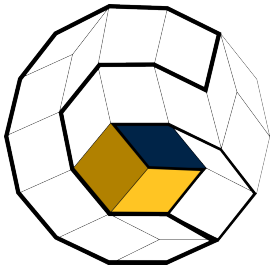
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Dipl.-Ing. Michael Stengel

November 10, 2014



Real-time Computer Graphics WS14/15
Assignment 2

Present your solution to this exercise on Monday, November 17th.

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.cs.tu-bs.de instead.

In this assignment you will be introduced to the basics of OpenGL programming. You will create a simple application drawing a static Stanford Bunny on screen. This bunny is represented as vertex array and gets passed to a simple shader pipeline. Please have a look at the official OpenGL documentation (`http://www.opengl.org/sdk/docs/man3/`), if you encounter any problems or if you want to look up the proper signature of a required method. Of course, you can also contact me.

2.1 Load, compile and bind your first shader program (10 Points)

To load a shader implement the missing parts of

- `char* loadShaderSource(const char* fileName)`
- `GLuint loadShaderFile(const char* fileName, GLenum shaderType)`.

defined in `Ex01.cpp`. The first method is used to simply load a given textfile and return the shader code as a `char` array. This code can then be used within the latter method to set up and compile a GLSL shader.
Finally, in `initShader()` you will need to load and compile two shader types:

- A `GL_VERTEX_SHADER` processing the geometry data passed from your code to the GPU.
- A `GL_FRAGMENT_SHADER` responsible for assigning a specific color to every pixel covered by visible geometry.

Use `glCreateShader()` to create a specific (vertex, geometry or fragement) shader. Attach the shader's source code (given as `char` array) to the newly created shader using the method `glShaderSource()`. After you've uploaded the shader source, compile the shader using `glCompileShader`.
Having the code for the given shaders compiled, you need to attach them to a custom shader program. Use `glCreateProgram()` to create such a program and attach the shaders for each shader type (`GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`) to the program using `glAttachShader()`. When all needed shaders are attached, the program needs to be linked, making it ready to be used. Execute `glLinkProgram()` and you're good to go.
Of course the shader needs to be activated before rendering, but this is already done in the given code and you don't need to bother.
The two shaders are already given as (very basic) source code. The geometry shader simply transformes any given vertex by the user defined *modelview matrix*, describing the geometry transformations, and the *projection matrix*, describing the camera's viewing frustum.

Load the two shaders `simple.vert` and `simple.frag` (located in `shaders/`), attach them to a shader program `shaderProgram` and finally link the shader program.

2.2 Draw geometry using Vertex Array Objects (30 Points)

In the exercise stube there is a file named *bunny.h*. This file contains the geometry of the Stanford Bunny as indexed vertices and vertex normals. There are three arrays of floating point data. The 3D vertex positions are stored in `GLfloat bunny[] =` $[x_0, y_0, z_0, \ldots, x_n, y_n, z_n]$. The shape of the bunny is then encoded as triangles in `GLint triangles[]` indexing the vertices of `bunny[]`. The triangles are defined in a counter clockwise orientation and are also stored on triangle after another: $[t_{0_A}, t_{0_B}, t_{0_C}, \ldots, t_{m_A}, t_{m_B}, t_{m_C}]$, where $t_i = (t_{i_A}, t_{i_B}, t_{i_C})$ represents a triangle and $t_{i_P}, P \in A, B, C$ indexes a vertex tuple $v = (x_j, y_j, z_j)$ out of `bunny[]`. Furthermore there is an array `GLfloat normals[]` containing normal information for each defined vertex. The structure is the same as for `bunny[]` and is also indexed in the same way.
Implement the missing parts within the method `void initScene()` of `Ex02.cpp`. In order to render this bunny later on, you need to setup a buffer structure that can be rendered by OpenGL. We will use a *Vertex Array Object* (or short VAO) for this, as presented in the lecture.
In order to set up the VAO properly, we need to organize the data it will hold. Since we have vertex indices available by `triangles[]`, we can render the data as a *Vertex Buffer Object* (VBO) indexed by a *Index Array Object*. The indexed data itself is contained in a single data array, which we need to define. This data array consists in our case of the 3D vertex coordinates and the normal data for each vertex. For simplicity we just concatenate these two data arrays instead of interleaving them. Then we have an array like `GLfloat a[] =` $[x_0, y_0, z_0, \ldots, x_n, y_n, z_n, N_{0_x}, N_{0_y}, N_{0_z}, \ldots, N_{n_x}, N_{n_y}, N_{n_z}]$.
Using this array, you need to create a VBO and upload the data of this array to it. Because the geometry data gets passed to the shader later on, we need to define *where* we can access *which* data. This is done by setting up specific layout locations within the shader code. Have a look at `shader/simple.vert` where the layout is defined for vertex and normal data. We need to upload the vertex data to channel 0 and the normal data to channel 1. Do as follows:

