Computer Graphics DT 3025
Lab 2

Stevan Tomic and Martin Magnusson, November 2015
Based on material by Mathias Broxvall

Lab 2

3D graphics, lighting and textures

In this lab you will experiment with how to draw 3D graphics with perspective and modelling transformations. You will learn how to perform lighting computations, give normals for the objects to be drawn and separate the *modelview* matrix from the *projection* matrix. Finally you will load some image files to be used as a 2D texture that is draped over the object you draw and experiment with letting this texture modify the different properties used in the lighting computations.

As you perform more and more advanced math operations you will need to perform operations such as matrix multiplications, matrix inversions and some other operations that might be hard for you to implement by hand.

To make the code more readable and easier to use, it is therefore recommended to use GLM (OpenGL Mathematics). This is an open-source headers-only implementation of the GLSL primitives (`vec2, vec3, mat4, ...`) in C++ that is released under an MIT license. You will find it at `http://glm.g-truc.net/`. **Go to this website and read the GLM manual before the lab.**

You will need to download these headers and include them in your project files (starting with exercise 2.3).

Exercise 2.1 - Performing perspective transformations.
Download, unpack and compile lab 2 from Blackboard. This will draw one quad over the full screen, but with different z values on the left and the right edge – but this is not yet visible since we do not have any perspective transformations.

Start by adding a `uniform mat4 projectionMatrix` to your vertex shader code. This will become the matrix to be used for your perspective transformation.

Create a corresponding 4×4 `GLfloat` array on the CPU side and initialise this matrix to:

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -a & -b \\ 0.0 & 0.0 & -1 & 0.0 \end{pmatrix}$$

where a, b are computed from the *near* and *far* clipping planes $n = 1, f = 10$ by

$$a = (f + n)/(f - n),$$

$$b = 2fn/(f - n).$$

Pass this matrix into the corresponding projection matrix above using the `glUniformMatrix4fv` function (refer to the tutorial given in the introduction section if you find any difficulties). Take care to note if you have specified your matrix in *row major* order or *column major* order.

Finally, perform the projection by multiplying this matrix in front of the position variable in your vertex shader. Note that GLSL has built-in matrix–vector and matrix–matrix multiplications.

However, since the quad above ranging from $z = -1$ to $z = 1$ lies ahead of your near clipping plane (at $z = -1$) you need to first move the quad into the right position. In the next exercise you will do this with a matrix – but for now you can simply subtract 4 from `inPosition.z` before you multiply it by the projection matrix.

If you have done everything correctly you should now see the quad above with a perspective projection.

Question 1.
Explain *why* the order of multiplication is important.

Exercise 2.2 - Performing 3D world transformations.
In this exercise you will continue from the previous exercise and use a *modelview* matrix to perform 3D transformations on your scene.

Continue from the code from the previous exercise and start by introducing a new variable called `modelviewMatrix` in your vertex shader.

Create again a corresponding array on the CPU side and initialise it to the following:

$$\begin{pmatrix} \cos\alpha & 0.0 & \sin\alpha & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ -\sin\alpha & 0.0 & \cos\alpha & \Delta z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

and use $\alpha =$ time and $\Delta z = -4.0$. This transformation matrix corresponds to a rotation around the y-axis by α radians followed by a translation of Δz units along the z axis.

Pass this array to your vertex shader and multiply it *in the proper order* with your projection matrix and the input vertices. *Note: you should no longer manually change the z-value of the input vertices to offset it along the z-axis since this is done by the matrix above.*

Continue by extending your CPU application to pass in 6 quads corresponding to the 6 different sides of a cube that is ± 1 units on each side. Let each side have a different colour. *Note: you should not try to re-use the vertices for the different sides on this exercise. You will need to give it 6x4 vertices.*

Your result should now be a rotating cube that has a flat uniform colour on each side (and the top/bottom sides are not visible). To make sure that you have added the top/bottom sides correctly try to introduce a y translation in your modelview matrix above.

Note. This exercise does not need to be included in the report or demonstrated.

Exercise 2.3 - Using GLM for your math operations.
Continue on your exercise above and change your modelview matrix on the CPU side to be a `glm::mat4` value (if you have not already done so). You will need to include `glm/glm.hpp` to do this.

