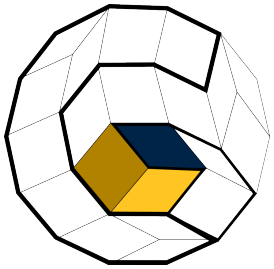
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Matthias Überheide

November 24, 2014



Real-time Computer Graphics WS1415
Assignment 3

Present your solution to this exercise on Monday, November 24th, 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 make use of simple affine transformations as discussed in the last lecture. To be able to render more objects than just geometry defined as a header file and to be more flexible, you will also create a structure to load geometry from separate 3d model files and render it conveniently.

3.1 Load a 3D mesh from an OBJ file (70 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, texture coordinate, normal and face data for a 3D mesh object and to import data into a vertex list, a texture coordinate list, a normal list and a list of vertex indices used to form faces (defined as triangles). To render an 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:

- Open the attached model files `meshes/bunny.obj` and `meshes/armadillo.obj` to get used to the OBJ file format.
- In the `loadObjFile` method in `ObjLoader.cpp` read the model file given by the parameter `fileName` line by line.
- Scan each line for the key defining the contents of the line. The keys `"v"`, `"vn"`, `"vt"` and `"f"` indicate a vertex (3D), a vertex normal (3D), a texture coordinate (2D), resp. a face definition. Although normals and texture coordinates may be optional, the object loader should be able to read this information. You should use an instance of `std::stringstream` for the reading task. The shift operator `>>` as well as the functions `peek()` and `get()` may help you. Add all the data to the predefined respective local lists (`localVertexPosition`, `localVertexNormal`, `localVertexTexcoord`, `localFace`).
- The face definition includes vertices, vertex texture coordinates, and vertex normals (`f v/t/n`), whereby texture coordinates and normals may be optional (`f v//n` or `f v//`). The parser should be robust against all possible combinations. Also allow to import quads as faces instead of triangles. For this task directly split a quad up into two triangles. Be sure to use correct indices, since a `std::vector` starts indexing at 0, the Wavefront OBJ format however starts at 1.
- Create an indexed vertex array for every triplet of `vertexId, normalId and texCoordId`. For this task use a local `MeshData` struct as a temporary container. Every triplet is unique and indexed by `meshData.indices`. Follow the TODO descriptions in the source code and complete the missing parts in the code. Again, be sure to use correct indices.

Congrats! 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/bunny.obj` and `meshes/armadillo.obj` contain 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. Hint : Texture coordinates are not used yet for rendering. This task will be a part of a later exercise.

3.2 Organize geometry in a scene graph structure (30 Points)

Using vertex array objects like the `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 2×2 grid of pairs of geometry instances (Stanford Bunny and Armadillo from the given files `meshes/bunny.obj` and `meshes/armadillo.obj`). When rendering the grid the armadillos should be located around the origin and the bunnies at opposed positions (see the screen-shot in figure 1).
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.
Try to create the following:

- Rotate the scene clockwise around the y-Axis about `rotAngle` degrees to imitate a rotating camera.
- Render a 2×2 grid of pairs of bunny and armadillo geometry instances. Use a translation of $0.15LE$ in x and z direction for the armadillos and $0.35LE$ for the bunnies in world space coordinates. Shift each object about $0.25LE$ in y direction.
- Scale all objects uniformly down by a factor of 10.
- Use different rotations about the y-Axis so that the armadillo and bunny of each pair look at each other.
- Right before rendering an object, upload the current state of the `modelView` matrix stack.

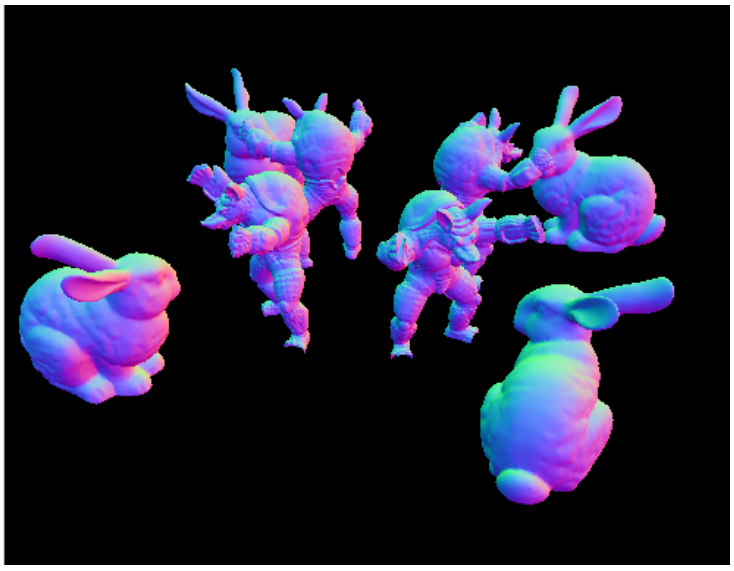


Figure 1: Screen-shot of the rotating scene.