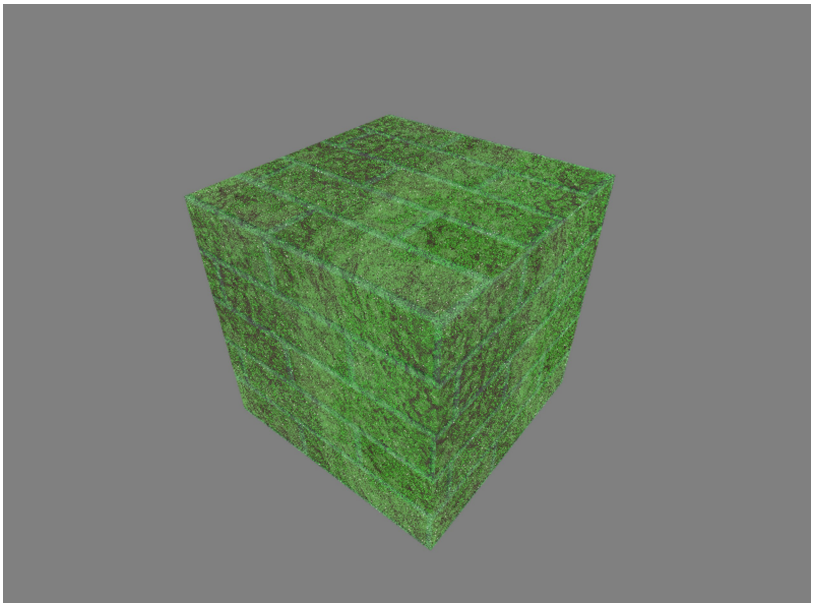
`mixedColor = firstColor*(1-`$\alpha$`) + secondColor*(`$\alpha$`)`

Here I used a value for the interpolation parameter that was the same across all segments:

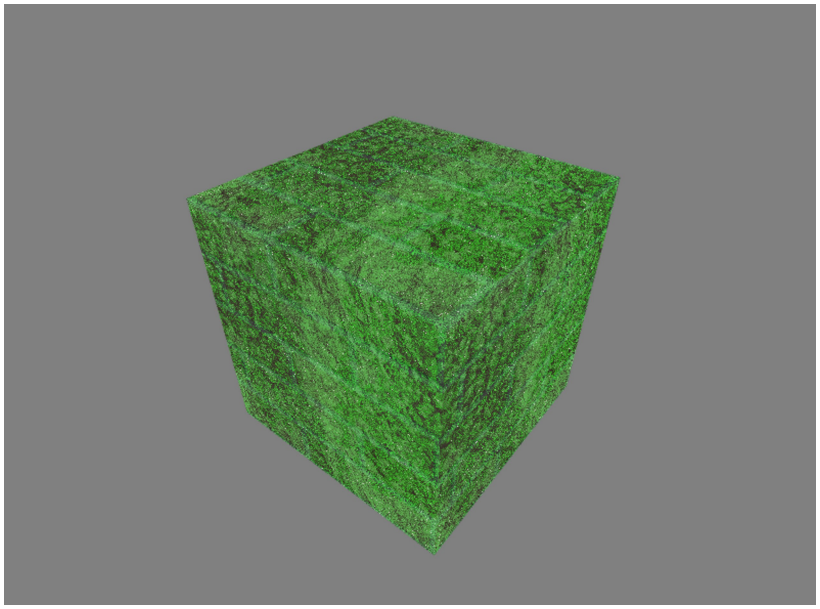
$\alpha$ = interpolation parameter = 0.4



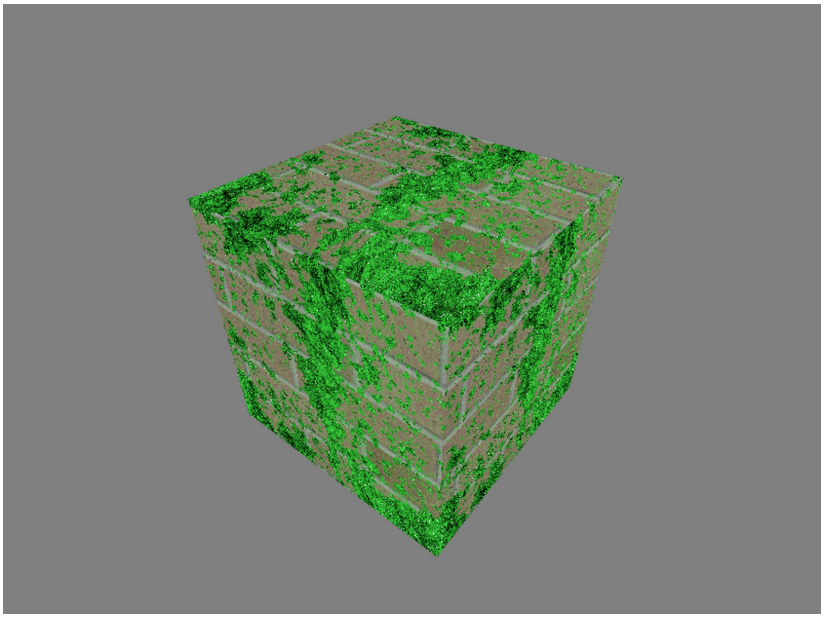
$\alpha$ = interpolation parameter = 0.5



