Programming Assignment #6

CS 200, Fall 2024

Due Friday, November 1

Textured mesh interface

In semi–optional assignment #1 we introduced the `TexturedMesh` class, which extended the `Mesh` class to include texture coordinates for each vertex. In particular, the header file `TexturedMesh.h` contains the declaration:

```
namespace cs200 {

  struct TexturedMesh : Mesh {
    virtual const glm::vec2* texcoordArray(void) const = 0;
    virtual const char *textureFileName(void) const = 0;
  };

}
```

The function `texcoordArray` returns a pointer to an array of floating point pairs, 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

```
  glm::vec4 P = tm.vertexArray()[i];
  glm::vec2 uv = tm.texcoordArray()[i];
```

will retrieve the texture coordinates $(uv.x, uv.y)$ of vertex $P = (P.x, P.y, 0, 1)$. The function `textureFileName` returns the name of the bitmap image file that is to be used to texture the mesh.

For a simple example of using this interface, see the files `SquareTexturedMesh.h` and `SquareTexturedMesh.cpp` which implement a textured square mesh, from semi–optional assignment #1.

Your task: a textured mesh rendering engine

In this 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.

```
namespace cs200 {

  class TextureRender {
    public:
      TextureRender(void);
      ~TextureRender(void);
      static void clearFrame(const glm::vec4 &c);
      void loadTexture(unsigned char *rgbdata, int width, int height);
      void unloadTexture(void);
      void setTransform(const glm::mat4 &M);
      void loadMesh(const TexturedMesh &m);
      void unloadMesh(void);
      void displayFaces(void);
    private:
      GLuint program,
             texture_buffer,
             vertex_buffer,
             texcoord_buffer,
             face_buffer,
             vao;
      GLint utransform;
      int mesh_face_count;
  };

}
```

Note that only faces of the mesh can be rendered with this class. To render edges, you can use the `SolidRender` class from assignment #3.

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 graphics hardware 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
in vec4 position;
in vec2 texcoord;
uniform mat4 transform;
out vec2 vtexcoord;
void main() {
  gl_Position = transform * position;
  vtexcoord = texcoord;
}
```

Here the mesh vertices are to be associated with the attribute (in) 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 `SolidRender` 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, and (4) get the locations of the shader uniform variables. 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.

`setTransform(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. A vertex array object (VAO) for rendering the mesh should also be created. 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.

What to turn in

Your submission for this 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/gl.h` and `TexturedMesh.h`.