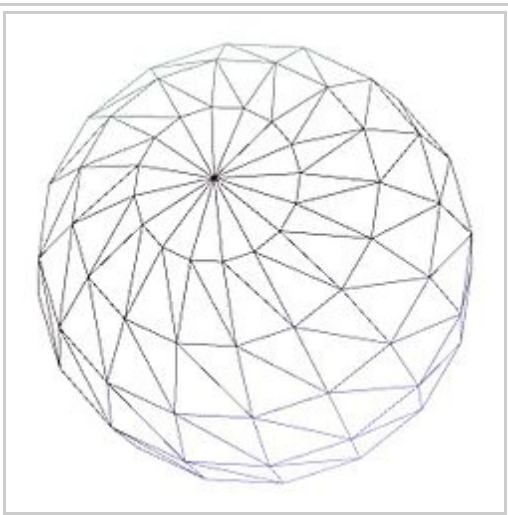
GLSL Programming/GLUT/Textured Spheres

The Earth seen from Apollo 17. The shape of the Earth is close to a quite smooth sphere.

This tutorial introduces **texture mapping**.

It's the first in a series of tutorials about texturing in GLSL shaders in OpenGL 2.x. In this tutorial, we start with a single texture map on a sphere. More specifically, we map an image of the Earth's surface onto a sphere. Based on this, further tutorials cover topics such as lighting of textured surfaces, transparent textures, multitexturing, gloss mapping, etc.

Contents



A triangle mesh approximating a sphere.

An image of the Earth's surface. The horizontal coordinate represents the longitude, the vertical coordinate the latitude.

Texture Mapping

The basic idea of “texture mapping” (or “texturing”) is to map an image (i.e. a “texture” or a “texture map”) onto a triangle mesh; in other words, to put a flat image onto the surface of a three-dimensional shape.

To this end, “texture coordinates” are defined, which simply specify the position in the texture (i.e. image). The horizontal coordinate is officially called `s` and the vertical coordinate `t`. However, it is very common to refer to them as `x` and `y`. In animation and modeling tools, texture coordinates are usually called `u` and `v`.

In order to map the texture image to a mesh, every vertex of the mesh is given a pair of texture coordinates. (This process (and the result) is sometimes called “UV mapping” since each vertex is mapped to a point in the UV-space.) Thus, every vertex is mapped to a point in the texture image. The texture coordinates of the vertices can then be interpolated for each point of any triangle between three vertices and thus every point of all triangles of the mesh can have a pair of (interpolated) texture coordinates. These texture coordinates map each point of the mesh to a specific position in the texture map and therefore to the color at this position. Thus, rendering a texture-mapped mesh consists of two steps for all visible points: interpolation of texture coordinates and a look-up of the color of the texture image at the position specified by the interpolated texture coordinates.

In OpenGL, any valid floating-point number is a valid texture coordinate. However, when the GPU is asked

to look up a pixel (or "texel") of a texture image (e.g. with the "texture2D" instruction described below), it will internally map the texture coordinates to the range between 0 and 1 in a way depending on the "Wrap Mode" that is specified when importing the texture: wrap mode "repeat" basically uses the fractional part of the texture coordinates to determine texture coordinates in the range between 0 and 1. On the other hand, wrap mode "clamp" clamps the texture coordinates to this range. These internal texture coordinates in the range between 0 and 1 are then used to determine the position in the texture image: $(0, 0)$ specifies the lower, left corner of the texture image; $(1, 0)$ the lower, right corner; $(0, 1)$ the upper, left corner; etc.

Texturing a Sphere

To map the image of the Earth's surface onto a sphere, you first have to load the image. For this, use SOIL as explained in OpenGL Programming Tutorial 06.

```
glActiveTexture(GL_TEXTURE0);
GLuint texture_id = SOIL_load_OGL_texture
   (
    "Earthmap720x360_grid.jpg",
    SOIL_LOAD_AUTO,
    SOIL_CREATE_NEW_ID,
    SOIL_FLAG_INVERT_Y | SOIL_FLAG_TEXTURE_REPEATS
    );
```

`SOIL_FLAG_TEXTURE_REPEATS` will make the texture repeat itself when using texture coordinates outside of [0, 1].

