Umeå University
Department of Computing Science
Computer Graphics and Visualization

Pedher Johansson, pedher@cs.umu.se
Stefan Johansson, stefanj@cs.umu.se

3D Studio – Part 2

Projections and camera

Purpose

In this part we will extend our software from Part 1 to handle different projections using the camera model. The aim is to get a good understanding of the different transformations a single vertex must undergo before it is shown on the screen, in particularly the different aspects of the camera model. We also use a GUI.

Specification

Extend the code from Part 1 such that, in addition to the model matrix *M*, the shader also receives a view matrix *V*, and a projection matrix *P*:

```
gl_Position = P*V*M*vPosition
```

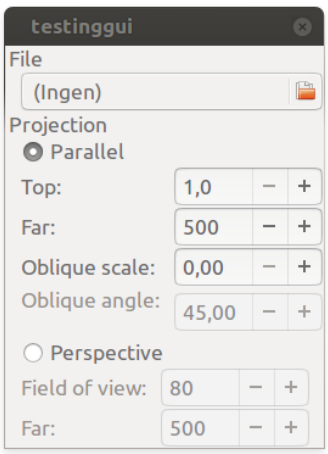
At start, the camera should be located at (0.0, 0.0, 2.0, 1.0) with the reference point at (0.0, 0.0, 0.0, 1.0) and an up-vector (0.0, 1.0, 0.0, 0.0). This is used to define the *V* matrix. Using the GUI, you get values for either *top* and *far,* or *FOV* (Field of view) and *far*. Use these values together with a suitable *near* to define *left* and *bottom*. This is then used to define your frustum for *P*. The *T* matrix can be computed just like in Part 1. Opening an OBJ-file should now, however, be done in the GUI.

Aspect ratio

Set the viewport as the entire window. Adjust left and right in the frustum so that the aspect ration of the front plane of the frustum equals the one of the window.

Graphical User Interface

Under Resources on Cambro, there are some code you may use to get a basic GUI functionality going. This code can be used to create a window that looks like in the picture to the right. You can however also write your own GUI.



There are two options for the GUI. If you have chosen GLFW and C then download ass2*gui-gtk.zip*, which contains a GTK+ 3.0 GUI created with *Glade*. Otherwise, if you use Qt download ass2*gui-qt.zip*, which contains the Qt version of the GUI.

In the GTK+ version, the GUI is specified in the XML-file `ass2gui.glade` which can be loaded in the `initGuiWindow` function. The GUI is handled by a number of callbacks. Stubs for the callback functions are found in `ass2gui.c`. There are also

a number of functions used to initialize the GUI and to set initial values (if you like to have other default values).

In the Qt version, the GUI is specified in the XML-file `ass3widget.ui` and the callback functions in `ass2widget.cpp`.

GUI functionality

From the GUI the projection type and parameters are set.

Parallel	Use a *parallel* projection (default orthographic).
- Top	Specifies the *top* value in the frustum
- Far	Specifies the *far* value in the frustum
- Oblique scale	Ratio (a) between z- and xy-plane, [0,1] (default 0.0).
- Oblique angle	*Degree* (ϕ) of obliqueness (angle between DOP and PP), [15˚, 75˚] (default 45˚).
Perspective	Use a *perspective* projection.
- Field of view	Change the angle of *field of view* (angle between top and bottom planes), [160˚, 20˚] (default 80˚).
- Far	Specifies the *far* value in the frustum

For parallel projection the projection matrix $P = M_{orth}*S*T*H(\phi)$, where

$$H(\phi)=\begin{bmatrix} 1 & 0 & a*\cos(\phi) & 0 \\ 0 & 1 & a*\sin(\phi) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the product $S*T$ is the scaling and translation performed on the frustum to get it to NDC. However, we do not need to explicitly multiply with the M_{orth} matrix, hence we get $P=S*T*H(\phi)$. Note that if $a = 0$ we have *orthographic* projection and if $0 < a \leq 1$ an *oblique* projection. Especially, an oblique projection with $a = 0.5$ is called *cabinet* projection and with $a = 1$ *cavalier* projection.

Camera functionality

The camera should be possible to rotate around the camera's x- and y-axis using the mouse and translated using the following keys:

Up (W)	Moves p_0 and p_{ref} relative the camera's positive y-axis.
Down (S)	Moves p_0 and p_{ref} relative the camera's negative y-axis.
Right (A)	Moves p_0 and p_{ref} relative the camera's positive x-axis.
Left (D)	Moves p_0 and p_{ref} relative the camera's negative x-axis.
Forward (Z)	Moves p_0 and p_{ref} relative the camera's negative(!) z-axis.
Backwards (X)	Moves p_0 and p_{ref} relative the camera's positive z-axis.

- When rotating the camera up or down, the look-at-point should be rotated

around the camera's x-axis. When rotating left or right, the look-at-point should be rotated around the camera's y-axis.

- Each time the camera moves, a new look-at-point should be computed such that the look-at-point always is at certain distance from the camera. Notice the following:

$$V = M_{wc} T(-p_0)$$
$$V^{-1} = T(-p_0)^{-1} M_{wc}^{-1} = T(p_0) M_{wc}^T$$

$$p^{(camera)} = Vp^{(world)}$$
$$p^{(world)} = V^{-1} p^{(camera)}$$

$$V^{-1} p_0^{(camera)} = p^{(world)}$$
$$\Rightarrow V^{-1}[0\,0\,0\,1]^T = p_0^{(world)}$$
$$\Rightarrow V^{-1}[0\,0\,-d\,1]^T = p_{ref}^{(world)}$$
$$\Rightarrow V^{-1}[0\,1\,0\,0]^T = V^{(world)}$$

That is, an update of the camera position, reference point, or up-vector is easy to express in camera coordinates, which can be multiplied with V^{-1} to get them back to world coordinates. For example, a move of the camera along it's positive y (up) can be expressed as below, resulting in an updated camera position and reference point, that can be used to update V.

$$p'^{(world)}_0 \leftarrow V^{-1}[0\,d\,0\,1]^T$$
$$p'^{(world)}_{ref} \leftarrow V^{-1}[0\,d\,-1\,1]^T$$
$$Up' \leftarrow Up$$
$$V' \leftarrow view(p'_0, p'_{ref}, Up')$$

Instructions

This is an individual assignment. The code should follow good programming practice and be compliant and executable on the computers in MA416. If you are using any libraries that are not pre-installed, include them in the separate subfolders ./lib and ./include and set appropriate parameters in the Makefile. No written report is required.

Due date is at latest December 1, 2016, 8.15. Time for demonstration is provided December 1, 8:15-12:00, in MA416, if nothing else is agreed upon.