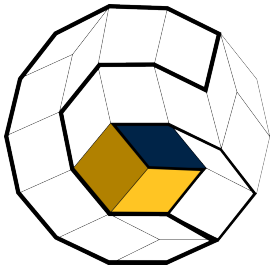
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Lorenz Rogge (rogge@cg.tu-bs.de)



May 3, 2012

Echtzeit Computergraphik SS 2012
Assignment 4

Present your solution to this exercise on Monday, May 14th, 2012.

The exercises are going to take place in the CIP pool, room G40 in Mühlenpfordstraße 23. Please make sure your solutions compile and run on the CIP pool computers. Note that you need a y-number, which can be obtained at the Gauß-IT-Zentrum, to use the computers. If for some reason you are not able to attend the exercise, you may send your solution to rogge@cg.tu-bs.de instead.

In this assignment you will implement a camera controller, allowing to freely move a OpenGL camera within your scene using your mouse and keyboard. To achieve this, you will need to correctly set up projection and view matrices, as introduced in the last lecture. Also you will learn how to use multiple viewports allowing to display two different renderings side-by-side in a single window.

4.1 Implement a Camera Controller (20 Punkte)

Implement the missing parts of `CameraController.cpp`. The class has already a basic get-/set-interface and is already fully intergrated into the main program of `Ex04.cpp`. Simply complete the missing parts of the methods `updateMousePos(...)`, `updateMouseBtn(...)`, `move(...)`, `getProjectionMat(void)`, and `getModelViewMat(void)`.
The first two methods handle the mouse input and allow to control the camera's viewing direction. The camera controller uses two angular values θ and ϕ to rotate the viewing direction d around the y-axis in world space (θ) and then around the rotated x-axis (ϕ). These values can be set according to the mouse cursors movement.
The method `move(...)` allows to move the camera's origin around. Forward and backward motion goes along the viewing direction of the camera, while sideways motion is perpendicular to the viewing direction and stays within the x-z-plane. Adjust the member `mCameraPosition` according to the given motion direction.
The two latter methods return the matrices used by the OpenGL pipeline to transform the geometry according to the camera's parameters. The projection matrix is used to transform everything from camera space to the normalized device space being a unit cube. The modelview matrix is used to transform the given geometry from world space to camera space. Compute the matrices using the available parameters like opening angle, camera position and viewing angles and return them as `glm::mat4`.
For a better understanding of the camera's parameters have a look at figure 1.
The `CameraController` is already intergrated into the provided `Ex04.cpp` and is fully connected to the keyboard and mouse callback functions of *GLUT*.
Using the mouse you can control the viewing direction by pressing the left mouse button and moving the mouse. The camera position can be controlled using the *W S A D* keys of your keyboard. You may also control other camera parameters like near and far clipping plane (*R/F, T/G*) and the camera opening angle (*Z/H*).

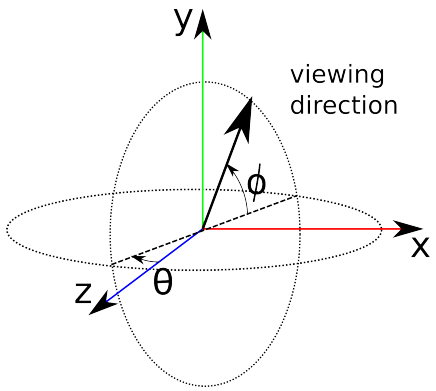


Figure 1: Used angles describing a viewing direction. θ corresponds to a rotation in the x-z-plane, while ϕ describes the tilt around the rotated x-axis.

4.2 Use multiple Viewports (20 Points)

In this task we want to visualize the camera's frustum in a second viewport.
To setup a viewport use `glViewport(x, y, w, h)` to define the lower left corner of the viewport within your rendering window and the width and height of the viewport. Since you need to render two separate views, create a first viewport on the left half of your window, which is initially defined to be 1024×512 pixels wide. Use this left half to render original camera view (using `cameraView` as `CameraController`). Render `scene.obj` which is already imported within `initScene`.
For the second viewport you need to do a little more work. To create another view of the scene, observing the scene *and* the camera and its frustum, we use another `CameraController` called `sceneView`. Use this camera to render the `scene.obj` from another perspective. To render the camera position of the left viewport, you need to get the information about how the camera is placed relative to the scene it was rendering. Fortunately we do have the opposite of this available, being the modelview matrix of that particular camera. It describes the scene relative to the camera origin. Thus, using the inverse of this camera matrix is exactly what we need. Invert the modelview matrix of `cameraView` and apply it the the modelview transformation of the right viewport. This transforms everything that we render now relative to the camera position of the left viewport. Simply render the already imported `camera.obj` at the origin.
To render the camera's frustum in correct shape, we somehow need to reconstruct it from the camera parameters. This is not as complicated as one my think, again. Remember that the perspective matrix of a camera transforms everything from camera space into normalized device space - which is a cube from $(-1, -1, -1)^T$ to $(1, 1, 1)^T$. The code in `Ex04.cpp` already provides a vertex array object `cubeVAO` representing such a cube. Transform this cube into the proper frustum shape and render it as `GL_LINES`. Now you have rendered two views using two camera. The second view also displays the other view's camera position and frustum. Your resulting window should look like figure 2.

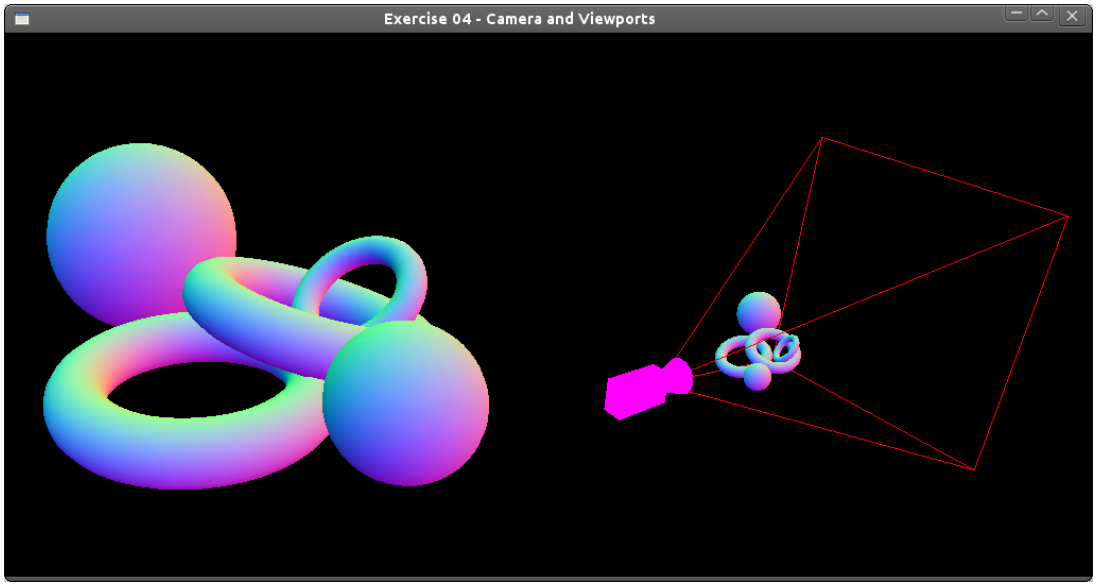


Figure 2: Example view with multiple viewports and a camera frustum visualization.