In this computer graphics assignment we are going to implement a volume renderer using OpenGL and the OpenGL Shading Language (GLSL), which compiles and runs on the graphics card (GPU). The data set used in the assignment is a 3D computer tomography (CT) image of a foot, see Fig. 1.

The idea behind the volumetric rendering we are going to implement is illustrated in Fig. 2. A virtual camera is pointed towards a 3D volume containing the 3D data set of the foot. We wish to color each pixel on the sides of the volume visible to the camera according to the following principle: For each surface pixel on the volume visible from the camera, imagine a ray starting at the camera position, entering the volume through the pixel and then passing through the volume. Examples of rays are shown in Fig. 2, denoted “a”, and “b”. As they pass through the volume, they encounter different bone densities along their way. The maximum density value found along each ray is used to color the pixel at the entry point of the ray (a.k.a. maximum intensity projection, MIP).

In order to perform the search along the rays very fast, this part of the code should be implemented in GLSL and executed on the GPU, allowing the processing of many pixels in parallel. These GLSL programs are called shaders. If each surface pixel on the volume facing the camera is colored according to the volume rendering method described above, your program will produce an image similar to Figure 1. The image shows the provided data set rendered with each pixel set to the maximum intensity found along the ray passing through it. Try to reproduce it.

Shaders

Shaders are code written for execution on the graphics processing unit (GPU) to affect the behavior of the rendering pipeline, allowing us to create a wide range of rendering effects. Use the `loadProgram()` function provided in the assignment code to load your shaders and `glUseProgram()` to make the GPU use the shader code previously loaded with `loadProgram()`. We will look into two different types of shaders, Vertex shaders and Fragment shaders. Vertex shaders are run on each vertex in the scene and can manipulate properties such
Figure 1: Data set: Computer Tomography (CT) of a foot. The 3D Volume seen by the camera, with each volume surface pixel colored according to the maximum intensity value found along the corresponding ray.

Figure 2: Basic idea of volumetric rendering, rays pass through the 3D volume containing the 3D image from the data set. Each surface pixel on the cube is colored according to the data found along the corresponding ray.
as vertex position, color, and texture coordinates. A vertex shader has to at least write to the variable \texttt{gl\_Position}, usually transforming the vertex with the modelview and projection matrices. To set the color of the vertex, we can use \texttt{gl\_FrontColor} and/or \texttt{gl\_BackColor} to set the color(s) to be rendered on the front and backside of the polygon. We can access the current input vertex position via \texttt{gl\_Vertex}. The vertex shaders should be implemented in the \texttt{texCoordVertex.glsl} and \texttt{tracerVertex.glsl} files which can be found in the Resource Files directory in the Solution Explorer in Visual Studio after loading the solution \textit{shaders.sln}.

Fragment shaders run on each pixel inside the polygons after the rasterization step. Fragment shaders can manipulate color of individual pixels, apply textures, compute per-pixel normals and even more advanced tasks. In this assignment our fragment shaders will write to \texttt{gl\_FragColor} to manipulate the color of the pixels. The input to a fragment shader is interpolated values computed in the previous stage of the pipeline - such as vertex positions, colors, normals, etc.. The fragment shaders should be implemented in the \texttt{texCoordFragment.glsl} and \texttt{tracerFragment.glsl} files which can be found in the Resource Files directory in the Solution Explorer in Visual Studio after loading the provided solution file \textit{shaders.sln}. How do we know the 3D position of the entry and exit points of each ray? The start and end points of the ray through the volume can be found by drawing the bounding box of the data set (use the \texttt{drawCube()} function in combination with a GLSL shader), “colored” by the texture coordinates. Figure 5 show the outside (front-facing polygons) and Figure 6 show the inside (back-facing polygons) of the bounding box, with the r,g,b channels representing volume texture coordinates. To render the back and front sides of the polygons you can use culling (removing back or front side of polygons) which is controlled by \texttt{glCullFace} (look it up) and \texttt{glEnable(GL\_CULL\_FACE)}.

These colorful (and useful!) textures can be created by rendering the bounding box with a texCoord shader implemented in GLSL that copies the desired texture coordinate to the fragment color. Then, in the final render, these textures are bound for use and are read by the tracer shader to obtain the start and end point for the part of the ray intersecting the volume. With the ray known, the volume can be sampled along it by iteratively following the ray until the end is reached or it can be determined an early exit will not affect the end result. The steps should be small enough so that no significant features in the volume are missed.

### Rendering to texture

By default, OpenGL renders to the framebuffer making graphics visible in the graphics viewport. We can choose to render to other buffers, such as a texture. Here we want to render the colored bounding boxes in Fig. 4 and 3 to textures and make these textures available for the tracer shader code that calculates the intersecting rays. To render the framebuffer to a texture, use the Frame Buffer Object (FBO) class. First bind the FBO by calling \texttt{bind()} pro-
Figure 3: Exterior of volume colored with RGB values corresponding to each surface points’ coordinates in XYZ. The white corner corresponds to coordinate (1.0, 1.0, 1.0).

Figure 4: Interior of volume colored with RGB values corresponding to each surface points’ coordinates in XYZ. The black corner thus corresponds to coordinate (0.0, 0.0, 0.0).
vided in the FBO class, render the graphics and then `unbind()` the FBO to get back to accessing the regular framebuffer again. Everything you draw between `bind()` and `unbind()` will be rasterized and stored in the texture in the FBO object, rather than shown on the screen. There is also a help method for debug rendering called `render()`, the input is in window coordinates. To get the actual texture object from the FBO to be used by your shader code, call `getTexture()`. The FBO works as a regular framebuffer so don’t forget to call `clear()` if you need to clear the contents of the buffer. In order for your shader to access multiple textures, you can bind textures to different texture units; these units can then be accessed from the shader. This can be done by calling `glActiveTexture(GL_TEXTUREn)` where `n` is an integer 0, 1, 2, ... before calling `glBindTexture()` to bind the texture to unit `n`.

You must then set the value of the sampler uniform variable (used in the GLSL code) that will be used to read from the texture to the number of the unit to which it is bound i.e. `n`.

```c++
glUniform1i(samplerParameterLocation, n);
```

The process is repeated for each texture, assigning to a new unit every time.

To find the location `samplerParameterLocation` of the GLSL parameter you can use: `glGetUniformLocation()` (Please see OpenGL documentation on how it is used) Hint: These functions should be called from the draw function in `scene.cpp`.

**Some useful GLSL variables and functions**

- For Vertex shaders
  
  - `gl_Vertex` gives you the position of the current vertex in the object.
  - `gl_FrontColor` and `gl_FrontColor` can be set to affect the current vertex color. Interpolated color values between the vertices will then be accessible from the fragment shader. The front and back side of the polygon can have different colors, hence the two functions.

- For Fragment shaders
  
  - `gl_FragCoord.xy` provides the coordinates (in screen coordinates) of the fragment currently handled by the fragment shader. `texture2D()` and `texture3D()` are used to read values from a position in a texture. `gl_Color` provides a color value of the current fragment, calculated as an interpolation between the colors of the vertices of the polygon containing the current fragment. `gl_FragColor` set this to manipulate the current fragment color.
Remarks

The instructions are not self contained and you might have to revise the lecture notes. If you want further insight, we can recommend the following resources:

- http://www.lighthouse3d.com/opengl/gls1/
- http://nehe.gamedev.net/data/articles/article.asp?article=21

Please note that this assignment typically requires some more work than assignments 1-3, we have scheduled one extra session to provide extra time for questions.

GOOD LUCK!
Pontus & Gustaf