Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

computer graphics laboratory

Prof. Markus Gross

Visual Computing
Exercise 6: Introduction to OpenGL

Hand-out: 7. November 2017 – Hand-in: 14. November 2017

Goals

- Getting acquainted with the OpenGL GLFW library
- Learning about the GL graphics pipeline, shaders and buffers
- Understanding how to apply color spaces for visualization of scientific data

General Remarks

All practical exercises will provide a framework and CMake configuration files that can be used to create Makefiles or IDE solution. The files can be downloaded from the course web page. Please refer to the included readme files for further instructions.
Please submit your modified `color.cpp` and shader files by email to leonhard.helminger@disneyresearch.com. Submissions are due by 14. November 2017.

Resources

The lecture slides and exercise slides are accessible via the course web page http://graphics.ethz.ch (Teaching-> Visual Computing)

For further information on OpenGL, refer to the OpenGL Specification and the OpenGL Shading Language Specification. Both are available from http://www.opengl.org/registry

Exercise 1) Mesh setup and Initialization

In this exercise, we will setup a mesh (St. Helens volcano) for rendering. All code needed for this project is contained in the file `color.cpp`.
First, get acquainted with the program structure, especially the `loadMeshObject` routine. Then, have a look the shader code. Note which attributes and uniforms are used in the fragment and vertex shader.

Find the part of `loadMeshObject` denoted by Exercise 1). Insert code to create a vertex buffer and an index buffer and download the data in `pVertexArray` and `pIndices` to these buffers. Next, identify the per-vertex attribute inputs of the vertex shader and bind them to the correct array pointer. (The procedure is covered in the exercise slides)
When implemented correctly, the mesh will be displayed on the screen (still all-grey as no lighting is performed yet).

Exercise 2) Normals for Lighting

In this exercise, we will add simple lighting in order to see the details of the 3D mesh. In the vertex shader, a simple diffuse lighting method is implemented. This code however requires per-vertex normals to calculate the lighting.

a) Face normals

First, calculate the face normal for each triangle out of the vertex positions. Find the part of `loadMeshObject` denoted by Exercise 2a). Insert code to fill the normal component of the `pTriangleArray`. The comments in the code provide some guidance to the individual steps.

b) Vertex normals

Vertex normals are generated by averaging the face normals around each vertex. Implement the function `calcVertexNormals` to do this.

Exercise 3) Coloring the mesh

While lighting helps to see details of a 3D mesh, in scientific visualization color mapping is often used to convey information is an easy manner. We will implement an elevation map (i.e. mapping height to a color) in this exercise. As we don't want to disturb the lighting, only color hue should convey the information, intensity should remain unchanged.

The function `calcVertexColor` calculates the per-vertex color input the vertex shader. Implement this function to map height to color hue. Hint: Use the provided `hsv2rgb` helper function.

Exercise 4) Adding color effects

In this exercise, a specific region of the mesh surface should be dynamically highlighted (i.e. without changing the mesh). We want to do this by transitioning the color to white in the vicinity of the point to highlight.

First, add a uniform to provide the global variable `g_interestingPoint` to the vertex shader. To do this, set the uniform value each frame in `displayFunc` and add the respective definition to the vertex shader.

Secondly, add code to the vertex shader that works according to the pseudocode:

- Get squared distance of current vertex to interesting point
- Divide this value by 400 and clamp to 0…1 range
- Use the result to fade between `color_in` and white

Look up the functions needed for fading, clamping and distance calculation in the OpenGL Shading Language specification (there's a chapter called built-in functions).