OpenGL 2

March 1, 2017

Computer Graphics

Texture mapping and animation

Please note that you will need to re-use your code from the first OpenGL assignment.

In this assignment you are given the opportunity to work with your own models and textures! These can be added in the subdirectories `models` and `textures`, respectively. By editing the `resources.qrc` file, these files will be included in your executable (so it really is a standalone application) and can be accessed with a colon (:) in front of their (relative) file names by the different file reader functions.

It is recommended to start with a simple model to test your code. Once your implementation works properly, a more complex model can be used.

1 Texture mapping

Using textures is an easy way to create more detail in objects. Instead of using a single color, an image is mapped to an object, such as a logo on a cube.

You may use the cube and/or the sphere model provided in the previous assignment, but you may also use your own models and textures or models/textures available online (see today's slides for a couple of links to repositories). Please use textures images of size 1024 pixels by 1024 pixels. Using sizes that are powers of 2 are generally preferred for performance reasons. You can use a tool like Blender (`https://www.blender.org/`) to export other models to the `.obj` file format. **Important!** Remember to set the export options to *triangulate faces*.

Basic method

Map a square *diffuse texture* (also referred to as a *color texture*) to the different sides of the cube.

1. You will need a GLuint to store a pointer to the texture (just like Buffer Objects) and a GLint to store the location of an extra uniform. Add these to the `mainview.h` header.

2. Next create a function `loadTexture(QString file, GLuint texPtr)` in the MainView class. This function will load the image file to the specified texture pointer.

3. OpenGL expects raw image data. Use the provided function `imageToBytes()` from Figure 2.1 in the **Appendix** to transform a QImage to a QVector of bytes (quint8).

4. Bind the texture using `glBindTexture(GL_TEXTURE_2D, texPtr)` so that we can set the following parameters with the function `glTexParameteri(GL_TEXTURE_2D, <Parameter Name>, <Parameter Value>)` where `<Parameter Name>` is one of the following:

   - `GL_TEXTURE_WRAP_S` and `GL_TEXTURE_WRAP_T` define the wrapping
   - `GL_TEXTURE_MIN_FILTER`, defines the minifying filter
   - `GL_TEXTURE_MAG_FILTER`, defines the magnifying filter

   See https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexParameter.xhtml for more info on possible values for `<Parameter Value>`.

5. With all the parameters set we can upload the data using the `glTexImage2D()` function. It takes the following parameters:

   (a) Target texture, `GL_TEXTURE_2D` for simple 2D textures.
   (b) Mipmap level, use 0 for now.
   (c) Internal data format, use `GL_RGBA8` for RGBA values, each 8 bits.
   (d) Texture width, width of the texture image.
   (e) Texture height, height of the texture image.
   (f) Border, always use 0 here.
   (g) Format, the format that will be used in your shader. Use `GL_RGBA`.
   (h) Type of the data in the pointer, use `GL_UNSIGNED_BYTE`
   (i) Pointer to the data to be uploaded, `<yourvector>.data()` where `<yourvector>` is the QVector of bytes you stored the image in using `imageToBytes()`.

6. **Important!** If you use mipmapping, call `glGenerateMipmap(GL_TEXTURE_2D)` after uploading the image or your texture might not show up.

7. In your initialization, use `glGenTextures(1,&texPtr)` to generate a pointer for your texture to be used with `loadTexture()`, where `texPtr` is your GLuint for the texture.

8. Create and allocate a buffer for the texture coordinates in the `createBuffers()` method. This is similar to the vertex position and normal buffers. Texture coordinates however, are 2D coordinates, so use QVector2D instead of QVector3D and change the number of elements accordingly in your call to `glVertexAttribPointer()`. Also use `glEnableVertexAttribArray(3)` to enable the texture coordinates in the shaders.

9. **Important!** Destroy all buffer and texture pointers in the destructor of the `MainView` class. Use `glDeleteTextures(1,&texPtr)` to free textures, where `texPtr` is your GLuint for the texture.

