

## CM0304 Graphics Labs

The main purpose of the lab sessions is to help you getting started with OpenGL and the practical coursework. The sources of the programs, etc. are available at the module web-site: <http://www.langbein.org/teaching/graphics/labs> and on blackboard. You can use these and the exercises below to start with the coursework. The web-site also provides links to additional resources on OpenGL.

### 1. Wireframe sphere

`wireframe` draws a simple wireframe sphere. You can rotate the scene by pressing the right mouse button and move the mouse; you can rotate the camera by pressing the left mouse button and move the mouse; you can move the camera position along the viewing direction by pressing the middle mouse button and moving the mouse forward/backward; you can exit the program by pressing escape.

#### (a) Build the program

Download, compile and run the C++ and/or Java wireframe program. The sources contain further instructions on how to compile them.

#### (b) Examine the sources

In the sources identify the functions responsible for drawing the sphere, handling user input, changing the camera, and initialising OpenGL.

#### (c) Display lists

Change the code such that instead of using display lists the CPU is sending the complete command set to the display processor each time the scene has to be redrawn.

#### (d) Draw a cylinder

Add a function for rendering a cylinder at the origin around the  $z$ -axis from  $z = 0$  to  $z = 1$  using only OpenGL primitives. Specify the polygons yourself and do not use a library function for drawing a cylinder! Then render the cylinder instead of the sphere with the program (do not delete the sphere function as you will need it again). Note that a cylinder can be generated by moving a circle along a straight line segment or moving a straight line segment along a circle.

#### (e) Using transformations

Now call both functions to render the sphere and the cylinder. As they are drawn at the origin they will overlap. You can move either one of these to another position using transformations: move the centre of the sphere to position (1,1,1) and rotate the cylinder such that it is around the  $x$ -axis.

Transformations are states of the pipeline and OpenGL has a single modelview transformation matrix modifying all primitives send through the pipeline in the same way until it is changed. To specify a translation use the function:

```
glTranslated(x_translation,y_translation,z_translations)
```

and for a rotation around an arbitrary axis through the origin use:

```
glRotated(rotation_angle,x_direction,y_direction,z_direction)
```

where the three direction parameters specify the direction of the rotation axis (for the  $y$  axis use the direction (0, 1, 0)).

These commands modify OpenGL's modelview transformation matrix by multiplying the existing matrix with a matrix representing the new transformation. E.g. by calling `glTranslated` twice the two translations will be applied in sequence to the primitives. It will not replace the first translation with the second one. To selectively apply different transformations to objects you have to store the current matrix on a stack (with `glPushMatrix()`), add a new transformation and send the primitives through the pipeline, and then recall the previous transformation by getting it back from the stack (with `glPopMatrix()`).

Note that the sequence of transformations matters! You get different results if you first apply a translation and then a rotation compared to applying a rotation and then a translation. In OpenGL the transformation you specify first is the one executed last; or

in other words the transformation specified last is the one applied first to the primitives (the new transformation matrix is multiplied from the right to the current matrix).

For the above task you have to apply the transformations in the following sequence:

```

Set transformation for all primitives
glPushMatrix()
    Set additional transformations for primitive 1
    Draw primitive 1
glPopMatrix()
glPushMatrix()
    Set additional transformations for primitive 2
    Draw primitive 2
glPopMatrix()

```

Note that the modelview matrix also contains the viewing transformation specifying the camera position. It is the first transformation specified as it is the last one which has to be applied to all primitives. If you reset the modelview matrix to the identity with `glLoadIdentity()` you will also lose the viewing transformation for the camera position.

Shaded below contains an example for using the matrix stack and transformations.

## 2. Shaded

`shaded` is a modification of wireframe which draws shaded objects. This means in addition to the geometry the material of the objects has to be specified, light sources have to be defined and the OpenGL initialisation has to be changed to allow us to draw shaded objects.

### (a) Material properties

Identify the changes required for shading objects in the sources. Change some of the material properties of the objects and the light properties and see how this affects the scene.

### (b) Transformations

Have a closer look at the function drawing the teapot. Here a translation and a rotation is used to position the teapot. To avoid changing the position of any other objects, the current modelview matrix of OpenGL is stored on a matrix stack before the transformations are added. After the teapot is drawn the original matrix is restored from the stack.

Change the the transformations for the teapot and see how this changes the position of the teapot. In particular consider changing the sequence of translations and rotations and see what happens.

### (c) Shading a cylinder

Now add the cylinder you added to the wireframe scene to this scene. Note that you have to compute surface normals for correct lighting and have to provide material properties. You may also have to move the cylinder to an appropriate position.

In general to find the surface normals at a point you have to compute the partial derivatives  $f_u(u, v) = \frac{\partial f(u, v)}{\partial u}$ ,  $f_v(u, v) = \frac{\partial f(u, v)}{\partial v}$  of the surface parametrisation  $f(u, v)$ , take the cross-product  $n(u, v) = f_u(u, v) \times f_v(u, v)$ , and then normalise this vector such that it has the length one:  $n^*(u, v) = \frac{n(u, v)}{\|n(u, v)\|}$ . If you have an explicit surface parametrisation, this gives the best normals. Note that for a cylinder is is easy to “guess” the answer (similar to the sphere).

Often it is sufficient to approximate surface normals for a vertex. Consider a vertex  $v_l$  in a polygon with vertices  $v_1, \dots, v_{n-1}$  in proper sequence for the OpenGL orientation rule. Then take the two neighbours  $v_{l+1}$  and  $v_{l-1}$  of  $v_l$  and compute  $d_0 = v_l - v_{l-1}$  and  $d_1 = v_{l+1} - v_l$ . An approximate normal direction at  $v_l$  is given by  $n_l = d_0 \times d_1$  (you still have to normalise its length!). But note that this may create visible artefacts.

## 3. Anim

`anim` is a further modification of the original source. More objects have been added, some of the objects are animated and some are transparent.

- (a) In order for the objects to be transparent blending has to be enabled. The fourth colour value (the alpha channel) in the material properties has to have a smaller value than 1.0 in order for a particular object to be transparent. In addition the sequence in which the objects are drawn is important.

Change the alpha channel values for some objects and see what happens.

- (b) When the bouncing ball is in front of the stable, transparent sphere you should notice that you cannot see through the ball to see the sphere and the teapot behind. This is due to the fact that the objects are drawn in the wrong sequence.

You may try to change the sequence in which the objects are drawn to change this. What is the problem of getting it right for all camera positions?

- (c) In order for the animation to work additional display lists are created. Before the display lists are executed to draw the object transformations, depending on a time parameter, change the position of the object. This allows us to draw the stable world first and then draw the animated objects separately.

Try to identify this in the sources and change some of the movements. You can also try to add another object and animate it.