Remove your hand-coded transformation and translation matrix and compute instead the modelview matrix as the product of one `glm::translate` that moves the object along the z-axis and one `glm::rotate` that rotates it around the y-axis as in the previous exercise.

Next, add one more rotation – this time by time $\times$ 0.4 along the x axis. Merge this with your modelview matrix above and pass only the composite matrix along to your vertex shader.

Your result should now be a rotating cube that has a flat uniform colour on each side and that rotates slowly along two axises at different speeds. This will let you see all of the cube given enough time.

This exercise *DOES* need to be reported and demonstrated.

Exercise 2.4 - Adding some lights to your scene.

In this exercise you will continue on your code above and add lighting computations to get a *"correct"* colour on each pixel of the cube.

Begin by adding a new vertex attribute `in vec4 inNormal` to your vertex shader. Add a new corresponding array on your CPU side and *give normals for each vertex in each quad* that is pointing out from the cube. Ie. for the quad that is facing the camera (at $z = -1$) you should have the normal $(0, 0, -1, 0)^{\mathrm{T}}$. Note especially that you should have $w = 0$ in these normals – this is since the normals are not a *point* but rather a *direction*.

Next, continue by adding two output variables `position` and `normal` to your vertex shader. The `position` variable should be used to pass along the transformed 3D coordinate to the fragment shader, ie. without a perspective projection. The `normal` variable should be used to pass along the transformed 3D normal to the fragment shader.

Finally, create a new uniform variable `light0pos` in the fragment shader, initialize a corresponding variable to be $(2, 1, 0, 1)^{\mathrm{T}}$ on the CPU side and pass it to the shaders. You are now ready to start computing colours.

Implement the Phong model on the GPU by the following formula:

$$C = C_a I_a + \sum_I \Big(C_d I_d (\mathbf{n} \cdot \boldsymbol{\omega}_{\mathrm{i},I}) + C_s I_s (\boldsymbol{\omega}_{\mathrm{r},I} \cdot \mathbf{v})^f \Big) \tag{1}$$

You can at this stage use the hardcoded constants:

$$C_a, C_d = (0.8, 0.8, 0.5)^{\mathrm{T}}$$
$$C_s = (1.0, 1.0, 1.0)^{\mathrm{T}}$$
$$I_a = (0.2, 0.2, 0.2)^{\mathrm{T}}$$
$$I_d = (0.8, 0.8, 0.8)^{\mathrm{T}}$$
$$I_s = (3.0, 3.0, 3.0)^{\mathrm{T}}$$
$$f = 20.0$$

For $\mathbf{n}$ you should use your `normal` variable that is interpolated from the vertex shader – but you need to *normalize* it. This can be done by the built-in function `normalize(...)`.

For $\boldsymbol{\omega}_{\mathrm{i},I}$ (the vector towards light source I) you should use the expression `light0pos - position` and *normalize* the result.

For $\mathbf{v}$ (the vector the eye) you should use just `position` since you know that the eye is in origo. Again, *normalize* the result.

Finally, for the reflection vector $\boldsymbol{\omega}_{\mathrm{r},I}$ you can use the built-in function `reflect`, or the expression below.

$$\boldsymbol{\omega}_{\mathrm{r}} = \boldsymbol{\omega}_{\mathrm{i}} - 2(\boldsymbol{\omega}_{\mathrm{i}} \cdot \mathbf{n})\mathbf{n}$$

If you do these steps correctly you should now have a correctly lit object.

This exercise does not need to be reported and demonstrated.

Exercise 2.5 - Texturing your objects.
In this exercise you should build on the code in your previous exercise 2.4 and add two textures that give the *ambient and diffuse* respectively the *specular* colour and intensity of the object.

Start by adding a `GLuint diffuseTexture` and `GLuint specularTexture` to your class `Resources`. Add the following two lines to the constructor of this class.

```
GLuint textures[2];
glGenTextures(2,&textures);
diffuseTexture=textures[0];
specularTexture=textures[1];
```

What this does is to create two new texture objects (with no actual content) that are referenced by the above two integers. (Note that you could also have used two calls to `glGenTextures(1, ...)` instead of the above code if you wanted to.)