10. Next, make sure the vertex shader receives and passes the incoming texture coordinates to the fragment shader. In most cases you do not have to perform any transformations on the texture coordinates since they need to be interpolated directly. See Figure 2.2 in the **Appendix** if you have trouble implementing this.

11. In the fragment shader create a uniform of the type **sampler2D**. This will be used to retrieve the color (a **vec4**) from the texture at the provided coordinates. You also need to store the location of this uniform in `createShaderPrograms()`

12. In order to set the color of the fragment to the color of the texture, use the following line:
    `fColor = texture2D(samplerUniform, textureCoords)`, where `samplerUniform` is the name of your sampler uniform and `textureCoords` is a vec2 of your texture coordinates you retrieved from the vertex shader.
    Note: Some systems require `texture` instead of `texture2D`. It takes the same parameters.

13. In `paintGL()`, call `glActiveTexture(GL_TEXTURE0)` and bind the texture using `glBindTexture(GL_TEXTURE_2D, texPtr)` before your call to your drawing function. Replace `texPtr` with your GLuint for your texture.

14. All sampler2D uniforms are bound to `GL_TEXTURE0` by default. To specify to which texture they are bound, call
    `glUniform1i(<samplerUniformLoc>, <texture number>)` to specify the texture for each sampler (`glUniform1i(uniTex,1)` for `GL_TEXTURE1` for example). Also see the bonus assignment on using multiple textures.

15. Experiment with different wrapping and filter methods in your final application and describe what each one does in your `README` file.
    Tip! For experimenting with wrapping method try to multiply the vertex coordinates with (2,2) in the vertex shader to see the effects.

    An example of a textured cube is illustrated in Figure 1.1.

Bonus

Possible bonus features include:

- Use multiple texture maps (*diffuse* and *bump/normal* maps, or more advanced, *displacement* and *ambient occlusion* or *specular* maps). You need multiple calls to `glActiveTexture` and `glBindTexture` with a different texture number. Also make sure you set the right location of your sampler uniforms!

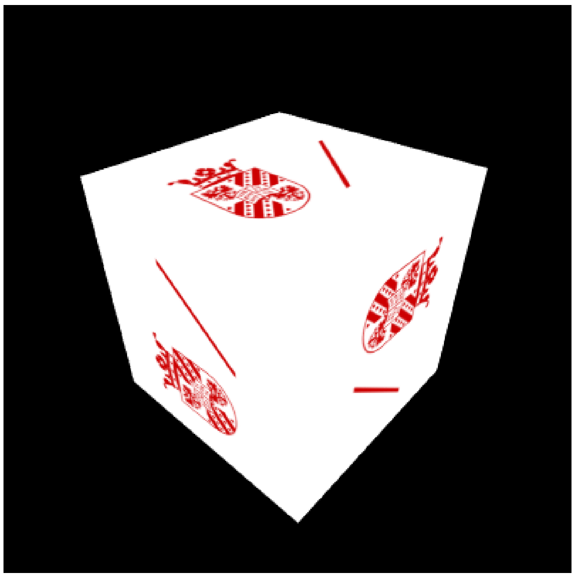

Figure 1.1: *A textured, rotated cube*

Optional

These steps are optional, but might improve your application a bit.

- Use *anisotropic filtering* on your texture. This will prevent blurring of the texture when your viewing vector is almost tangent to the surface of the textured object. This can be enabled by placing the following code snippet in `loadTexture()` before you upload your texture.

  ```
  GLfloat f;
  glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT,&f);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAX_ANISOTROPY_EXT,f);
  ```

- Implement the function `unitize()` in the `Model` class. As you might have encountered, the scale of a model is determined by the creator of the model. The provided cube has sides of length 2, whereas the sphere has a diameter of 50. It is also possible that the creator of the model did not center it around the origin of the model. By pre-scaling each model, you can make your program more robust and show each model at a similar scale and set the real size with a scaling transformation. Try to determine the size of the bounding box of your model and uniformly scale it to fit inside the unit cube (a cube with sides of length 1) and center it around the origin.

2 Animation

In this part we take a look at simple animation.