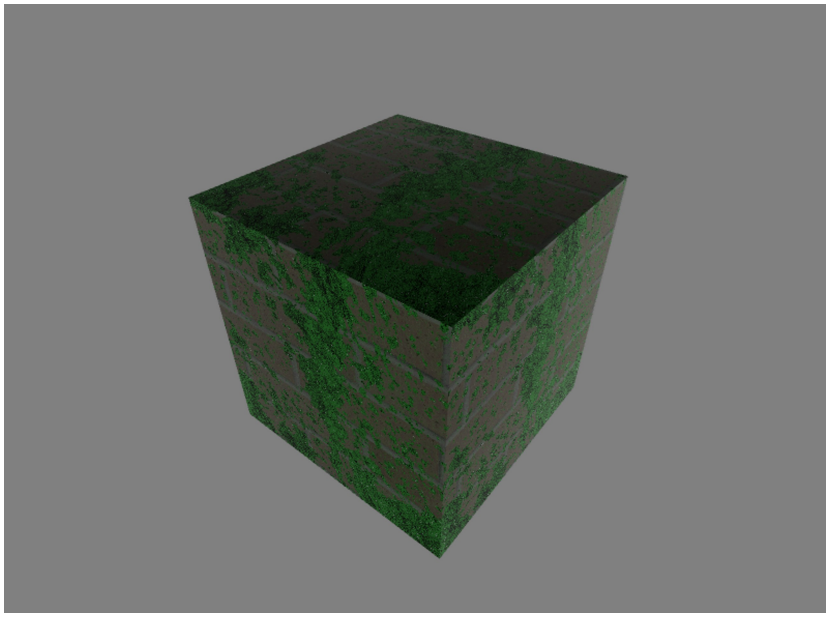
$\alpha$ = interpolation parameter = 0.6



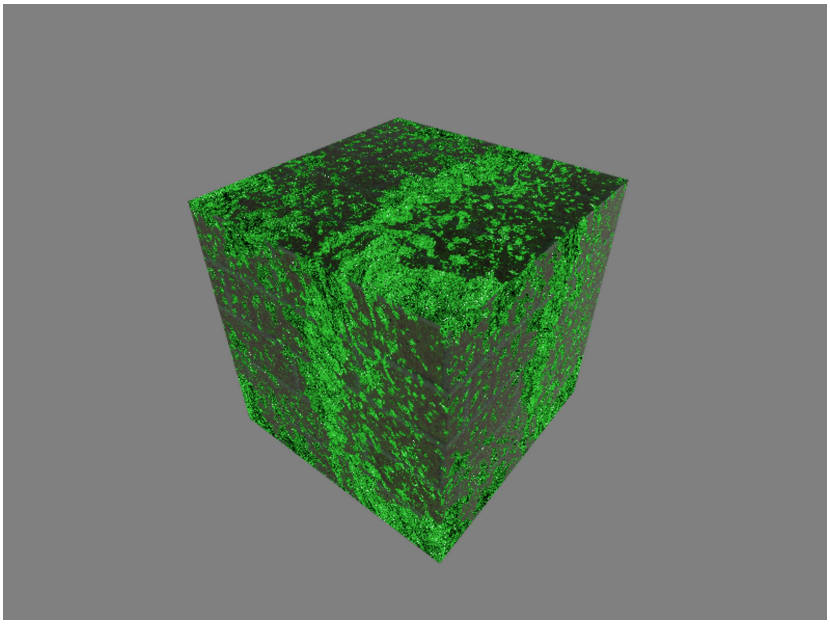
Here I used the moss ***image's encoded alpha value*** (the image is PNG and it encodes an alpha value per pixel). Therefore, each fragment had its own alpha value (the alpha value of the vertex-interpolated texture coordinate):



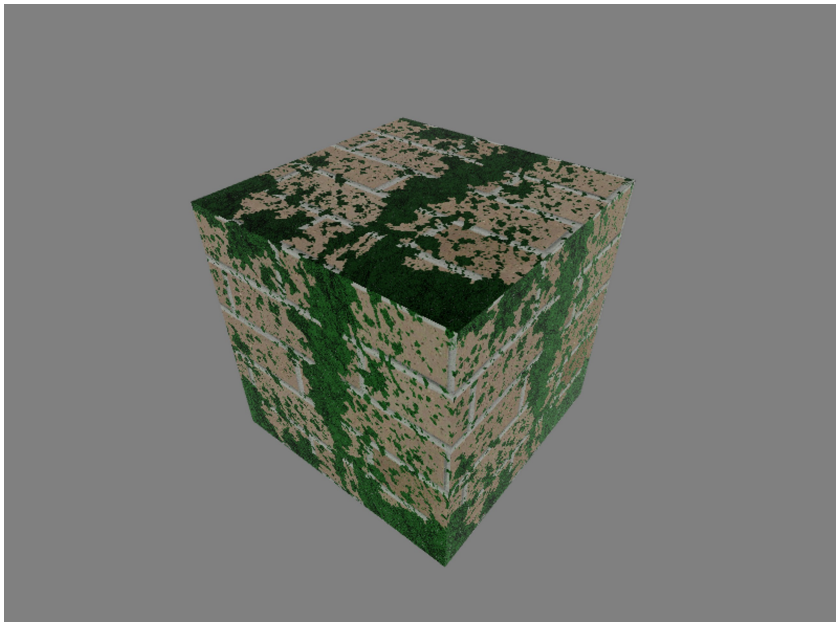
Here I used the Phong reflection model. The alpha blending of the 2 textures is subject to ambient and diffuse treatment, and the specular component is added to the result.