Add the code to load some images to these textures. For this purpose you can use the function `loadTexture` that I have provided for you in `glUtils.cc`. Please take a look at this code and read it. The following code snipped should be added to your `Resources::reload` function.

```
loadTexture(diffuseTexture,"texture1.png", 1);
loadTexture(specularTexture,"texture2.png", 1);
```

Continue by adding `uniform sampler2D diffuseTexture, specularTexture` to your fragment shader. These variables are used as references to two different *texture units* on the graphics card. A texture unit is a dedicated hardware unit that can quickly read from and interpolate values from the texture memory in an efficient manner (and cache results between different invokations).

To let the fragment shader know which hardware unit to use you need to set it from the CPU. Use the CPU side of the code to set these two uniform variables to the values 0 respectively 1. Note that you use `glUniform1i` for this (you give sampler objects an integer number from $0 \ldots$ MAX_TEXTURE_UNITS).

Next, add the code to *bind* the two textures above to the first two available hardware texture units, numbered 0 and 1. You can do this by selecting the hardware texture unit to modify using `glActiveTexture(GL_TEXTURE0+i)` where i is the hardware unit to modify. (Note that `GL_TEXTUREi` for any i is also defined as the above expression). After selecting the hardware unit to modify, bind the selected texture to it by using:

```
glBindTexture(GL_TEXTURE_2D,r->diffuseTexture};
glEnable(GL_TEXTURE_2D);
```

You are now almost ready to start performing texturing inside your fragmentshader. But first you need to decide which part of the texture images should correspond to each fragment that you are drawing. To do this, add one more `vec2` vertex attribute `inTextureCoordinate` to your vertex shader. Let the vertex shader pass along this coordinate unchanged to the fragment shader in a new `out` variable. Add the corresponding array on your CPU side to initialize these attributes and give *reasonable* values in the range 0–1 for these texture coordinates so that the full textured image will appear on each side of your cube (after the next step).

Finally, you should add the fragment shader code that will actually fetch the data from the texture units and do something interesting with it. You can do this by adding:

```
vec4 Cd = texture(diffuseTexture, textureCoordinate);
vec4 Cs = texture(specularTexture, textureCoordinate);
```

Now, use the two values C_d and C_s above for you lighting computation.

You should now have textured cube where one texture is deciding the general colour of the cube and the second texture decides which parts of the cube have a specular reflection.

Try experimenting with the texture coordinates. What happens if you multiply the texture coordinates by a scalar constant (e.g. 2) or by a vector constant (e.g. `vec2(2.0, 1.0)`) before the texture fetch.

This exercise *DOES* need to be reported and demonstrated.

Exercise 2.6 - Experiment with other material models.
This exercise is required for grade 5, but optional otherwise.

Remove the textures from Exercise 2.5 so that it is easier to see the effects of a particular material model. Compare the Phong model from Exercise 2.4 with other methods for computing glossy (specular) reflection, such as Blinn–Phong or Cook–Torrance. You do not need to implement a BRDF function (as in Listing 14.9 in the book) but it is enough to change the “specular” term. Report on a systematic evaluation of the parameters for the models you have implemented and discuss the change in appearance.

Examination

To pass the lab you must hand in a report showing the screen-shots from your program and answering the questions. Lab 2 will be graded, so please, make sure that in your reports, you include (copy/paste) important parts of your source code used to solve related problems. Code should be well commented and explained!

For grade 3 it is required that

- the report is well structured and written in clear English, and that the report is complete (given instructions in the text above) with all questions answered,

- all students in a group (pair) also orally can describe and argue for the method and the algorithms that have been used in each task.

For grade 4 it is also required that

- the report clearly describe the problem statement and what has been done in a way that can be understood by someone who knows the course contents but has not done the labs,
- the report critically reflects on what has been learned in the lab,
- the program code contains clear comments to explain it and that names of functions, variables and classes are understandable,

For grade 5 it is, in addition, required that

- the report demonstrates the ability to methodologically experiment with algorithm and parameter choices,
- and contains a solution of the additional assignment 2.6.

Deadline November 24