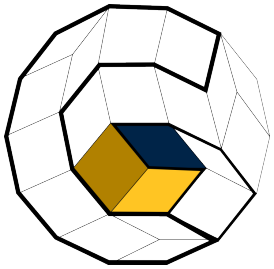
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Matthias Überheide

November 23, 2014



Real-time Computer Graphics WS 14/15
Assignment 4

Present your solution to this exercise on Monday, December 1st, 2014.

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 ueberheide@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 three different renderings side-by-side in a single window. You will render the perspective of a camera into the first view, a visualization of the camera frustum in world space into the second, and a visualization of the canonical volume of OpenGL into the third view.
For this excercise you need the AntTweakBar library. This library enables a simple and efficient setup of a graphical user interface (GUI). First download the AntTweakBar library from `http://anttweakbar.sourceforge.net/doc/tools:anttweakbar:download`, compile it on your computer and make sure that `AntTweakBar.h` and the corresponding library can be found correctly by the project.
Next you should check the resolution in the decleration part of `Ex04.cpp`. Currently the program has window size of 1736 x 512 (3*512+200 x 512). If your display has a smaller resolution you have to reduce the values.

4.1 Implement a Camera Controller (30 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. Up and down motion goes along the y-axis. 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 (canonical volume, abbreviated with CV in the code). 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 in the way you learned in the lecture 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 integrated 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 $W\ S\ A\ D\ R\ F$ keys of your keyboard. You may also control other camera parameters like near and far clipping plane and the camera opening angle using the GUI. The respective parameters are connected already in the `main()` function in `Ex04.cpp`. There are two `CameraController`s defined: The first one (`cameraView`) represents the camera shooting the scene. The second camera (`sceneView`) is used to visualize the viewing frustum of the first camera.

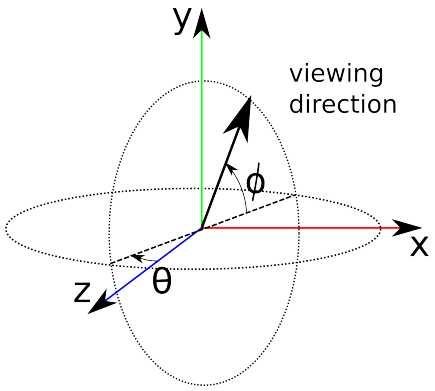


Figure 1: Used angles describing a viewing direction. θ corresponds to a rotation in the x-z-plane, while ϕ describes the tilt around the rotated x-axis.

4.2 Use multiple Viewports (70 Points)

4.2a In the first (most left) viewport you should render the scene from camera perspective view. Use the left third to render the original camera view (using `cameraView` as `CameraController`). Your task is to complete the `renderCameraView()` method which is called in the method `updateGL()`. Before that, you have to complete the method `setViewport(VIEW)`. Each viewport should take a third of the available space, subtracted by the amount of the GUI width. The method has to call the function `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. Now you can call `setViewport(...)` (in this case the parameter should be `CAMERA_VIEW`) in the render function to set the viewport correctly. In the vertex shader there is an input variable `cv_transform`, which is used in task 4.3c to transform a vertex after projection has been performed. For the first view this matrix should be identity so that the post-transformation has no effect. Create an identity matrix and set it to the corresponding variable in the shader. Last but not least you have to render the scene. The scene is already defined within `initScene`.

4.2b For the second viewport you need to do a little more work. We want to visualize the camera's frustum in world space in the second viewport. For visualization import the model `camera.obj`. This model will be rendered later additionally to the scene. In the render function of this view, `renderCameraSpaceVisualization()`, for viewport estimation again use `setViewport(...)`, (with `WORLD_VIEW`). To create a second 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 from another perspective. 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 first 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 might think. 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 vertices and indices for such a cube. Complete the creation of the cube (VAO, VBO, IBO) in `initScene(...)`, transform this cube into the proper frustum shape and render it as `GL_LINES`.
Now you have rendered two views using two cameras. The second view displays the first view's camera position and frustum.

