

Lab Assignment 5 – CSE 332/564, Spring 2008

Due: Tuesday, April 21, 11:59pm

This lab will introduce you to X-ray, MIP, and α -composited rendering. Continue to use the code you developed for lab 4. Upon completion of the assignment you will submit the following, via blackboard:

- the complete software (all that is needed to build the executable: source code, project files, etc)
- an executable (.exe file) of your work
- a comprehensive report that illustrates with images and narrative text all aspects of your work

All of these components are equally important. Please note the policies posted on the class webpage.

The overall theme of this project is to fill the interior of the lab 4 box with a volume dataset and visualize it with raycasting. For CSE 332 you may implement the lab either for the CPU or the GPU. For CSE 564 a GPU implementation is mandatory. Use the volume datasets given on the lab webpage. High-resolution versions of these and also additional datasets will be made available soon at the same place.

Note that the fragment code in the demo is NOT what this lab asks you to do (it uses different ‘technology’). Rather, for both GPU and CPU-based rendering, you should carefully examine the raycasting fragment code given in the book and reproduced in the class handout, which we discussed at length in class. You should follow the general concepts outlined there. But note, in contrast to the book algorithm, we will be using the box entry/exit information you computed in lab4 to get the ray direction information and determine if you exited the box. This is more general because it enables non-cubic volumes. Be sure you read the instructions provided in an extra handout on the class lab webpage for helpful information on GPU rendering.

You need four transfer functions: RGB and A (for opacity α), each of length 256. Simply use the transfer functions you already used for your previous labs, just extend by one channel. On the GPU they will fit into 1D RGBA textures.

For the GPU-based rendering the program flow is as follows, for a given view:

1. Set up FBO rendering (see lab page for a simple example)
2. Load the fragment program for raycasting, the 3D volume texture, and the transfer functions (if applicable). Also set the various other parameters required in the fragment program
3. The previous step generates fragments, one of each pixel ray. Each of these gets processed in the fragment shader. The only private parameter is the texture coordinate, returned from the rasterizer, which stores the entry position into the volume.
4. The rendering result is stored in the texture pointed to by the FBO. The final step is to copy the FBO to the display where it can be visualized.

The CPU-based rendering will follow the same principles. The only difference is that here you will replace the GPU fragment-parallelism by a CPU *for-loop*. This will make debugging somewhat easier and you will also not have to deal with the GPU setup. For a given view:

1. Render the box with the depth test set to `GL_LESS` and read the frame buffer with `GLReadPixels()`

2. This will give you the entry points into the volume for each ray. Now set up a for-loop for each entry point and run a program body equivalent to the GPU fragment program (do these calculations in floating point precision).
3. When casting the ray you will need to implement the 3D volume interpolation in software (the GPU simply calls a function). The same goes for interpolating the transfer functions and for normalizing the various vectors.
4. Store the final result into an image array and display it as a texture on the screen

The different parts of the lab assignment differ mainly in what you will do with the samples you interpolate along the ray. Check out the demo that has some of the requested functionality.

1. New menu items

Add a new menu item ‘Volume rendering’ and add to it three sub items: ‘X-ray’, ‘MIP’, and ‘Compositing’ (see below). Also add a new menu item ‘Interpolation’ and add to it two sub items: ‘Nearest Neighbor’ and ‘Linear’. Finally, on the GUI canvas add two sets of slider bars: (1) one slider named ‘Ray step size’ that can be set from 0.1 to 5 with increments 0.1, and (2) two sliders ‘Min’ and ‘Max’ that can be set from 0 to 1000 with increments of 1.

2. X-ray rendering

Sum the interpolated contributions. You will need to scale the rendered (floating point) image back into the displayable range (only for X-ray this is needed). Use the Min and Max slider bars to specify the value range that gets scaled between 0 and 255 (the values below Min get mapped to 0 and those above Max get mapped to 255). This will allow the image to get brighter or darker. By tuning Min and Max, you will be able to interactively enhance details (in the intensity range [min,max]), giving them a wider range of intensities on display. Make sure that you account for ray step size in the X-ray accumulation, or you get very large values for small step sizes. Just multiply the result by the step size, this will compute the accurate Riemann sum.

