Volume rendering in GLSL

Computer Graphics 1TD388
Uppsala University, Spring 2015
May 6, 2015

Figure 1: Computed tomography (CT) volume image of a foot, rendered with GPU-accelerated ray-casting. The 3D volume image is seen from the camera’s perspective, with each viewport pixel colored according to the maximum intensity value (i.e., the maximum tissue density value) found along the corresponding ray cast from the pixel through the volume.

1 Introduction

The goal of this project is to implement a volume rendering technique called GPU-accelerated ray-casting. This rendering technique can be used to visualize the content of volumetric (3D) images such as the computed tomography (CT) foot image shown in Fig. 1. Ray-casting (and volume rendering
in general) generates a 2D projection of 3D volume data so that the data can be displayed on the screen. As you saw in lecture 10, we can alter the appearance of this 2D projection by using different volume rendering modes. This allows us to render semi-transparent objects as well as opaque shaded surfaces, and to create X-ray-like images like the one shown in Fig. 1.

The basic idea of ray-casting is to, from each pixel in the graphics viewport, cast a ray through the volume image and sample (look up) the voxel intensity values found along the ray at even intervals. One can then extract, e.g., the maximum intensity value found along each ray to create a 2D projection of the 3D data (a so-called maximum intensity projection, MIP).

To obtain a high enough framerate (20 frames per second or more), the ray-casting should be implemented in a GLSL fragment shader and be executed on the GPU, allowing the processing of multiple rays and pixels in parallel (ray-casting is a so-called embarrassingly parallel problem, making it ideal to be executed on GPUs). The basic steps for implementing GPU-accelerated ray-casting are as follows:

1. Render the frontface of the volume image’s bounding box to a 2D RGBA texture to obtain ray starting points (Fig. 2).

2. Render the backface of the bounding box to a 2D RGBA texture to obtain ray end points (Fig. 3).

3. Render a fullscreen quad (i.e., two triangles that together fill up the entire screen) and (in the ray-casting fragment shader) subtract the backface texture from the frontface texture to obtain ray direction vectors.

4. Given the ray starting points and direction vectors, cast (still in the fragment shader) a ray from each fragment into the volume image texture (Fig. 4).

5. Let the ray gather intensity values from the voxels it pass through until it reaches the end point. To create a maximum intensity projection, select the maximum intensity value found along the ray as fragment color (see Fig. 1).

The remaining sections in this document will provide some further details about the shaders you need to implement and how to use framebuffer objects (FBOs) to render the front- and back faces of the volume bounding box to 2D textures.
Figure 2: Front faces of the volume bounding box colored with RGB values representing 3D texture coordinates. The white corner corresponds to texture coordinate (1.0, 1.0, 1.0). This image is stored in a 2D RGBA texture.

Figure 3: Back faces of the volume bounding box colored with RGB values representing 3D texture coordinates. The black corner corresponds to texture coordinate (0.0, 0.0, 0.0). This image is stored in a 2D RGBA texture.
Figure 4: Basic idea of ray-casting. A ray is cast from each pixel/fragment in the viewport through the 3D volume image. The rays enter the volume at the front faces of the bounding box and sample (look up) the voxel intensity values at even intervals until they hit the back faces of the bounding box. Each fragment in the viewport can then be colored according to the voxel intensity values found along the corresponding ray.

2 Rendering to texture

By default, OpenGL renders to the Default framebuffer, which makes the graphics visible in the graphics viewport on the screen. By using framebuffer objects (FBOs), we can choose to render to other buffers such as 2D RGB or RGBA textures. This is useful for volume rendering, where we want to render the colored bounding boxes in Fig. 2 and 3 to 2D RGBA textures and make these textures available to the ray-casting shader. The source code framework provides two FBO utility objects (frontFaceFBO and backFaceFBO) that you can use for this purpose. First bind the FBO object that you want to use by calling its bind() method, then render the graphics, and finally call the unbind() method to unbind the FBO and go back to using the Default framebuffer again. Everything you draw between the calls to bind() and unbind() will be rendered to the 2D RGBA texture attached to the FBO object, rather than shown on the screen. The FBO class works as a regular framebuffer, so don’t forget to call clear() in each frame to clear the depth and color of the buffer.

To fetch the actual OpenGL texture object from the FBO, call its getTexture() method with GL_COLOR_ATTACHMENT0 as argument.
3 Shaders

When you first run the volume rendering program (without modifying any of the code) it will load a volume image from file into a 3D texture and display a white cube representing the bounding box of the volume image. We have already set up the basic components of the program (volume loading, camera, trackball navigation, FBOs, etc), so your main task is to complete the following shaders (located in src/shaders):

- boundingGeometry.vert
- boundingGeometry.frag
- rayCaster.vert
- rayCaster.frag

boundingGeometry.vert and boundingGeometry.frag should be used to render the color-coded front- and backfaces of the volume bounding box, and rayCaster.vert and rayCaster.frag should perform the ray-casting. You need to write some code in main.cpp as well to pass uniform variables, bind textures, set up the AntTweakBar GUI, etc, but the main work will be to implement the shaders.

The start and end points of the rays can be generated by drawing the front- and backfaces, respectively, of the volume bounding box and color-code these faces with 3D texture coordinates (use the drawBoundingGeometry() function in combination with the boundingGeometry.vert and boundingGeometry.frag shaders). Basically, you just draw an RGB color cube as you did in assignment 2. Figure 2 shows the outside (front-facing polygons) and Fig. 3 shows the inside (back-facing polygons) of the bounding box, with the RGB channels representing 3D texture coordinates. To render either the front- or the backfaces of the bounding box, you need to (in main.cpp) enable face culling, which is controlled by glCullFace (look it up) and glEnable(GL_CULL_FACE). The vertex shader (boundingGeometry.vert) should generate a 3D texture coordinate for each vertex and pass this texture coordinate as a vec3 variable to the fragment shader. The fragment shader (boundingGeometry.vert) should just assign the incoming 3D texture coordinate to FragColor to color-code the fragment with the texture coordinate.

Then, in the ray-casting pass, the front- and backface textures are bound for use and are accessed by the rayTracer.frag shader to obtain the ray starting point, the ray end point, and the ray direction for each fragment.
rayTracer.vert should just define a 2D texture coordinate for each vertex of the fullscreen quad and pass this texture coordinate as a vec2 variable to the fragment shader. With the ray parameters defined, we can (in rayTracer.frag) iteratively sample the volume texture at even intervals along the ray until the end point is reached or it can be determined that an early termination of the ray will not affect the end result. The ray steps should be small enough so that no significant features in the volume are missed.

In order for your ray-casting shader to access the 2D front- and backface textures and the 3D volume texture, you need to bind these textures to different texture units. 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 also need to pass each texture unit number as a uniform int variable (use the setUniformi() method of the program object) to the rayCaster shader. The texture units can then be accessed as uniform sampler variables (sampler2D or sampler3D, depending on texture type) in the rayCaster.frag shader.

4 Blit

You can render ("blit") the frontFaceFB0 color texture to the screen by calling

blit(globals.blitProgram, globals.quadVBO,
    globals.frontFaceFB0.getTexture(GL_COLOR_ATTACHMENT0)->getHandle());

before the glutSwapBuffers() call in display(). This can be useful for debugging or if you, for instance, want to add an AntTweakBar option for displaying the color-coded frontfaces instead of the volume rendering result. The backFaceFB0 color texture can be rendered in a similar way.

5 Remarks

Please note that these instructions are not self-contained and that this project typically requires some more work than assignments 1-3. You should revise the lecture notes and browse through some of the references listed on the project web page before starting implementing the ray-caster.