# UM EECS 487
# Lab 6: Textures and Mapped Pixel Buffer Object

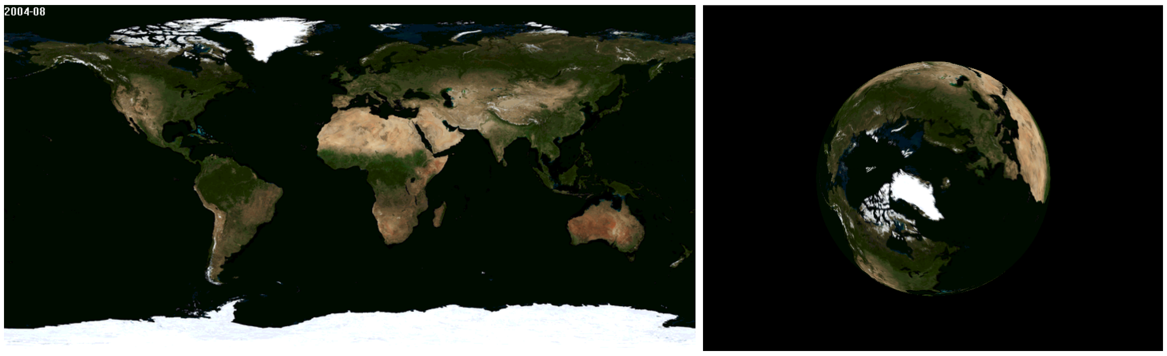

FIGURE 1. Screenshot showing the NASA Blue Marble map used as texture and the result of the mapping (http://en.wikipedia.org/wiki/The_Blue_Marble).

You're provided with the NASA Blue Marble world map. Your task is to map it onto the sphere you constructed in the previous lab. Aside from texturing, this lab will also exercise your understanding of the differences between client-side vertex array, vertex buffer object, and mapped vertex buffer object. To that end, you are to complete the following tasks:

1. Extend the previous lab to add texture mapping to the sphere using client-side memory to hold the texture image.
2. Enable mipmapping.
3. Load texture image from file directly into pixel buffer object using buffer mapped memory.
4. Modify your code from previous lab and task 1 above to map vertex buffer memory to client-side address space so that the vertex and texture coordinates arrays are written directly into the vertex buffer object.

## TASK 1: TEXTURED SPHERE

Without modification, the provided support code displays a lit sphere with no texture. The sphere is rendered using vertex buffer object with the vertices set up in client-side memory (RAM) and the vertex buffer object bound to client-side vertex attribute locations (`GL_VERTEX_ARRAY` and `GL_NORMAL_ARRAY`). Search for "TASK 1" in files `globe.cpp` and `objects.cpp` and add your code to complete this task. You may choose to use your own code from previous lab to render the sphere, instead of using the provided one.

Steps in applying texture:

1. Generate/obtain and pre-condition the texture image (using 3rd party software–see a sample list below).
2. Generate a texture object and bind it to OpenGL's 2D texture location (in `globe.cpp:init_texture()`).
3. Load the texture image from file and import it into memory (in `globe.cpp:read_texture()`).
4. Specify the image as texture using `glTexImage2D()` (in `globe.cpp:load_texture`) and specify texture-mapping options using `glTexParameter*()` and `glTexEnv*(decal|modulate|blend|replace)`.

5. While constructing the 3D/2D object, associate a set of texture coordinates with each vertex (in `objects.cpp:init_sphere()`).

Before you can load a texture, you need to generate a texture object and bind it to OpenGL's 2D texture location (`globe.cpp:init_texture()`).

To specify an image as a 2D texture, call `glTexImage2D()`. When the texture image resides as an array of pixels in client-side memory, the last parameter of this function must be a pointer to this array. In (`globe.cpp:pbo_alloc`) we use `malloc(3)` to allocate space for this array. The contents of this array and its data format are specified using the other parameters of `glTexImage2D()`.

In this lab, we use the Truevision TGA image format, a simple encoding of the color (or greyscale) information present in an image. If you want to use an image stored in a different format, you can first convert it to TGA, by using any graphics format conversion application (e.g., Apple's Preview, GIMP, Photoshop, or the `convert` program that is part of the ImageMagicK package that runs on Unix-based systems). The LTGA class (in `ltga.h` and `ltga.cpp`) has been provided so that you can easily work with TGA images. The function `read_texture()` loads a TGA file into memory. The code in `read_texture()` also shows how you can access various members of the LTGA class to obtain information about the image. The pixel array of the image itself can be obtained by calling the `LTGA::GetPixels()` method, which returns an array of unsigned byte/char.

**Specifying Texture and Setting Its Parameters.** Once the texture image is loaded into memory, you need to tell OpenGL about its characteristics. Texture images can be stretched to fit or tiled. This is specified using calls to `glTexParameter*()`. The following parameters are of use:

1. `GL_TEXTURE_MIN_FILTER`
2. `GL_TEXTURE_MAG_FILTER`
3. `GL_TEXTURE_WRAP_S`
4. `GL_TEXTURE_WRAP_T`

Also, once the texture is stretched or tiled, the texels can be copied to replace pixels on the surface, or can be blended or used to modulate the surface so that lighting and textures are combined. Use `glTexEnv*()` function on the `GL_TEXTURE_ENV`, **not** `GL_TEXTURE_2D`, target to specify the texture application mode you want. Consult the online reference pages for help.