Here I only applied the Phong reflection model to the brick texture. Then I alpha-blended the result with the moss texture. The moss texture is not shaded.



Here I did the opposite: applied the Phong reflection model to the moss texture only. Then I alpha-blended the result with the brick texture. The brick texture is not shaded.

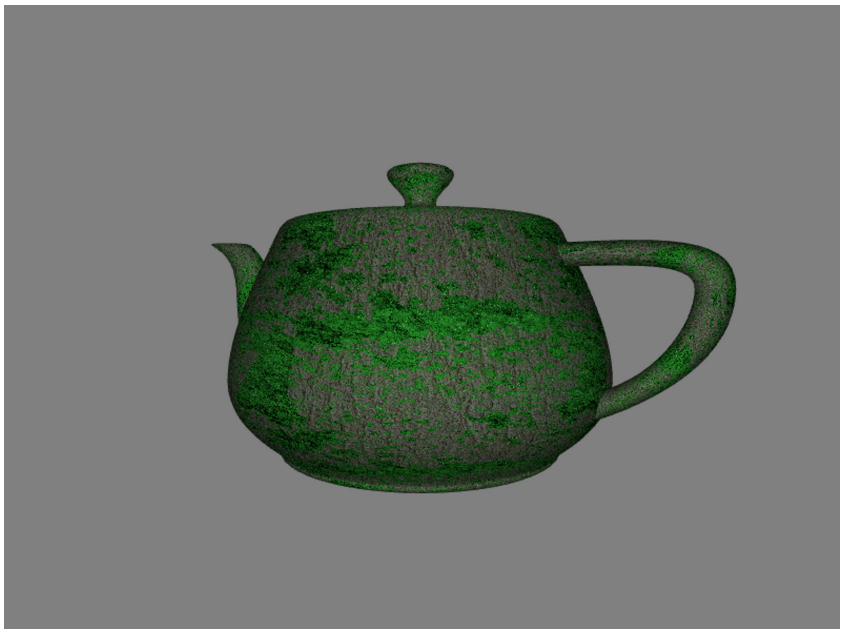


*A popular technique is to **use an additional vertex attribute to augment the amount of blending between the textures**. This additional vertex attribute would allow us to vary the blending factor throughout a model. For example, we could vary the amount of moss that grows on a surface by defining another vertex attribute, which would control the amount of blending between the moss texture and the base texture. A value of zero might correspond to zero moss, up to a value of one that would enable blending based on the texture's alpha value alone.*

***Alpha test***

The encoded alpha values of a PNG image texture can be used to choose what to do with fragments based on the alpha value:

*Alpha-blend between cement texture and moss texture*



*Discard fragment if alpha <= 0.5*

*Discard fragment if alpha >= 0.5*

*Phong-shaded pink if alpha >= 0.5*

*"It is a great way to make holes in objects or to present the **appearance of decay**. If your alpha map has a gradual change in the alpha throughout the map, (for example, an alpha map where the alpha values make a smoothly varying height field) then it can be used to animate the decay of an object. We could vary the alpha threshold (0.15 in the preceding example) from 0.0 to 1.0 to create an animated effect of the object gradually decaying away to nothing".*

## Normal maps

The difference between ***normal mapping*** and ***bump mapping*** is that the texture is used to vary the normal in normal mapping, and in bump mapping the texture is used to vary the position of the vertices. In none of the 2 is the geometry of the bumps actually provided by the application.

The normal map is used in the ***reflection model*** to vary the normal.

Normal maps are typically produced in 3D modeling applications (Maya, Blender, 3D Studio Max) or in Photoshop.

There are 2 ways to use normal maps:

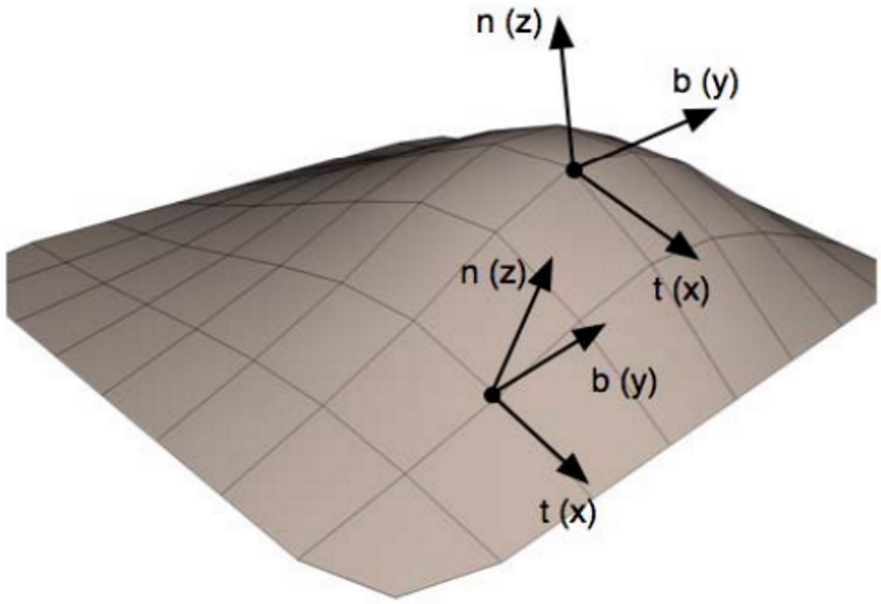
A. You take the vertex interpolated normal of the fragment, add the normal from the normal map to perturb it, renormalize the vector, and use it in the reflection model.