- First, generate a VAO organizing all data and its layout. Use `glGenVertexArrays()` to generate a new Vertex Array Object and store its OpenGL address in `bunnyVAO`, which is already defined.
- Bind your generated VAO to activate it for further configuration (setup of used buffers).
- To hold the actual data, we need a VBO. Generate a new VBO using `glGenBuffers()` and store its handle in `bunnyVBO` (also already defined).
- Bind the new generated VBO as a `GL_ARRAY_BUFFER`.
- Upload the data of the bunny to the VBO. Use `glBufferData()`, where you need to define what type of buffer your are uploading your data to (`GL_ARRAY_BUFFER` in this case), how large your data set is in bytes, where the data is stored (address of the first value in your concatenated array), and how the data should be handled later on. Since the bunny will not change its shape during runtime, we can set the VBO to `GL_STATIC_DRAW`.
- No we need to define, how to handle and interpret the just uploaded data. Set the data pointer for the first vertex attribute (which corresponds to the layout defined in the shader code) for your vertex position data. Use `glVertexAttribPointer()` to set the layout position, the number of values to be used for *one* element of the value defined at this layout position (in our case it's three floats per vertex), whether the data has to be normalized automatically or not (in this case: not), the padding between rendered elements (which is 0 in this case) and set the beginning of the data to the very first value of the bound VBO.
- Enable the defined vertex attribute using `glEnableVertexAttribArray()`.
- To pass the normal data to our shader, too, set the data pointer for the normals the same way as for your vertex positions, except that we need to use another layout address (being 1 as defined in the vertex shader) and the beginning of the data is the first value of the *normal data* in our concatenated array. Pass the correct offset (in bytes) as the last argument of the method.
- Enable the defined vertex attribute for normal data.

After having created a VBO using the bunny's data, we need to setup the correct indexing for this data to truely form a bunny and not an unstructured ball of triangles. This is done by creating an *Index Buffer Object* (or short IBO).

- Again, create a buffer object using `glGenBuffers()`, but this time store its handle in `bunnyIBO` (also already defined).
- Bind the buffer as `GL_ELEMENT_ARRAY_BUFFER`.
- Upload the index data to the IBO using `glBufferData()` (buffer type is `GL_ELEMENT_ARRAY_BUFFER`) and the index data from `triangles[]`.

Now that the geometry is set up properly, it needs to be rendered somehow. Implement the missing parts in `void renderScene()` to achieve this. Simply do this:

- Bind your VAO.
- Execute `glDrawElements()` to draw the data used by the currently active VAO. You need to set the type of geometry to `GL_TRIANGLES`, the number of elements to render from your buffers (number of defined triangles), the type of the index data, which is `GL_UNSIGNED_INT` in this case, and the position of the first index to process, which is the position of the very first value in our IBO (here: `(void*)0`).

After correct implementation you should see the Stanford Bunny on your screen (see figure 1) colored according to the defined normals.

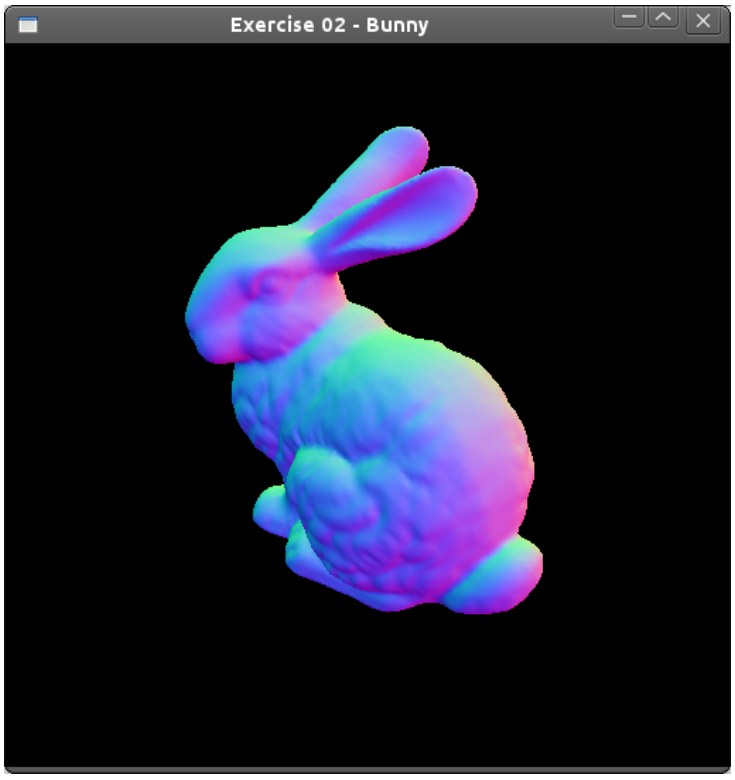


Figure 1: This is how your screen should look like after task 2.2