3. MIP rendering

Find the maximum density along each ray. You could use the Min/Max slider bars in conjunction with this to decide the range of values in which you make the comparison.

4. Composited rendering

Use the front-to-back compositing equations for rendering. To get the RGBA value at a sample index the corresponding transfer functions with the interpolated densities. Here it is assumed that these RGBA values are in the interval [0.0, 1.0] (you need to scale them into this interval or the compositing equations will not work out). Again, you could use the Min/Max sliders to bracket the active density interval. The simplest α -transfer function just uses a ramp, it may suffice for this lab assignment since you can use the Min/Max sliders to select certain volume features. Note, now your image will be in the range [0.0, 1.0], so you need to scale it back up to [0, 255] for display.

Extra credit (+10%): The composited rendering is also sensitive to step size. Let us assume a unit step that would produce a transparency of $(1-\alpha_l)$, where α_l is the value you get from the α -transfer function. If you have a number of steps in this unit interval, say n , then you’d get $(1-\alpha_l)^n$. Note here that n can be a

fractional number, for example, $n=1/0.3$ if you have a step size of 0.3. This would result in a much smaller transparency per unit interval, making the structures over-opaque. You need to correct for this. Let us assume that α is constant within this unit interval. Then you will need to scale α_1 down to a value α_2 to achieve $(1-\alpha_2)^n = (1-\alpha_1)$:

$$(1-\alpha_2)^n = (1-\alpha_1) \rightarrow \alpha_2 = 1 - \sqrt[n]{1-\alpha_1}$$

This means you would take the value obtained from the transfer function, α_1 , scale it down to α_2 , and use this α_2 in the compositing. Of course, in practice α_1 is not constant since the density slopes within the unit interval (this is why you did the smaller step sizes in the first place), but this equation will be a good approximation.

5. Study tradeoff speed vs. image quality

There are various parameters that influence speed and image quality: ray step size, image size, and interpolation filter. Take a dataset with fine detail, such as the lobster's legs to study these effects. A ray step size of 1.0 is generally considered the upper tolerance, above which important detail may be overlooked. Experiment with wider and smaller step sizes. Which one is faster? One important

Next, note that the volumes will render faster if you zoom out since less rays will be generated. Now take the rendered image and zoom it up in image processing mode, or save the rendering and enlarge with `irfanview` or read it back in as an image. Compare this image with an image that was rendered at higher resolution, which should be better. Typically, 3D rendering should be performed at the same resolution than the volume, or better. That is, cast one ray per voxel.

Now, compare the rendering obtained with nearest neighbor 3D interpolation and linear 3D interpolation. The difference is best observed when you render at or below the resolution of the volume dataset. Which is faster and why?

Finally, play with the color transfer functions and see if you can enhance certain volume features with it.

Extra credits (+10% each):

1. At times, the in-slice resolution is larger than the distance between slices. This can be due to the scanning procedure used in volume acquisition. Medical scans, such as MRI, or microscopy often acquire the data slice by slice. This will make the object appear flatter than it is in real life. You can account for this in raycasting by making the ray step size in this axis direction smaller. For example, if your z -resolution is $1/5$ of the x - and y -resolution, then you would step the ray at $(dx, dy, dz/5)$ where (dx, dy, dz) would be the direction vector you would have chosen for a uniformly sampled volume. This will make the ray in the z -direction virtually slower to account for the slices that are indexed too closely.

2. In the trackball interface, if you sense a large distance between mouse move events you could make the volume spin, that is, the trackball continuously rotates using the vector and plane set when the large movement was detected. Alternative interfaces for this are possible – just pick what you like best.