A. Evaluate the reflection model in ***tangent space***: you place a coordinate frame at each vertex, the origin is the vertex, the normal is the z axis, and the x and y axes are, of course, orthogonal to the normal (Akenine-Moller notes that *"These vectors do not have to be truly perpendicular to each other, since the normal map itself may be distorted to fit the surface". Real-Time Rendering, Section 6.1 Bump Mapping*). Usually, you evaluate the reflection model in ***view space***.

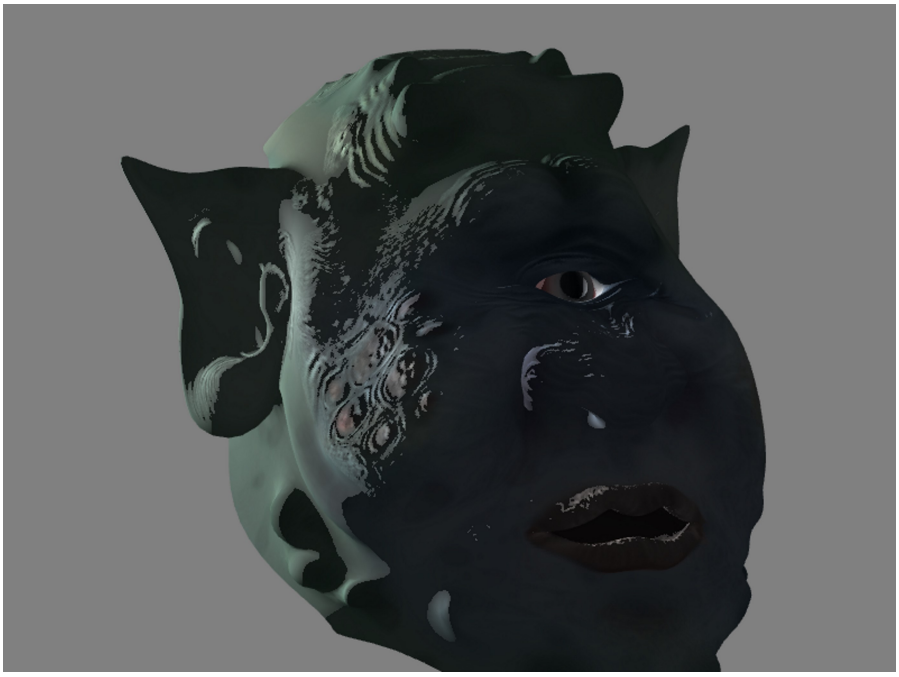
   To work in tangent space, the vertex shader transforms the vectors of direction to the light and camera from world or view space to tangent space. The fragment shader can then evaluate the reflection model in tangent space using these vectors and the normal read from the normal map. The ***view to tangent space transformation matrix***, AKA ***TBN*** is:

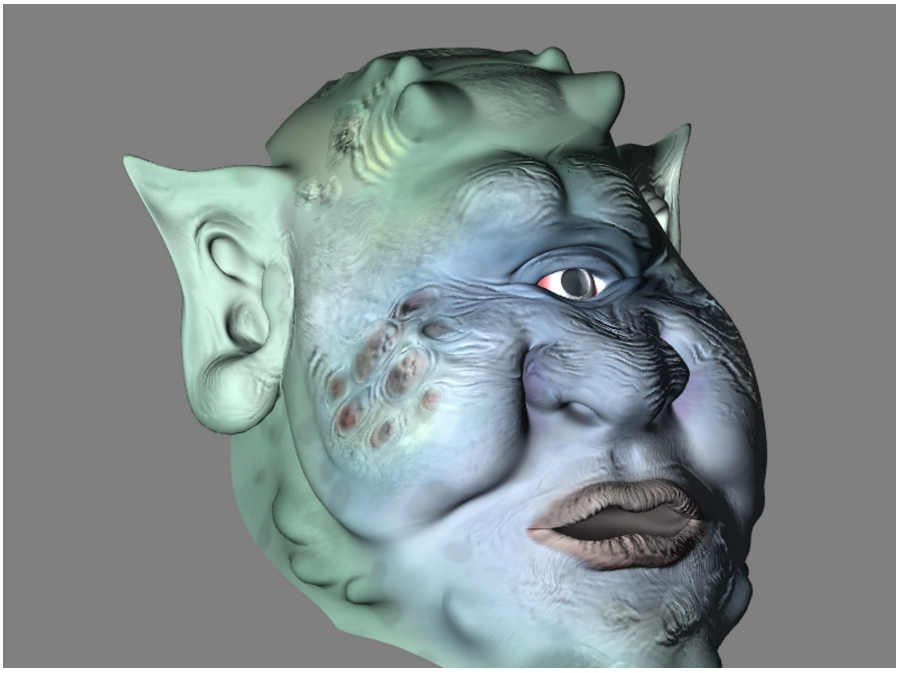
$$\begin{matrix} S_x \\ S_y \\ S_Z \end{matrix} = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

   where *n* (normal vector), *t* (tangent vector), and *b* (bitangent vector) are as shown here:




