ETH zürich

computer graphics laboratory

Prof. Markus Gross

Visual Computing
Exercise 9: OpenGL Shading Language and Blending
Hand-out Date: 29 November 2022

Goals

- Learning the OpenGL Shading Language (GLSL) in more detail
- Learning the difference between vertex and fragment shaders
- Understanding depth test and the blending stage

General Remarks

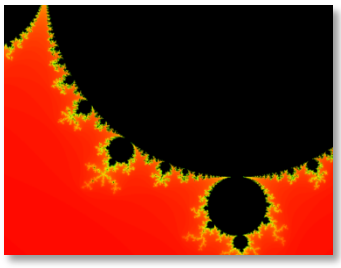
Please read the README.md file in the main folder.

Resources

The lecture slides and exercise slides are accessible via the course web page https://cvg.ethz.ch/teaching/visualcomputing.

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) Mandelbrot fragment shader



In this exercise, we will generate a full screen Mandelbrot fractal. The given framework draws a screen filling rectangle which is made up of several triangles. We will use the OpenGL fragment shader to texture the rectangle procedurally with the Mandelbrot fractal.
Your task is to finish the fragment shader in `ex2.glslf` by implementing the helper function `complexProduct` and the pseudo code given below.

a) Helper Functions

Implement the function `complexProduct` in `ex2.glslf`, which takes two `vec2` values representing complex numbers and returns their complex product. The x-component is defined to be the real part and the y-component to be the imaginary part of the complex number.

b) Mandelbrot

In this exercise the fragment shader is invoked once for every pixel on the screen. The varying `screenPos_out` is precomputed by the vertex shader and can be used to get the current pixel location.
For every pixel a complex number c is initialized with the pixel's location, a given scale and center. The fractal's value for this pixel can then be computed using the pseudo code given

below. The pre-implemented function `floatToColor` can be used to retrieve the final RGB color.

$$s \leftarrow zoom \cdot \begin{pmatrix} 4/3 \\ 1 \end{pmatrix}, \qquad c \leftarrow s \cdot screenPos_{out} - s/2 - center$$
$$z \leftarrow c, \qquad i \leftarrow 0$$
do
$$z \leftarrow complexProduct(z, z) + c$$
$$i \leftarrow i + 1$$
until $|z|^2 > 4.0$ ***or*** $i \geq maxIter$
$$color \leftarrow floatToColor\left(\frac{i}{maxIter}\right)$$

Exercise 2) Vertex Shader

In this exercise we will utilize the vertex shader to deform and translate the fractal rectangle. The rectangle is made up of 64*64*2 triangles (see `MainProgram::Initialize`). This gives us the required degree of freedom to reshape the rectangle by modifying its vertex positions with the vertex shader (`ex2.glslv`).
Our goal is to get a more interesting and moving shape instead of the rectangle. You are free to implement any interesting or good-looking deformation. To get the intended result of the exercise, please follow the steps below.

a) Translating the rectangle

First, we want to rescale (by 30%) and translate the whole rectangle. This can be done by multiplying every vertex with 0.3 and adding the offset `posObject`.

$$\text{posVtx} \leftarrow 0.3 \cdot \text{pos} + \text{posObject}$$

You need to add the required new uniform `posObject` to `ex2.glslv`.
Furthermore, in `MainProgram::renderObject`, add the code to set the value of `posObject` to (posX, posY, posZ). Now you can calculate a new position for the rectangle every time the window is redrawn. In a later exercise we will use alpha blending, therefore you should also change the z coordinate of the object such that it moves forward and backward. The z component defines the distance to the camera. The x and y components represent the position inside the window. All coordinates need to be in the range [-1, 1]. We have chosen a circular animation in the xz-plane:

$$\text{posX} \leftarrow -\sin(\text{time} \cdot 0.7) \cdot 0.8$$
$$\text{posY} \leftarrow 0$$
$$\text{posZ} \leftarrow -\cos(\text{time} \cdot 0.7) \cdot 0.8 + 1.5$$

b) Deforming the rectangle

We want to further deform the rectangle in a non-linear way. To deform the rectangle every vertex has to be displaced depending on its initial position. This can be done most efficiently in the vertex shader. To get a time-dependent deformation, you can add a new uniform `time` to the vertex shader. Set this uniform to the current time before a new frame is drawn. We have chosen a sinusoidal deformation using the following equation to displace the vertices `posVtx`:

$$\begin{aligned}\text{posVtx}.x &\leftarrow \text{posVtx}.x + \cos(\text{posVtx}.y \cdot 8.0 + \text{time} \cdot 10.0)^2 \cdot 0.1\\ \text{posVtx}.y &\leftarrow \text{posVtx}.y + \sin(\text{posVtx}.x \cdot 8.0 + \text{time} \cdot 10.0)^2 \cdot 0.1\end{aligned}$$

Exercise 3) Blending

a) Set Blending State

In `MainProgram::Display`, draw a second object at the following position:

$$\begin{aligned}\text{posX} &\leftarrow 0\\ \text{posY} &\leftarrow \sin(\text{time} \cdot 0.7) \cdot 0.8\\ \text{posZ} &\leftarrow \cos(\text{time} \cdot 0.7) \cdot 0.8 + 1.5\end{aligned}$$

You should now see two moving rectangles. In `MainProgram::Display`, set up the blending stage so the final color is computed as follows:

$$\text{final}.rgb \leftarrow \text{src}.rgb \cdot \text{src}.a + c\text{dest}.rgb \cdot (1 - \text{src}.a)$$

To enable blending and set the blending parameters, you will need the following OpenGL functions: `glEnable`, `glBlendEquation`, `glBlendFunc`.

b) Compute per-pixel alpha

In `ex2.glslf`, compute the value of `gl_FragColor.a`. Note that due to how we have set up the blending stage, an alpha value of 0.0 results in a fully transparent pixel, and 1.0 is opaque. Compute the alpha value such that the inner part of the rectangle is transparent. As a hint, the distance to the center of the rectangle can be computed as follows:

$$\text{r} = \text{distance}(\text{screenPos\_out}.xy, \text{vec2}(0.5, 0.5))$$

c) Bug?

You may have noticed that when the first object moves in front of the second one, transparency on the first object does not work as expected. Do you have an explanation for this ‘bug’? How could it be fixed?