DVA338 – Assignment 1

1. Transformations and Simple Scene Rendering

In this assignment, you will write a program that renders triangle meshes using OpenGL. It is recommended that you use the start-up code available on the course home page, which among other things includes a simple triangle mesh data structure, some triangle models, a small algebra package, and initializations. Run the program and make sure you understand how it works.

OpenGL has evolved dramatically during recent years to increase flexibility and to better match the power of modern GPUs. Newer versions of OpenGL still support the traditional OpenGL API in a so called "compatibility profile", but it is recommended that you use modern style OpenGL (core profile, version 4.0 or later) in these assignments.

1.1 Understand the Code

In the *main.cpp* you might want to take a special look at the `display()` and `renderMesh()` functions. That is where most of the magic happens.

The `display()` function is called by OpenGL every frame. Its job is to prepare the transformation matrices and send them to OpenGL pipeline with the mesh to perform the transformations. Here we need to calculate a minimum of 2 matrices. These are called matrices **V** and **P** and at the moment we have given constant values to both of them.

Matrix **V** is the *View matrix* and represents the transformation that moves the object to the camera view. Part of your job in this assignment will be to calculate this matrix, given the location and direction of the camera.

Matrix **P** is the projection matrix and is responsible for projecting the 3D scene to a 2D space. Later you will need to create this matrix too.

The `renderMesh()` function is responsible for sending the mesh vertices and the transformation matrix to the OpenGL pipeline. At the moment you do not need to understand the code much. Just notice that here is where we perform our local to world transformations. Because this is something specific to each object.

Also take a look at *mesh.cpp*, this is where we store the mesh information from the input files into memory structures of our own. Try to investigate this code and also the mesh files to get a grasp of mesh structure. Notice that here we also have some part marked for this assignment. This is where we need to calculate the surface normal given all the faces.

1.2 Viewing Transformation

Extend the algebra module with your own functions for creating translation, scaling, and rotation matrices. Use these functions to define your view matrix. Let the camera location in the world be defined by three rotation angles α, β, γ and a position $c = (c_x, c_y, c_z)$. Then the transformation from world to camera coordinates can be defined as:

$$V = R_z(-\gamma)R_y(-\beta)R_x(-\alpha)T(-c)$$

To start using this camera transform during rendering, you must of course pass it to the vertex shader. Also, make it possible to interactively move the camera to demonstrate that the camera transform works as expected. For example, use the keys x, X, y, Y, z, Z to change the position of the camera along the world's coordinate axes, and the keys i, I, j, J, k, K to rotate the camera in the world coordinate system with respect to the x-, y-, and z-axes, respectively. Finally, note that although this rudimentary camera model works, the navigation is not very convenient. See suggestions for an improved camera model in Section 1.6.

1.3 Perspective and Parallel Projection

Extend the algebra module with your own functions for creating suitable perspective and parallel projection matrices. To demonstrate that the projection matrices created with your functions work as expected during rendering, make it possible for the user to interactively vary both the projection type and some of the parameters that define the projection matrices.

1.4 Generating Vertex Normals

Calculate and store vertex normals for all triangle meshes during program initialization. These normals will be used later in these assignments to realize simple shading models. However, you should be able to see whether the normals seem correct or not, since the provided shader program renders them as if they were RGB colours over the surfaces of the meshes. To see this effect, change the polygon mode to get the triangles filled. You also need to enable the z-buffer for proper hidden surface elimination. You can read about Polygon mode and Z-Buffer on the net.

1.5 Scene Composition

The geometry in the included 3D models are defined in their own local coordinate system. When several models are put into a virtual scene, we would like to place the objects at specific locations of our own choice. Make it possible to specify the position, orientation, and size of several models in your scene.

For each model, specify and store parameters for scaling $s = (s_x, s_y, s_z)$, rotation $r = (r_x, r_y, r_z)$, and translation $t = (t_x, t_y, t_z)$. Note that the transformation of the models, using these parameters, should take place when the objects are rendered, and not initially when the models are stored in the triangle mesh data structure. Each time a model is rendered, you compute the model's composite transformation matrix **W** directly from the scale, rotation, and translation parameters:

$$W = T(t)R_x(r_x)R_y(r_y)R_z(r_z)S(s)$$

Then the combined transformation matrix **M = VW** can be used during rendering to transform the vertices of a model directly from local to view coordinates. Finally, make it possible to interactively animate the models in your scene independently from each other by allowing the user to select a model and modify its transformation parameters.

1.6 Extra: Navigation

Implement a camera model which makes it possible for the user to interactively navigate along the principal axes of the current view, rather than along the axes of the world coordinate system. Support a full 6-DOF interactive camera model, i.e., it should be possible to move the camera forward/backwards, up/down, left/right as well as rotating the camera using pitch, yaw, and roll rotations.

Hint: Define the camera coordinate system using a camera position, ***c***, a normalized view direction vector, ***v***, and a normalized up vector, ***u***. Note that the third axis of the camera coordinate system is implicitly given as the cross product of the view direction and the up vector. To render the current view, you also need to create a `lookAt()` function that, given ***c***, ***v***, and ***u***, sets the view matrix appropriately.

To move the camera, translate and rotate the camera parameters appropriately. For example, to move the camera along the line of sight, we can simply compute a new camera position as $\boldsymbol{c_0 = c+v}$, which will move the camera coordinate system forward 1 length unit. Or to rotate the camera sideways (yaw), we can simply rotate the view direction, ***v***, a few degrees using ***u*** as the rotation axis. For this operation, you need to define a function that generates a matrix for rotation about an arbitrary axis.

1.7 Extra: Loading 3D Models from Disk

Create a function for loading polygon meshes defined in the OBJ file format into you scene data structure. Feel free to ignore features defined in the OBJ files which are not related to simple polygon meshes.

1.8 Extra: Use the GLM Library

Use the GLM (OpenGL Mathematics) library instead of the small algebra module you have used so far, i.e, remove all dependencies of *algebra.h* and *algebra.cpp* and use the corresponding GLM functions instead. Test that all the features you have implemented still work as expected together.