*Wrinkles, multiple light positions*

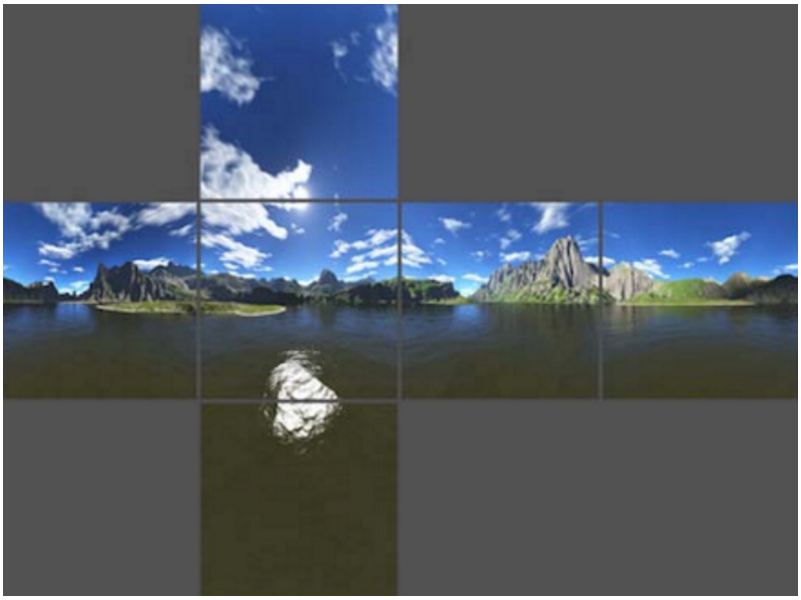




## Environment cube maps and skyboxes

Environment maps are textures that are used to have mirror-like objects reflect the surroundings of the scene, the environment.

A cubemap, in particular, is sampled using a 3D coordinate ***(s, r, t).*** The frame of reference is the reflective object.

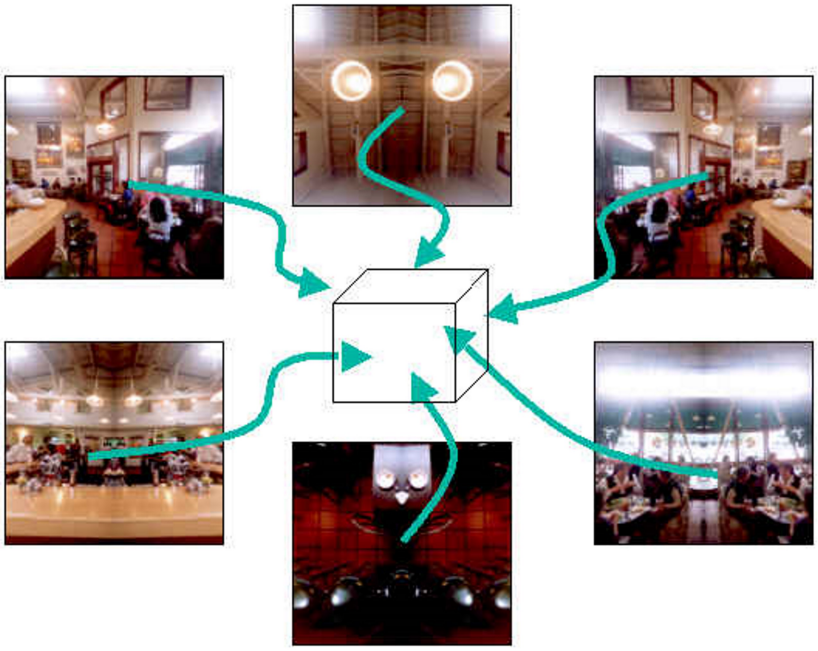


OpenGL implements it as a texture where each mipmap level is a set of 6 square images. The binding target is GL_TEXTURE_CUBE_MAP.

***Skybox***

A ***skybox*** is just a cube whose inner faces have been mapped to the faces of a cubemap, so as to be textured by them.

The skybox cube mesh doesn't have to have texture coordinates. You just access the cubemap in the shader using a uniform of type **`samplerCube`** and then pass the vertex's world coordinates to **`texture()`**.

