Individual Assignment - Practice your OpenGL!

Andreas Bodén

Experimental setup

The assignment was performed on the computers in the computer lab and partly on my office computer running the same operating system and using a NVIDIA Quadro K620 GPU. Core-Profile OpenGL version 3.3 was used. For window creation and management FreeGLUT was used and for function pointer management GLAD was used. The GLM library was used for matrix and vector creation and manipulation.

Exercise 1 - Your first OpenGL application!

The task of the first exercise was just to get the skeleton code of the exercise running and opening an empty window. This was simply done by checking that all the required dependencies were installed and building the libraries followed by running the code. As expected, this opened a first green window.

Exercise 2 - Your (second) first OpenGL application!

The task of this exercise was to set up some additional function in the application to prepare for rendering more sophisticated scenes. One important part of this was to setup double buffering. This means that the rendering is first done to a buffer called a back buffer. This is distinct from the front buffer which is the one actually displayed on the screen. The reason for using a double buffering system is to prevent flickering of the screen caused by rendering multiple objects sequentially directly to the buffer displayed on the screen. Instead with a double buffered rendering. The object are first rendered sequentially to the back buffer and at the final step of the rendering loop, the front and back buffers are swapper, instantaneously displaying the fully rendered scene on the screen. The final step was to also setup functions to capture user input from the keyboard during runtime in order to enable future user interaction.

Exercise 3 - Defining the Program Object: Vertex and Fragment Shaders

In this step it was now time to create the vertex and fragment shaders and link them together to a program. A program defines a set of shaders to be used in the graphics pipeline and using glUseProgram we tell OpenGL which program (set of shaders) to currently use. At first, we create the most basic shaders that simply forward the vertex and fragment data along the pipeline. We then render a basic sphere on the screen using glutSolidSphere().

Exercise 4 - Creating the Model-View-Projection

In order to create a more versatile rendering pipeline that allows for user interaction with the rendered scene, we create what is called a Model-View-Projection matrix. It is created from the multiplication of three separate matrices each with its own interpretation. The idea is that each initial object is first defined in a coordinate system called the local coordinate system. In this space, coordinates only define a vertex' position in relation to other vertices of the same object. Each vertex' coordinates then needs to be transformed into a space common for all the objects to be rendered, this space is referred to as world space and the transformation from model space to world space is defined by the M matrix. We then need to take into account the position of the thought observer. To transform the coordinates from the world space into the space of the viewer (where the viewer is always at a fixed position) we define the V matrix. The final step, and the role of the P matrix, is to take into account the perspective projection needed to simulate how we are used to viewing the world. Combining

these three matrices gives us the final MVP matrix as MVP = P*V*M. Since we want to be able to change the MVP matrix during runtime e.g. to simulate motion of the viewer or objects, the MVP matrix needs to be defined in the shaders as a uniform variable. Uniform variable are variables within the shaders that can be accessed and changed from the main program running on the CPU. From the main program, we can then get the location of the uniform variable and change the value of it during runtime. To define the different transformation matrices, some handy functions of the GLM library is used such as glm::perspective(), glm::lookat() and glm::translate(). In this step a glutReshape function is defined to handles changes of the window size. Changing the window size requires changes to be made in the viewport and the projection matrix in order to retain a realistic aspect ratio in the final rendered scene.

Exercise 5 - Visualizing Particles

The final step was to visualize an ensemble of particles at the same time. In order create the feeling of depth in a simple way GL_BLEND was enabled and a gl_BlendFunc was defined. This enables making objects semi-transparent and blending colors from overlapping objects by setting the alpha channel of the RGBA vector to < 1. In order to change the position of the viewer, the keyboard arrow keys were captured and prompted a change of the view matrix using glm::rotate(). This created the feeling of moving the observer around the object/s when the arrows were pressed. Other button functions were also implemented to enable moving the observer closer to or further away from the objects by scaling the view matrix appropriately.

The last step was then to create an array of particle coordinates that can be updated before each rendering and then render one sphere at each particle position. The position update was done using a simple loop over the particles and for each particle generating a random vector that was used to update the position of the particles.

In the rendering function, another loop was created to render a corresponding number of spheres. Before rendering each sphere, the M matrix was redefined according to the position of that particular particle.

To make the simulation look more realistic, some inertia was added to the particles by making the new motion depend both on the previous motion and the new random motion vector. Larger particles were also assigned more inertia than smaller particles. The amount of inertia can be changed during runtime by scrolling the mouse.