4.2c Next you will visualize the normalized render cube containing the visible transformed geometry in its volume. This is done in the following way. You will transform the vertices in the vertex shader after the projection has been performed with an additional matrix `cv_transform`. As you know from the lecture the projection matrix brings all the scene geometry seen from the camera perspective to the inside of the canonical volume, a cube also called clipping space. The matrix `cv_transform` performs scaling and rotation of this cube. Scaling down the cube is necessary for our visualization since clipping against the normalized cube would produce a partly incomplete scene.

In the render function of this view, `renderCanonicalVolumeVisualization()`, specify the viewport (`CV_VIEW`). Now set the projection value of the `cameraView` camera controller to the projection matrix in the shader program. Now, set the matrix `cv_transform` to the corresponding value in shader.
Set the model view matrix of the model view matrix stack, which is used in the `renderScene()` function. Since the clipping of the scene does not look as wanted anymore, clipping against the canonical volume has to be performed manually. This is already done in the vertex shader, but not yet activated. Turn on custom clipping in OpenGL by calling `glEnable()` with parameter `GL_CLIP_DISTANCE0` and set the variable `cv_scale` used in the GUI as `clip_plane_distance` in the shader program. This sets the clipping planes to the correct distance. Now render the scene. The scene should be correctly clipped against the additionally introduced clipping space. Next, you want to visualize the borders of the canonical volume. You should therefore disable custom clipping again. Then set the projection and model view matrix to identity - since we defined the vertices of the canonical volume in the normalized coordinates - and render the `cubeVAO` as `GL_LINES`. `cv_transform` has not to be changed and will transform the lines in the same way as the scene previously. Next, we want to visualize the side of the cube from where the camera is looking. For the triangles of this side create another VAO, VBO, IBO in `initScene(...)` using the already available vertex information of the cube, but also using a new array with six indices forming a plane with two triangles. Render the triangles in the `renderCanonicalVolumeVisualization()` function. Set the color by overriding the color in the shader program as described in the code. Last but not least, render the `camera.obj`. Scale the model down to 0.1, translate and rotate it to the appropriate position using the model view matrix before rendering. Your resulting window should look like Figure 2. You can freely play around with the parameters in the GUI to better understand the usage of the projection matrix.

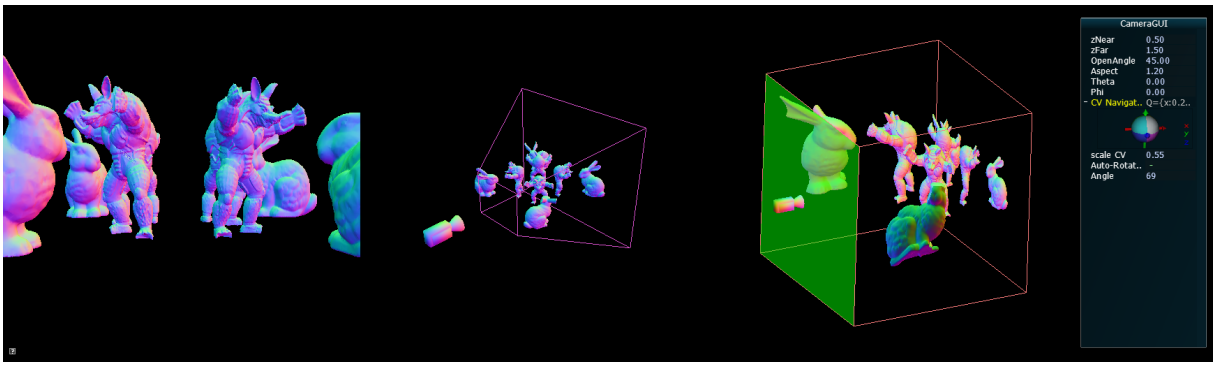


Figure 2: Example view with multiple viewports and a camera frustum visualization.