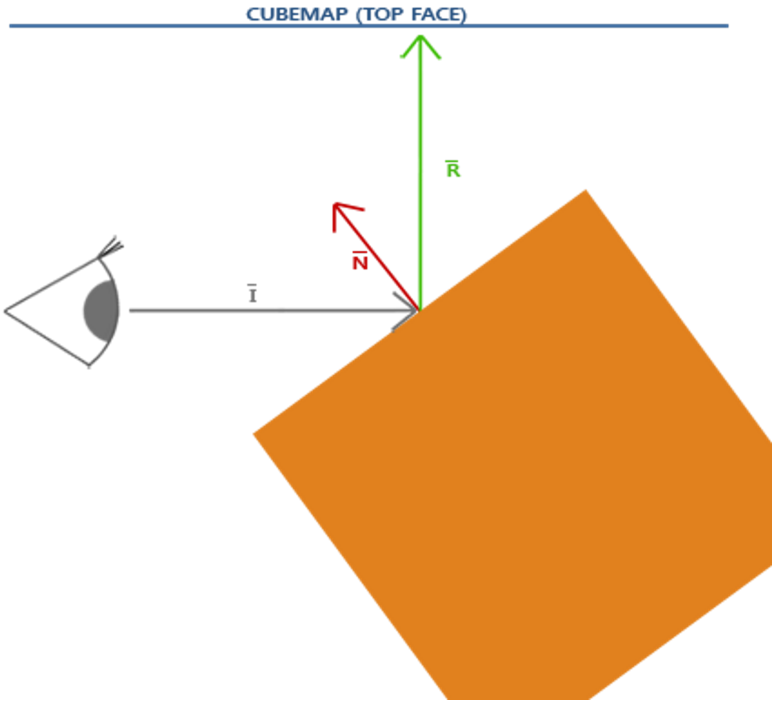


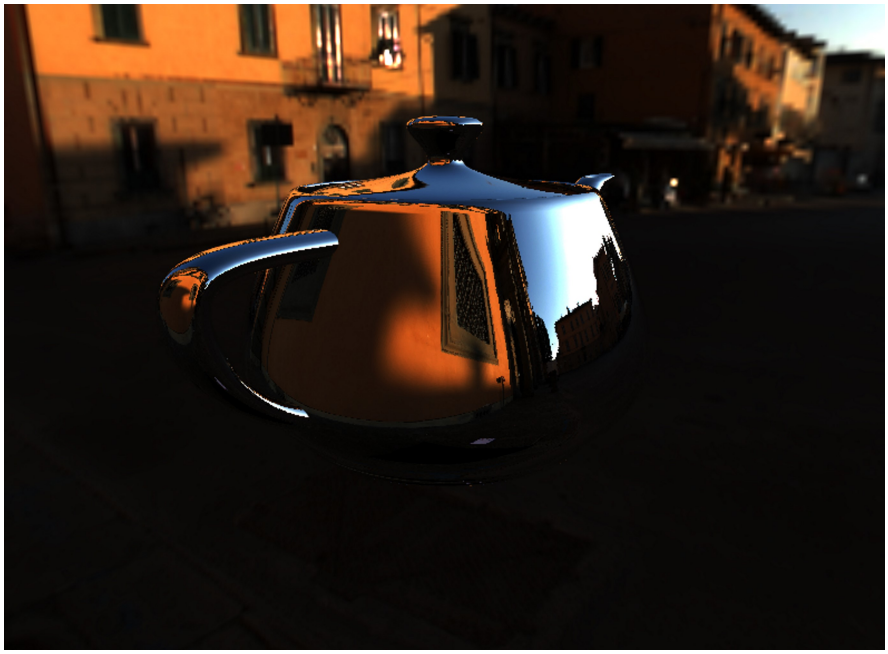
***Environment mapping via cubemap reflection***

The ***computations*** for having a surface ***reflect a cubemap environment*** are done in ***world space***:

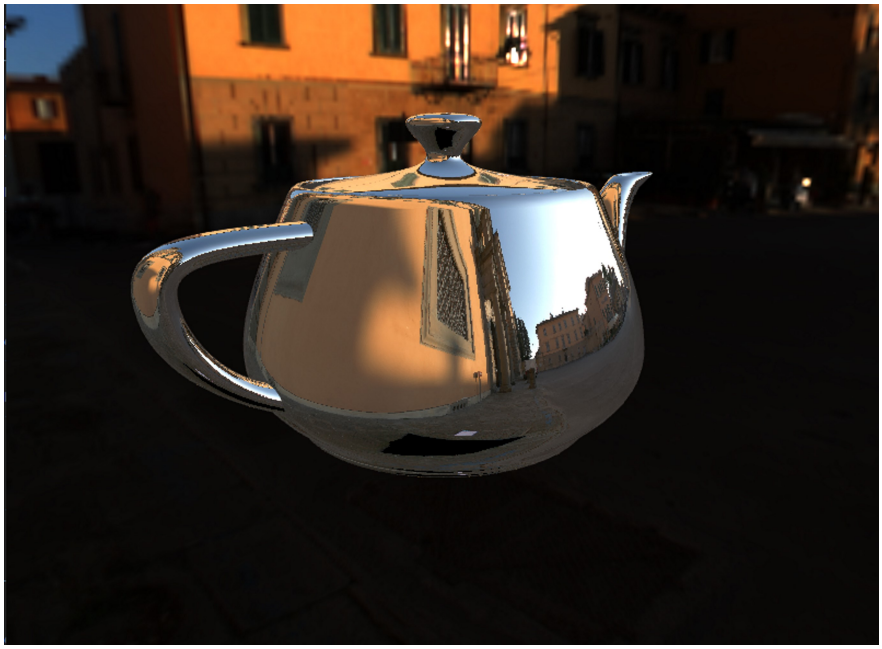
- The skybox is textured with the cubemap texture and is centered at the world origin.
- A reflection direction vector originating at the object's shaded vertex samples the cubemap. The length of this vector doesn't matter.
- The reflection on the surface of the object depends on the position of the eye. If the eye moves, the reflection changes.
- The reflection direction vector is a function of the incident vector of a view ray and the normal at the shaded vertex. The view ray is the additive inverse of a vector that originates at the shaded vertex and goes in the direction of the eye.

  $\bar{R}$ is easily computed in GLSL with the ***reflect(viewRay, normal)*** function.



*Gamma-corrected*



*The camera rotates around the teapot.*
*The reflection shows the face of the cubemap that is behind the camera at that frame.*

If you compute the direction of reflection in view space instead of world space, you are going to see the same reflection all the time, even when the camera is rotating around the teapot. This is because the computed reflection direction used to sample the cubemap for a given fragment doesn't change across frames: in view space, the camera's position is fixed at the origin and, with symmetric objects like this teapot, vertex positions processed by the vertex shader will always be the same in view space even when the vertices are actually different due to the rotation. A given pixel will always be given the same reflection direction vector, i.e. (same position, different vertex) - (0, 0, 0); that direction will always sample the same texel of the cubemap.

