Projects for
Chapter 10: Volume Visualization

1 PROJECT 1

Implement the dense vector-field visualization method described by Exercise 1 for Chapter 10, using a volume rendering setting.

Hints:

1. As a basis implementation, use the CUDA-based volume raycasting code provided in the Samples for Chapter 10;

2. As a vector field for testing, make first a synthetic 3D vector field exhibiting a simple pattern, such as a spiral, cylinder, or point or sink. The advantage of this approach is that you can generate a simple *analytic* description of the vector field, thus you can next evaluate it at every desired location, and thus for any volume resolution, without having to implement interpolation. Once your design works with this field, consider any of the 3D vector fields provides in the online Datasets section;

3. Design several visualization methods for your vector field, starting from a simple to a more complicated one:

    - visualize the vector field magnitude, encoded in opacity, using a constant color;

    - visualize the vector field magnitude, encoded in opacity; and vector field orientation, encoded in color, using a directional color coding;

    - keep the above directional color coding, but modulate the opacity to also include the angle of the vector field with respect to the viewing direction; this way, make vectors that are parallel to this direction more transparent, and vectors that are

orthogonal to this direction more opaque; this should generate images where you can focus better on flow structures aligned with the view plane;

- implement volumetric shading, based on the gradient of the vector field magnitude.

Demonstrate each implemented method by at least one snapshot for each considered vector field.

2 PROJECT 2

Modify the CUDA-based volume raycasting code provided in the Samples for Chapter 10 to implement the following ray functions:

- maximum intensity projection
- average intensity projection
- distance to given scalar value
- isosurface

Demonstrate the effect of your new ray functions by visualizing one of the provided scalar volumes with the original (compositing) function and next with your new ray functions. Be careful to tune the various transfer function parameters (color and opacity) suitably for each different ray function.

Hints: This modification is relatively easy, if you first examine the CUDA code that is used to traverse all points along a pixel ray. In this code, the current ray function (compositing) is implemented. You can easily copy-paste this code, and next change it, to implement different ray functions. All above ray functions are described in Section 10.2, Chapter 10. Separately, for your new ray functions, take care to apply the color and opacity transfer functions *after* evaluating the ray value, rather than *during* the ray traversal, as it is currently done for the compositing function.

3 PROJECT 3

Implement a visualization of a 3D vector field using dense streamlines and volume rendering. For this, you can proceed as follows:

1. As a basis implementation, use the CUDA-based volume raycasting code provided in the Samples for Chapter 10;

2. As a vector field for testing, make first a synthetic 3D vector field exhibiting a simple pattern, such as a spiral, cylinder, or point or sink. The advantage of this approach is that you can generate a simple *analytic* description of the vector field, thus you can next evaluate it at every desired location, and thus for any volume resolution, without having to implement interpolation. Once your design works with this field, consider any of the 3D vector fields provides in the online Datasets section;

3. Generate a dense streamline trace of the 3D vector field. For this, study the streamline construction methods described in Sections 6.5.1 and 6.5.3. Store your traced streamlines as a voxel occupancy grid: That is, construct first a 'streamline density' scalar voxel volume encompassing the dataset's domain; initialize this volume to zero; next, for each voxel through which a streamline passes, increase the voxel's value by one. For best results, eliminate very short streamlines from this process.

4. Test that your volume is correct by visualizing it with direct volume raycasting, using a constant color and and opacity transfer function that makes high-density regions more opaque.

5. Refine the above approach by applying a small-scale blurring to the streamline density volume. For this, convolve the volume with a radial isotropic Gaussian filter (*e.g.* 3^3 or 5^3 voxels in size). This will create 'halos' around high-density regions crossed by many streamlines, thus make their structure look thicker in the final visualization.

6. Experiment with different visual refinements to obtain the best-looking image, *e.g.* by using volumetric shading, coloring data by vector field magnitude, or using different opacity transfer functions.

Demonstrate your implemented method by at least one snapshot for each considered vector field and visualization option implemented.

4 PROJECT 4

Implement the 3D shape reconstruction from different 2D views described in Exercise 11, Chapter 10. In detail, the aim of this project is to show how we can reconstruct a (good) approximation of a 3D shape from a (large) number of 2D views of that shape.

Note: The instructor should make the solution provided for Exercise 11, Chapter 10 available to students prior to executing this project. This way, the students have a guided design, and

can focus on algorithmic issues rather than searching for a (complex) solution to the problem. Separately, it is much easier to check/guide their work if the students are given design guidance at the beginning.

To do this project, proceed as follows:

1. Download and get familiar with the (simple) sample program provided for Chapter 3 for the representation and visualization of 3D unstructured meshes. The program allows you to load a 3D mesh (in PLY format) and to visualize it from any view point.

2. Modify the program to draw the shape so that the color of any shape pixel is clearly different from the background color. For this, you can simply change the OpenGL material properties, *e.g.* make the shape green or blue. Also, make sure you remove strong specular effects, since these may create white pixels on the shape. Alternatively, you can leave the shape drawing as is, but use a different background color, *e.g.* red (this can be set using `glClear`). Also, make sure you use orthographic rather than perspective projection.

3. Design a function to programmatically grab a snapshot of the image currently displayed in the graphics window. This is simple to do using the OpenGL function `glReadPixels`. Test that the grabbed image is correct, *e.g.* by exporting it in a simple image format you can view offline, such as PPM. Code for writing PPM files is provided in the online samples for Chapter 9.

4. Together with each snapshot, save also the camera parameters that the snapshot was taken from (camera origin, viewing direction, up vector). You can save all these *e.g.* in a text file.

5. Implement a procedure to automatically create a large number of samples (snapshots and corresponding viewpoint information) from your viewing application. For instance, you can create a random uniform sampling of all viewing positions around the shape (with the view vector pointing at the same position, *e.g.* the shape's center), or you can interactively manipulate the viewpoint yourself and 'record' all the generated views.

6. Based on the set of recorded views and their corresponding viewpoints, implement the space carving solution outlined in Exercise 11, Chapter 10. For this, the first step is to implement a binary voxel volume of suitable resolution that will store your carved shape. Initialize the entire volume to foreground. Next, for each image s_i and corresponding viewpoint v_i, trace a ray from each background pixel in s_i, and mark all voxels along this ray as background. The fastest way to do this is to modify the CUDA-based volume raycaster code provided in the Samples for Chapter 10.

7. After processing all views, you are left with a binary volume which should show the re-

constructed shape. Verify the reconstruction quality by visualizing this volume using the CUDA-based volume raycaster code provided in the Samples for Chapter 10. Compare your result, seen from different angles, with the input mesh rendered from the same angles.

To demonstrate the quality of your reconstruction, show the reconstructed shape from several angles side-by-side with the input (mesh) shape rendered from the same angles. Do this for a few different shapes, of which some also have concavities. Show how the reconstruction result improves as you use more views.

End of Projects for Chapter 10: Volume Visualization