Vertex shader:

```
attribute vec3 v_coord;
varying vec4 texCoords;
uniform mat4 m,v,p;

void main(void) {
    mat4 mvp = p*v*m;
    gl_Position = mvp * vec4(v_coord, 1.0);
    texCoords = vec4(v_coord, 1.0);
}
```

Fragment shader:

```
varying vec4 texCoords;
uniform sampler2D mytexture;

void main(void) {
    vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5,
                                  (asin(texCoords.z) / 3.1415926 + 0.5));
        // processing of the texture coordinates;
        // this is unnecessary if correct texture coordinates are specified by the application

    gl_FragColor = texture2D(mytexture, longitudeLatitude);
        // look up the color of the texture image specified by the uniform "mytexture"
        // at the position specified by "longitudeLatitude.x" and
        // "longitudeLatitude.y" and return it in "gl_FragColor"
}
```

If everything went right, the texture image should now appear on the sphere. Congratulations!

How It Works

Since many techniques use texture mapping, it pays off very well to understand what is happening here. Therefore, let's review the shader code:

The vertices of the sphere object come from the FreeGLUT `glutSolidSphere` function. We'll need them in the fragment shader to convert them to texture coordinates in the space of the texture image.

The vertex shader then writes the texture coordinates of each vertex to the varying variable `texCoords`. For each fragment of a triangle (i.e. each covered pixel), the values of this varying at the three triangle vertices are interpolated (see the description in "Rasterization") and the interpolated texture coordinates are given to the fragment shader. The fragment shader then uses them to look up a color in the texture image specified by the uniform `mytexture` at the interpolated position in texture space and returns this color in `gl_FragColor`, which is then written to the framebuffer and displayed on the screen.

In this case, we generate the texture coordinates algorithmically, but usually they are specified through your 3D modeler, and passed as additional vertex attributes.

It is crucial that you gain a good idea of these steps in order to understand the more complicated texture mapping techniques presented in other tutorials.

Repeating and Moving Textures

In some 3D frameworks, you might have met parameters **Tiling** and **Offset**, each with an **x** and a **y** component. These parameters allow you to repeat the texture (by shrinking the texture image in texture coordinate space) and move the texture image on the surface (by offsetting it in texture coordinate space). To reproduce this behavior, another uniform has to be defined:

```
uniform vec4 mytexture_ST; // tiling and offset parameters
```

We can specify such a `vec4` uniform for each texture. (Remember: "S" and "T" are the official names of the texture coordinates, which are usually called "U" and "V", or "x" and "y".) This uniform holds the **x** and **y** components of the **Tiling** parameter in `mytexture_ST.x` and `mytexture_ST.y`, while the **x** and **y** components of the **Offset** parameter are stored in `mytexture_ST.w` and `mytexture_ST.z`. The uniform should be used like this:

```
gl_FragColor = texture2D(mytexture, mytexture_ST.xy * texCoords.xy + mytexture_ST.zw);
```

This makes the shader behave like the built-in shaders. In the other tutorials, this feature is usually not implemented in order to keep the shader code a bit cleaner.

And just for completeness, here is the new fragment shader code with this feature:

```
varying vec4 texCoords;
uniform sampler2D mytexture;
uniform vec4 mytexture_ST; // tiling and offset parameters

void main(void) {
    vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5,
                                  (asin(texCoords.z) / 3.1415926 + 0.5));

    // Apply tiling and offset
    vec2 texCoordsTransformed = longitudeLatitude * mytexture_ST.xy + mytexture_ST.zw;

    gl_FragColor = texture2D(mytexture, texCoordsTransformed);
}
```

You can try, for instance, to duplicate all continents (scale 2x horizontally to see the texture twice - make sure your texture is `GL_REPEAT`ing), and reduce the north pole (start at -0.05 vertically to reduce the top):

```
glUniform4f(uniform_mytexture_ST, 2,1, 0,-.05);
```

Summary

You have reached the end of one of the most important tutorials. We have looked at:

- How to import a texture image and how to attach it to a texture property of a shader.
- How a vertex shader and a fragment shader work together to map a texture image onto a mesh.
- How tiling and offset parameters for textures work and how to implement them.

Further Reading

If you want to know more

- about the data flow in and out of vertex shaders and fragment shaders (i.e. vertex attributes, varyings, etc.), you should read the description of the “OpenGL ES 2.0 Pipeline”.
- about the interpolation of varying variables for the fragment shader, you should read the discussion of the “Rasterization”.

< GLSL Programming/GLUT

Back to **OpenGL** Programming - Lighting section | Back to **GLSL** Programming - GLUT section