*Different angle, same reflection*

*Yet another angle, same reflection.*
*The reflection wrongly shows the cubemap face that's behind the teapot, instead of the face that's behind the camera*

Limitations of this technique:

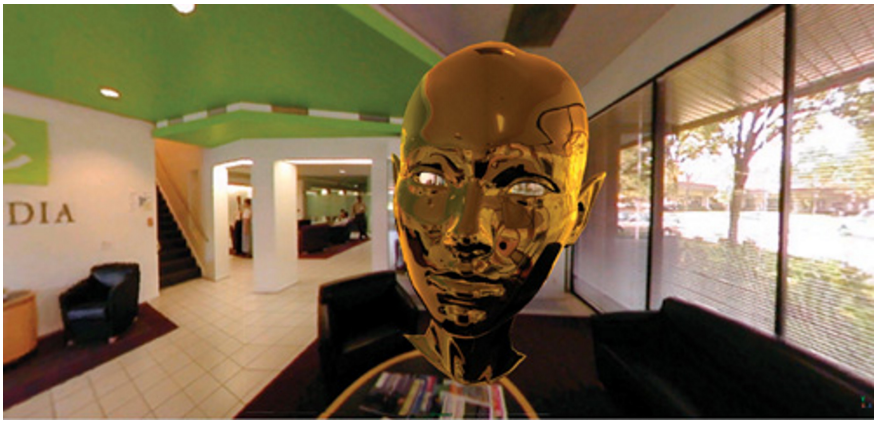
- The reflection ray only samples the cubemap and not the rest of the object in the scene. If a reflection ray intersects another object, you'd expect this ray to sample this object, and this object to appear in the reflection. If you want other objects to appear in the reflection, you'd have to generate a cube map ***at run-time*** on every frame, by rendering the scene to 6 textures, each with the camera placed at the center of the teapot and facing towards one of the faces of the cube (up, down, left, right, front, and back); these rendered textures would show objects that were there in the view frustum. You'd then produce the final render using these 6 textures as the cubemap to sample reflections from.

- The features of the cubemap reflected on the teapot are always of the same size (relative to the apparent size of the teapot), no matter where the object is located in the scene. For example, the reflection of the window on the teapot will always be of the same size, no matter how close the teapot appears to be to the face of the cubemap that contains the window.

The cubemap is said to be ***infinitely far way***. This is interesting because the cubemap actually has a geometry of finite dimension. But when sampling the cubemap texture using ***texture(cubeMap, worldReflectDir)*** there is no notion of distance.

This GPU Gem solves this problem by ***localizing the reflections***.

*Big green ceiling panel reflection*




*Smaller green ceiling panel reflection*

***Environment mapping and refraction***

I tried to code Snell's law to compute the direction of refraction, but I couldn't.

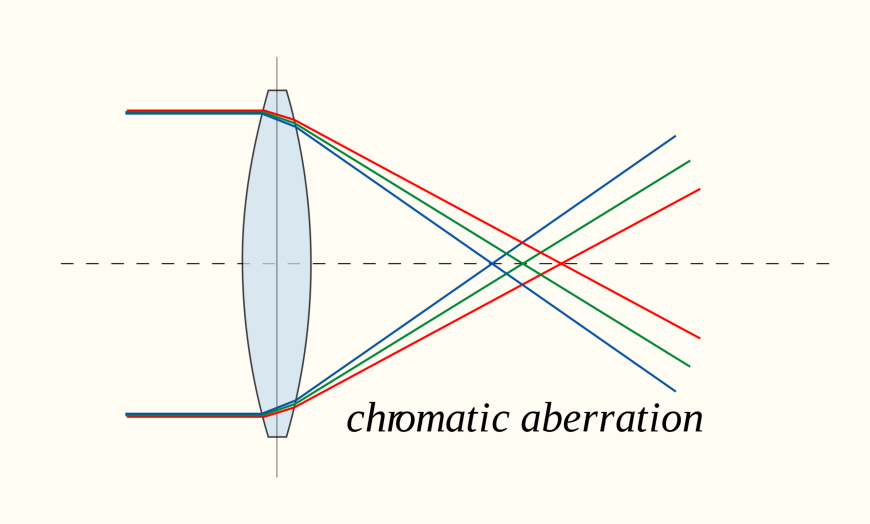
This uses the built-in ***refract(worldDirectionToEye, worldVertexNormal, etaPrime/eta)***, where *etaPrime* is the refractive index of the teapot and *eta* of the air.

The direction of refraction samples the cubemap through the teapot. This is, of course, not physically accurate, because we are only refracting when entering the object, but not when leaving it (since we are not refracting when leaving, there's ***no total internal reflection phenomenon***, assuming that the object is solid).

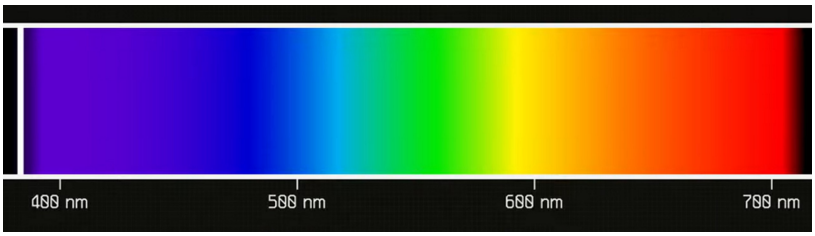
Note that each fragment refracts and reflects at the same time. This is done by a constant interpolation of the cubemap sample of the refraction and the cubemap sample of the reflection, with an interpolator of 0.1, i.e. 90% of the contribution to the fragment's color comes from the sample of the refraction and 10% from the reflection.

