Programming Assignment #4

CS 200, Fall 2017

Due Wednesday, October 4

Task #1: Make a Textured Mesh

For a textured mesh, we need to associate texture coordinates to each mesh vertex. We will do this by subclassing the `Mesh` class. The header file `TexturedMesh.h` contains the following declaration.

```
struct TexturedMesh : Mesh {
  virtual const Point* texcoordArray(void) = 0;
};
```

(the file `Mesh.h` has been included). Here `texcoordArray` simply returns a pointer to an array of points, where the i–th array entry gives the texture coordinates of the i–th mesh vertex. That is if `tm` is a `TexturedMesh` object, then

```
  Point P = tm.vertexArray()[i];
  Point uv = tm.texcoordArray()[i];
```

will retrieve the texture coordinates $(uv.x, uv.y)$ of vertex $P = (P.x, P.y)$.

Your task for this part of the assignment is to create your own textured mesh, and provide a bitmap image to use as the texture. As with the `MyMesh` class from assignment #2, your mesh cannot be a regular polygon and must have at least four triangular faces. You are welcome to use the mesh that you submitted for the `MyMesh` portion or assignment #2 as the underlying triangular mesh. The texture coordinates that you assign to the mesh vertices can give either a tiled or non–tiled texture to your mesh. However, the texture coordinates chosen must be reasonable. In particular there should be no visible folds along the boundaries of adjacent triangular faces of the mesh, and the texture should be displayed with the same aspect ratio as the bitmap image used for the texture. Note that you will find it easier if the bitmap image has equal width and height.

For this portion of the assignment, you should submit three files: (1) a header file named `MyTexturedMesh.h`, (2) an implementation file named `MyTexturedMesh.cpp`, and (3) a bitmap image file named `MyTexture.bmp`. You may include the header file `TexturedMesh.h` (which will include `Affine.h`) and any standard C++ header file. The bitmap image must be a 24 bit color image.

Warning. The test programs that I will give you for this assignment all use SDL2 to load the bitmap image. However, the resulting texture will (1) have the RGB byte order reversed for each pixel, and (2) be upside down! We will remedy these problems in the next assignment.

Task #2: Texture Rendering

In this part of the assignment you will use OpenGL to implement a class for rendering textured meshes. The class interface (both public and private) is contained in the file `TextureRender.h`, which I will provide.

```
class TextureRender {
  public:
    TextureRender(void);
    ~TextureRender(void);
    void clearFrame(const Hcoord& c);
    void loadTexture(unsigned char *rgbdata, int width, int height);
    void unloadTexture(void);
    void setModelToWorld(const Affine &M);
    void loadMesh(TexturedMesh &m);
    void unloadMesh(void);
    void displayFaces(void);
  private:
    GLuint program,
           texture_buffer,
           vertex_buffer,
           texcoord_buffer,
           face_buffer;
    GLint utransform,
          aposition,
          atexcoord;
    int mesh_face_count;
};
```

Note that only faces of the mesh can be rendered with this class. To render edges, you can use the `Render` class from assignment #2.

To implement the above interface, you are the use (without modification), the following fragment and vertex shaders. The fragment shader is

```
#version 130
uniform sampler2D usamp;
in vec2 vtexcoord;\
out vec4 frag_color;
void main(void) {
  frag_color = texture(usamp,vtexcoord);
}
```

The value of the *in* variable `vtexcoord` comes from the vertex shader, and gives the normalized texture coordinates associated to a pixel. This is done by the shader using an interpolation scheme — something we will cover later on in the semester. You will not be

setting the value of `vtexcoord`. The corresponding pixel color is determined by the bitmap image that is currently selected in GPU memory. The uniform variable `usamp` specifies how to *sample* the bitmap image: how to get a bitmap pixel color from normalized device coordinates. You will *not* be setting the value of `usamp` (it is set to a default value when it is declared). The vertex shader code is

```
#version 130
attribute vec3 position;
attribute vec2 texcoord;
uniform mat3 transform;
out vec2 vtexcoord;
void main() {
  vec3 v = transform * position;
  gl_Position = vec4(v.x,v.y,0,1);
  vtexcoord = texcoord;
}
```

Here the mesh vertices are to be associated with the attribute variable `position`. The uniform variable `transform` maps the mesh vertices from object space to OpenGL normalized device coordinates. Texture coordinates assigned to each mesh vertex are associated with the attribute variable `texcoord`.

`TextureRender()` — (constructor) creates a texture rendering object. Similar to the constructor of the `Render` class from assignment #3, several tasks need to be performed: (1) compile the fragment shader, (2) compile the vertex shader, (3) link the compiled shaders into a shader program, (4) get the locations of the shader uniform and attribute variables, and (5) enable the attributes (both the `position` and `texcoord` attributes). If the shaders fail to compile, or if the shader program fails to link, a standard `runtime_error` exception should be thrown.

`~TextureRender()` — (destructor) any resources allocated in the constructor should be deallocated here.

`clearFrame(c)` — clears the frame buffer using color c.

`loadTexture(rgbdata,width,height)` — uploads a texture to the GPU. The texture is in the form of a 24–bit RGB bitmap image of the specified pixel width and height. The value of `rgbdata` is a pointer to raw bitmap data. Data is assumed to be aligned on four byte boundaries. The texture should use (repeated) texture wrapping and use the nearest pixel operation for the minification and maxification functions.

`unloadTexture()` — removes the previously loaded texture from GPU memory.

`setModelToWorld(M)` — sets the value of `transform` in the vertex shader to M. The transformation M specifies how the vertices of the current mesh should map to normalized device coordinates.

`loadMesh(m)` — uploads the textured mesh data: the vertex array, the face list, and the texture coordinate array, to the GPU. The mesh m becomes the current mesh.

`unloadMesh()` — removes the previously loaded mesh from GPU memory.

`displayFaces()` — renders the current mesh using the currently loaded texture.

Your submission for this part of the assignment should consist of a single source file, which should be named `TextureRender.cpp`. In addition to `TextureRender.h`, you may include only the `stdexcept` standard header file. Note that `TextureRender.h` includes the header files `GL/glew.h`, `GL/gl.h`, `TexturedMesh.h`, and `Affine.h`.