Basic method

The main assignment is to implement an animated textured scene. Use your knowledge of transformations from the first assignment to set the location of the models you want to render. Remember you can use multiple Vertex Array Objects in your program for different (groups of) models.

You are free to come up with your own animation idea — please send us an email (`rugcomputergraphics17+opengl@gmail.com`) or tell us during the practical to verify that your idea is not too simple or too hard.

An example could be to implement a (simple) solar system: a star (textured sphere) at the center and a couple of rotating planets (textured spheres) in an orbit (a planet rotates around its own axis and around the star).

Another method could be to use uniforms to specify some parameters (such as an increasing integer, keeping the time) to perform simulations in the shader. One such example could be a "wave shader" on a flat surface or maybe using a sound source to create a rythm following animation.

Be creative as this is a good starting point for your OpenGL app for the final competition!

A regular *animation loop* consists of two phases:

- Update positions, rotations and apply input modifiers (e.g. key strokes).
- Render the scene using the updated positions, rotations etc. This is also known as drawing a frame.

The loop is called in quick succession to simulate movement. To achieve smooth movement, try to reach a framerate of 60 frames per second.
In Qt you can use a QTimer to schedule an update to the positions and schedule a new call to render. Please check the Qt documentation to learn how to use a QTimer for this purpose. A timer already exists in the skeleton code, you only need to start the timer at an appropriate time.

Bonus

Implement an interactive animation (e.g. a simple game)! Please email us with your idea, so we can check whether it is suitable.

Deadline

See Nestor (*Time Schedule*). Details on how to submit your work can also be found on Nestor (*Lab Assignments*).

Assignment submission

Please use the following format:

- Main directory name: `Lastname1_Lastname2_OpenGL_2`, with the last names in alphabetical order, containing the following:
- Sub-directory named Code, containing the modified Qt framework (please do **NOT** include executables)
- Sub-directory named Screenshots with your screenshots or videos.
- README, a plain text file with a short description of the modifications/addings to the framework along with user instructions. We should not have to read your code to figure out how your application works!

The main directory and its contents should be compressed (resulting in a .zip or .tar.gz archive) which is the file that should be submitted (using the *Assignment Dropbox*). Example: the name of the file to be submitted associated with the first OpenGL assignement would, in our case, be `Kliffen_Talle_OpenGL_2.tar.gz`

Assessment

See Nestor (*Assessment & Rules* → Assignment assessment form).

Appendix

```
// Add this as a public member in mainview.h
// (you might need to include QImage)

QVector<quint8> imageToBytes(QImage image) {

// Add this to utility.cpp

QVector<quint8> MainView::imageToBytes(QImage image) {
// needed since (0,0) is bottom left in OpenGL
QImage im = image.mirrored();
QVector<quint8> pixelData;
pixelData.reserve(im.width()*im.height()*4);
quint8 r,g,b,a; // Use an unsigned byte
int i, j;
for (i = 0; i != im.height(); ++i) {
for (j = 0; j != im.width(); ++j) {
QRgb pixel = im.pixel(j,i);
// pixel is of format #AARRGGBB (in hexadecimal notation)
// so with bitshifting and binary AND you can get
// the values of the different components
r = (quint8)((pixel >> 16) & 0xFF); // Red component
g = (quint8)((pixel >> 8) & 0xFF);  // Green component
b = (quint8)(pixel & 0xFF);         // Blue component
a = (quint8)((pixel >> 24) & 0xFF); // Alpha component
// Add them to the Vector
pixelData.append(r);
pixelData.append(g);
pixelData.append(b);
pixelData.append(a);
}
}
return pixelData;
}
```

Figure 2.1: *Code snippet for converting a QImage to byte vector*

```
// vertshader.glsl

// Other inputs
layout(location=3) in vec2 textureCoords_in;

// Uniforms etc.

// Outputs from the vertex shader
out vec2 vertexTexCoords;

void main() {
// ... main functionality

vertexTexCoords = textureCoords_in;

// .. other main functiononality
}

// fragshader.glsl

in vec2 vertexTexCoords;
// They now can be used in the fragment shader
```

Figure 2.2: *Code snippet for passing incoming texture coordinates to the fragment shader*