Volume rendering

Gerard Martin Teixidor

May 5, 2021

Functionalities implemented

All the basic functionalities have been implemented. Also, the following advanced functionalities are present:

- Advanced illumination (shadows)
- Ray casting rasterization acceleration

Ray casting

The ray casting implemented in this project uses rasterization acceleration to compute the starting and end point of each ray. This has been implemented by rendering the bounding volume mesh two times. First, the back faces of the bounding mesh are rendered and each framgnet stores the world position in a texture, which they represent the ray end position. Then, a second render is perform where the front faces are rendered getting the ray start position and computing the volume ray casting. To improve the performance, early alpha termination have been implemented. Finally, to reduce some sampling rate artifacts, a random offset is added.

The MIP mapping of the 3D volume texture have been disabled since, due to the nature of how the 3D texture is being sampled, it is impossible for the compiler to synchronize adjacent fragment samples, making it to select the wrong MIP mapping.

The *Volume* class coordinates the two rendering passes. The first rendering pass uses the *position* shaders, while the *volume* shaders are used in the second rendering pass.

As a curiosity, the new *OpenGL* QT class (*QOpenGLWidget*) does not grantee the default framebuffer to be *0*. The function *defaultFramebufferObject()* provides it, but it can vary over time.

Blinn-Phong shading

As a basic shading model Blinn-Phong shading model had been chosen.

It is important to notice that the normal, apart of being computed as the volume density gradient, it is also negated. This is because the input volume data density is inverted: *0.0* represents empty space while *1.0* represents a completely opaque volume.

The shading code can be found int the *lighting(...)* function of the *volume* fragment shader.

Shadows

Because how the volume is rendered, the *Half-Angle Slicing* technique for rendering the illumination did not make sense. Instead, this project improves the lighting by computing shadows in a recursive ray casting.

At each point where the illumination is computed, a second ray is thrown in the direction of the light to check if the light is visible. As in the first ray casting, early alpha termination have been implemented. Because this secondary ray only modifies a small aspect of the lighting, the sampling rate of this ray can be much lower without noticing. Finally, as in the primary ray casting, to reduce some low sampling rate artifacts, a random offset is added.

In contrast to the primary ray, is not possible for the shadow ray to use rasterization acceleration to get end point and instead uses box-ray intersection to get it. A camera from the light position could be used (like in a shadow map), but this technique has its own drawbacks.

The main shadows code can be found in the *shadow(...)* function of the *volume* fragment shader.

Interactive transfer function

This project has an interactive histogram which allows to modify each channel curve.

The user can add, remove and modify points to each channel to create a custom curve. It is important to notice that the interpolation between points is linear. A more advanced interpolation can be easily implemented but for demonstration purposes the linear interpolation is enough.

Each time a curve points is modified, a *255* texels *RGBA* texture is generated as a lookup table. Then, this texture is send to the volume rendering shader to correct the sampled volume density.

The main functionalities can be found in the *HistogramWidget*, *HistogramChannelWidget* classes. Also the *GLWidget* class contains the *histogramUpdated(...)* callback function which then calls *histogramUpdated(...)* in the *Volume* class.

As a side note, *QT* does not guarantee that all the *initializeGL()* functions are called before all *paintGL()* functions. This is the reason why the communication between the *GLWidget* and *HistogramWidget* classes so complex.

Tiled rendering

Some OS or graphics drivers kills the task when it takes to long. This is a problem since volume rendering is quite expensive an can take up to seconds to render a single frame. To avoid this the software allows to render the frame in tiles. Even though there is an overhead of rendering the same geometry and volume for every tile, because each rendering tile takes much less time, it allows to finish the draw call without being killed.

The implemented code can be found in the *paintGL(...)* function of the *GLWidget* class.