Limitations of this technique:

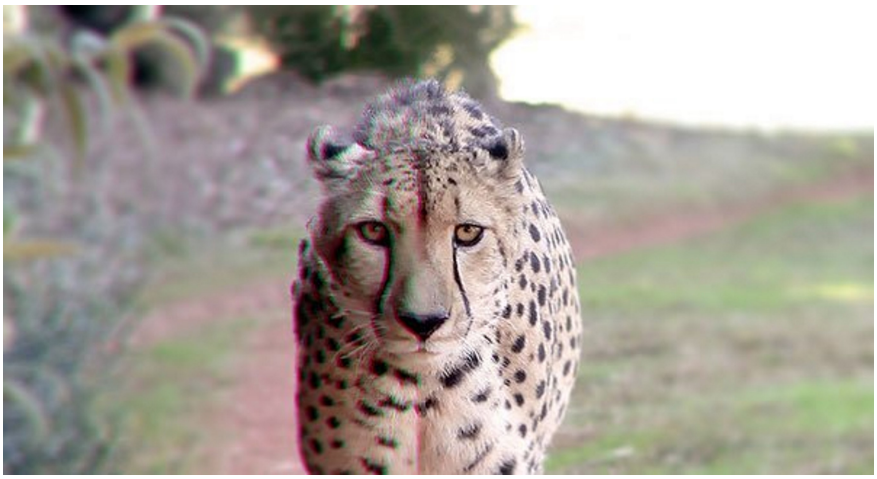
- The same limitations of the technique for reflections.
- No Fresnel effect.
- No ***chromatic aberration*** effect. This is an effect that occurs in real lenses. Lenses bend shorter wavelengths more than longer ones, resulting in a separation of white light into its constituent colors. In photographs, this manifests as ***rainbow fringes*** along edges that separate bright from dark regions.




Blue is the shortest wavelength; red is the longest one.




*The left half shows chromatic aberration (a rainbow fringe). The right half doesn't.*



## Image-based Lighting, Irradiance Environment Maps

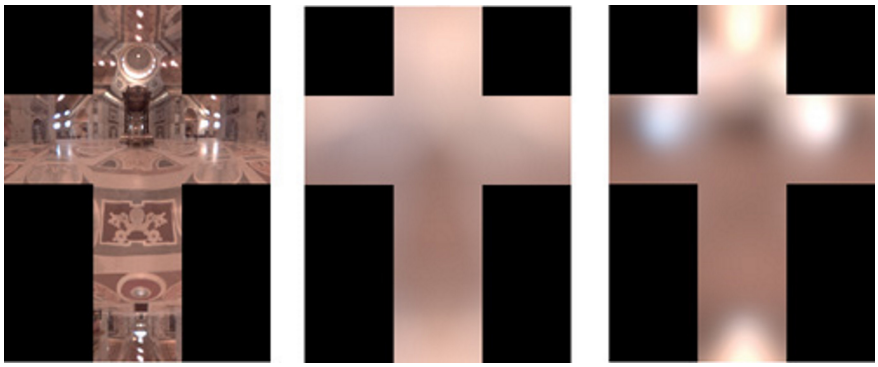
The sources of light are there in the environment image that surrounds the scene.

This technique is based on environment mapping using cube maps. The difference here is that we derive ***3 cube maps*** from the same set of environment images:

- The first cube map is used both for texturing the skybox and as source of color samples for reflections and refraction.
- The second cube map stores precomputed diffuse irradiance values.

- The third cube map stores precomputed specular irradiance values.

*Color, diffuse irradiance environment map, and specular irradiance environment map.*



The diffuse irradiance and specular irradiance maps ***don't store*** the brightness of the texel of the original color texture. No. Each texel of these maps is the computation of the ***Phong reflection*** model for any fragment that has a world space normal that samples it (recall that a cube map is sampled by ***texture()*** with a world space direction, in this case the world space normal of a vertex).

Unlike previous shaders I've written that had a ***single point light***, an environment map image may have multiple sources of light. Since it would be too expensive to analyze the image and identify the texels that are "real" light sources (as opposed to objects and non-emissive surfaces), we consider the ***contribution of every texel*** in the computation of the Phong reflection model ***at every texel***:

*For each vertex of the surface, we compute, $I$, the diffuse component of the Phong reflection model.*
*We store $I$ in the texel of the resulting diffuse irradiance cube map that is indexed/sampled by the vertex's normal $n$.*
*$n$, the normal, identifies a texel of the cube map uniquely, where we'll store the result.*
*$N$ is the total number of texels of the color environment cube map.*
*For each of the $N$ texels, we compute the direction vector $s_i$ from the current vertex, and sample the brightness of the color cube map texture using $s_i$ to obtain its light intensity $L_i$.*
*The sum of them all is what we store in the diffuse irradiance cube map indexed at $n$.*

$$I = K_d \left( \sum_{i=1}^{N} \max(0, \mathbf{s}_i \cdot \mathbf{n}) L_i \right)$$

*No Phong diffuse reflection model yet.*

*Phong diffuse reflection using a precomputed irradiance map.*

*The diffuse shading of the model corresponds to the lighting of the environment.*

*Each fragment simply samples the diffuse irradiance map using its interpolated world-space normal and combines this irradiance value with its base color (the cow texture).*

## Rendering to a texture

There is a default framebuffer that is the default render target.

Framebuffer objects or FBOs are additional framebuffers that the application can create and set as render targets.

Much like VBOs, *vertex buffer objects,* are application-side abstractions