**Texture Mapping.** Now we are ready to map coordinates of the texture image onto vertices of a 3D object, which is done at the time of object construction. The function `objects.cpp:init_sphere()` constructs a sphere of radius 1.0, centered at the origin. Associate texture coordinates with each vertex to map the Blue Marble world map (`BlueMarble2004-08.tga`) onto the spherical surface. Modern OpenGL no longer supports `GL_QUADS`, thus you *must* use `GL_TRIANGLE_STRIP` to model the sphere. Texel coordinates, $s$ and $t$, lie in $[0, 1]$ range in each dimension. To determine the texture-coordinates to associate with each vertex of the sphere, decompose the spherical surface into triangle strips, as you have done in the previous lab, and calculate what portion of the globe image should map to each triangle strip of the sphere.

In this task, we use client-side vertex array to pass the vertices to the vertex buffer object; we need a corresponding client-side texture-coordinates array to pass the texture coordinates. For best performance, vertex attributes are interleaved so that all the attributes of a vertex can be fetched at once. For good cache coherence, we also try to line up each vertex-attribute block on 32-byte alignment. The sample code allocates only one buffer object and put the vertex and texture-coordinates arrays interleaved on this buffer object. You need to enable the client-side vertex attribute location GL_TEXTURE_COORD_ARRAY using `glEnableClientState()` and let OpenGL know where the texture-coordinate array is located on the buffer object by calling `glTexCoordPointer()` When calling `glTexCoordPointer()`, the first element of the texture-coordinates array is after the first vertex position array, so you need to give the fuction an offset of `sizeof(XVec3f)`. Subsequent texture coordinates starts at byte offset `sizeof(XVec2f)` (the size of the texture coordinates) plus `sizeof(XVec3f)` (the size of the next vertex position coordinates), so set the `stride` argument of this function accordingly.

Once your texturing process is done, you can toggle texturing on and off using the keyboard shortcut '`t`'. Texturing is off by default.

## TASK 2: MIP MAPPING

Use the '`m`' and '`M`' keys to scale the sphere down and up respectively. You may notice aliasing of the texture when the sphere size is significantly smaller than the texture size, e.g., when the sphere has been shrunk twice. Use mipmapping to make texturing smoother. To autogenerate the mipmap pyramid, set `GL_GENERATE_MIPMAP` to `GL_TRUE` by calling `glTexParameteri()` *before* calling `glTexImage2D()` in `globe.cpp:load_texture()`.[1] To toggle mipmapping on and off, specify the appropriate `GL_TEXTURE_MIN_FILTER` parameter using `glParameteri()` in `globe.cpp:toggle_mipmapping()`. Pressing '`i`' then toggles mip mapping on and off. Search for `TASK 2` to complete this task. This is a 2-line task, don't overthink it!

## TASK 3: PIXEL BUFFER OBJECT

Pixel buffer object allows us to load texture image directly from file to graphics system memory, bypassing client-side memory. To use pixel buffer object, first generate and bind a pixel unpack buffer object in `globe.cpp:init_texture()`. Then replace the call to `malloc(3)` in `globe.cpp:pbo_alloc()` with code to allocate the required amount of graphics memory space and map the allocated memory to client-side address space and return the client-side address (pointer).

Then in `globe.cpp:load_texture()`, replace the last argument of `glTexImage2D()` with the byte offset from the start of the pixel buffer where the texture image is located (in this case, 0).

Back in `init_texture()`, once `read_texture()` returns and we're done with the mapped memory, release the mapping so that `glTexImage2D()` can have access to it. Once we're done loading the texture from the pbo to the texture object, i.e., upon return from `load_texture()`, delete the pbo.

Search for `TASK 3` to complete this task. This task involves a total of 7 lines of code. See the lecture notes on pixel-buffer object for the OpenGL APIs to call.

[1] Alternatively, you could use `glGenerateMipmap()` *after* the call to `glTexImage2D()` to autogenerate the mipmap pyramid, though this is usually used with framebuffer object, when the base level texture is modified and the mipmap needs to be regenerated.

## Task 4: Mapped Vertex Buffers

Now we apply what you've learned with mapped buffer in the previous task to vertex buffer, to hold the vertex, texture-coordinates, and element index arrays.[2] In `objects.cpp:init_sphere()`, replace the calls to `malloc(3)` with the allocation of the same amount of memory in graphics-system memory, then map it to client address space (search for "TASK 4").

After these variables are mapped to the graphics-system memory, when you populate these arrays, you're populating the graphics-system memory, by definition. So you do *not* need to copy data over to the graphics-system memory using `glBufferSubData()` anymore. Once the attribute array has been populated, release the mapped buffer so that OpenGL can have access to it. Similarly, convert the storage of `vertidx` in client memory to the use of mapped buffer object.

This task involves a total of 4 lines of code.

[2]In the previous task, instead of using mapped buffer to transfer texture image from file into PBO directly, we could have stored the texture image in client-memory as we did in Task 1 and copy it into the PBO using `glBufferSubData()`, which will not be as efficient.