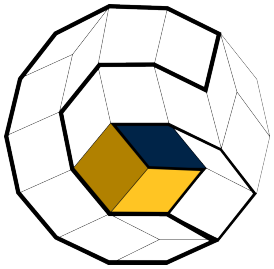
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Lorenz Rogge (rogge@cg.tu-bs.de)

April 27, 2012



Echtzeit Computergraphik SS 2012
Assignment 3

Present your solution to this exercise on Monday, May 7th, 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 make use of simple affine transformations as discussed in the last lecture. To be able to render more objects than just the Stanford Bunny defined as a header file and to be more flexible, you will also create a simple structure to load geometry from separate files and render it conveniently.

3.1 Load a 3D mesh from file (30 Punkte)

Implement the missing method `loadObjFile(...)` in `ObjLoader.cpp`. The purpose of this method is, to open a Wavefront OBJ file (`http://en.wikipedia.org/wiki/Wavefront_.obj_file`) containing vertex data for a 3D mesh object and to import data into a vertex list, a normal list and a list of vertex indices used to form faces (defined as triangles). To render a imported mesh, you will have to create VAO, VBO and IBO structures like you did in the last exercise. Use the imported lists to create all required structures to render the loaded object properly.
To implement a proper OBJ-Parser, do as follows:

- Access the file given by the parameter `fileName` and read it in line by line.
- Scan each line for the key defining the contents of the line. Until now only the keys `"v"`, `"vn"` and `"f"` are of interest, indicating a vertex, a vertex normal, resp. a face definition.
- Use a local `MeshData` struct as a temporary container for the imported data.
- For each vertex and vertex normal definition, read in the three vector values as `GLfloat` and buffer them in the designated array within your `MeshData` container. It will be used when the values are uploaded to a vertex buffer object.
- For each face definition, read in the vertex and normal indices used by the face and also store them in your `MeshData` container. Be sure to use correct indices, since rendering with IBOs starts indexing at 0, the Wavefront OBJ format however does not.

Now that you have parsed the OBJ file properly and imported its data into a `MeshData` container, use this data to create a `MeshObj`. Implement the missing parts of `MeshObj::setData` and `MeshObj::render` to render the imported geometry data as vertex array object using a single frame buffer object per vertex attribute. Since the provided data in `meshes/scene.obj` contains information about vertex positions and vertex normals, you will need two FBO structures. Also an additional index buffer object is required for indexed rendering of the imported data. Create all neccessary structures, upload the data and configure the layout of it.

3.2 Organize geometry in a scene graph structure (10 Points)

Using vertex array objects like the `bunnyVAO` or the more flexible `MeshObj` structure allows to render multiple instances of the same data. Using affine transformations and a scene graph structure these instances can be rendered at arbitrary positions in space.
In this task you will have to render a 5×5 grid of geomerty instances. When rendering the grid alternate the objects to be rendered between the bunny from the last exercise and the geometry loaded from the given file `meshes/scene.obj`.
Try to create the following:

- Render a 5×5 grid of geometry instances ($0.5LE \times 0.5LE$ in world space coordinates).
- Rotate this grid clockwise around the y-Axis about `rotAngle` degrees.
- Alternately render the data from `bunny.h` or `scene.obj`.
- Since `scene.obj` is much larger than the Stanford bunny, scale it down by a factor of 20 before rendering.
- Let the bunnies rotate counterclockwise about the same angle `rotAngle` and aroudn the same axis as the grid.

Make use of `push(...)` and `pop()` using the model view matrix stack `glm_ModelViewMatrix`. This allows to restore previous model view configurations, after having changed it to render an object relatively transformed to a parent object.
Your resulting animated grid should look like the one in figure 1.

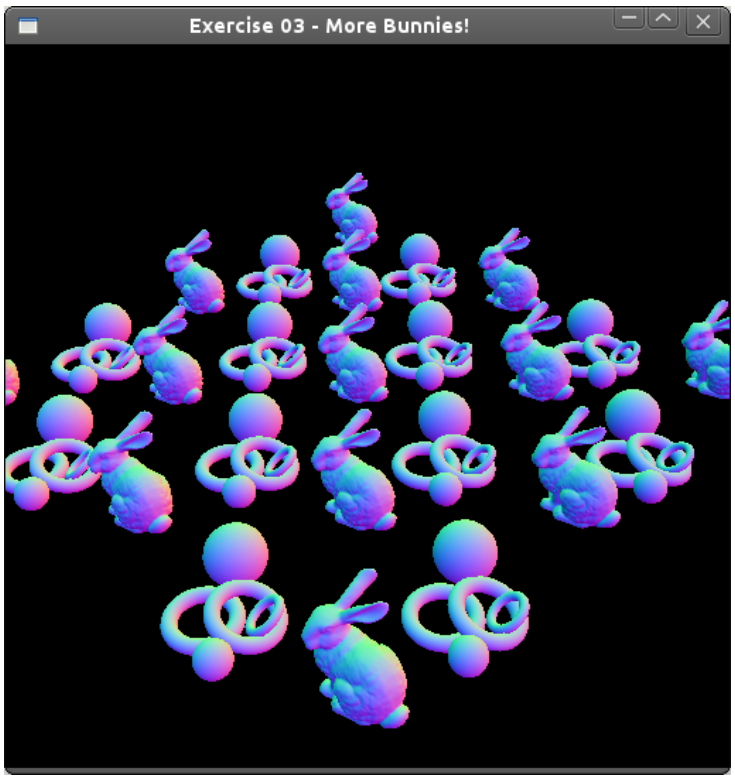


Figure 1: A rotating grid of geometry.