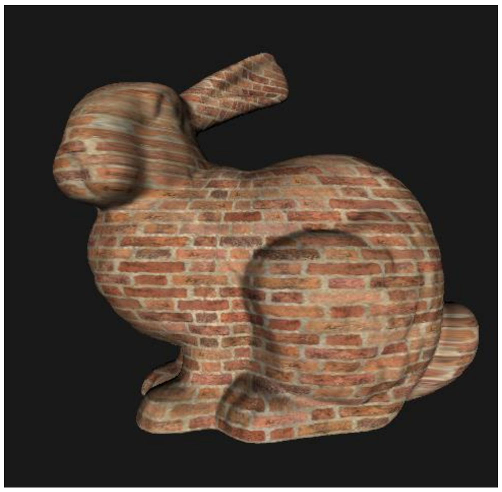
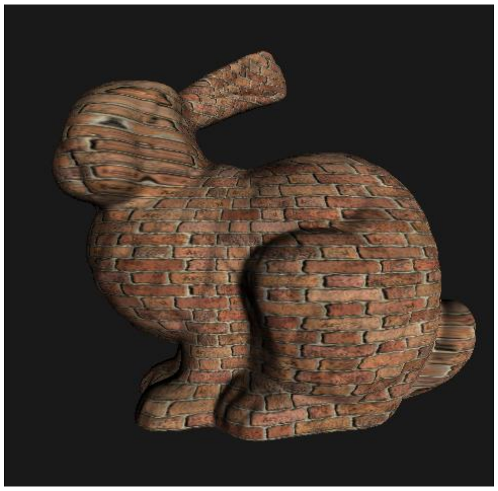
Shaders – Textures and Normal Map





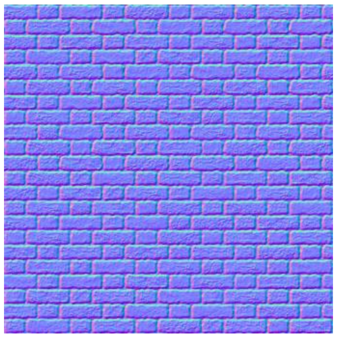
Description:

We are taking the next step in using the GL shading language (GLSL) in this lab. Last time we used GLSL to draw a model (and compute diffuse shading). In this lab, we will be learning how to use texture maps and experimenting with normal maps.

In this lab, you have three tasks. First, your task is to do texture mapping using a cylindrical projection. Cylindrical projection for texture-mapping is the idea that you first map some texture onto an imaginary cylinder that surrounds the 3D model. Then, the texture is "projected" onto the 3D model (see the left image above).

Your second task is to implement normal mapping. With normal mapping, the idea is to use two textures, a regular texture (e.g. a brick texture) and a texture that encodes the normals of the vertices as RGB values. With normal mapping, you can fake a sense of depth (e.g. see right image above). Finally, your third task is to repeat the texture patterns.

The texture maps you are provided with is shown on the left below (brick.ppm), and the normal map is shown on the right (bump.ppm):



Your Task:

Task 1:

- First, you should get the bunny to render by using your code from the previous lab. Note that I have changed some variable names (e.g. the light vector), so double-check!!
- Once that works, you will need to pass the textures and the light position from the CPU to the GPU. On the CPU side, things are all set up for you -- see MyGLCanvas::drawScene(). You need to edit your shaders to receive the information.
    - The data type of a texture is of type "sampler2D" in GLSL.
- You should start with just the basic texture mapping. To do so, you will need to figure out the texture coordinate given a vertex's 3D position in object space. This is where you'll implement the idea of a cylindrical projection.
    - To figure out where a vertex maps to on a cylinder, you can imagine casting a ray from the y-axis through the vertex in question towards an imaginary cylinder that lie outside of the object.
    - Then your task is to figure out the (u, v) coordinate of the intersection point if you were to "unroll" the cylinder.
    - Note that there's an easier implementation of this. Instead of casting a ray, the trick is to imagine that the vertex is already on the surface of a cylinder (note that this means that there's no longer a single imaginary cylinder). It's going to turn out that the math is identical.
- Once you figured out the texture coordinate, you will need to pass the texture coordinate to the fragment shader.
    - To make sure that you have done this correctly, check by setting the output color to (texCoord.x, texCoord.y, 1.0, 1.0);
- Given a texture coordinate, you can find the color of a pixel on that texture using the function vec4 texture(sampler2D textureName, vec2 texCoord)
    - Again, to make sure that you're doing this correctly, check by setting the output color to (texColor.x, texColor.y, texColor.z, 1.0);
    - Note: the returned values from calling the texture() function will be between 0 and 1.0 (not 0 and 255).
- To make the shading look right, add diffuse lighting.

Task 2:

- Your second task is to extend this code to support normal mapping.
    - You will need to pass the binary variable useNormalMap to the shaders.
- If "useNormalMap" is turned on, you will need to do two things:

    - First, use the normal values from the normal map. **Be careful** though, because the normal map is stored as a texture, all the values are positive! You will need to convert those positive values to sit between -1.0 and 1.0 for each of the x, y, and z directions. Per convention, x is red, y is green, and z is blue.
    - Apply diffuse lighting using the normal from the normal map.
- This results in a "flat" looking bunny, but as you move the light source, you should be able to see the lighting change as if the bricks are raised.
- To add shading from the geometry of the bunny, combine diffuse lighting from geometry and the diffuse lighting from the normal maps.
    - The idea is very simple, compute a dot product based on the diffuse of the geometry and multiply that by the dot product you just computed from the normal map. The resulting dot product is the value you use when multiplying with the texture color.

Task 3:

- Repeat the brick texture patterns 3 times.
- The two images below show: (left) the texture not repeated; (right) the texture repeated three times.
- The trick to solving this task is to realize that the GLSL's texture function (that is: vec4 texture(sampler2D textureName, vec2 texCoord)) automatically performs wrapping. For example, if texCoord.x = 1.2, it'll be auto-wrapped to 0.2.
    - p.s. this texture function also handles texture coordinate values that are negatives – this information may be useful for your Assignment 5.