John Wiggins
CS377 Semester Project
4/1/2003

Design and Implementation Plan

I will start with a brief overview of what volume raycasting is, then describe my implementation. Volume raycasting is a direct volume rendering method which roughly approximates the effect of light passing through a volume dataset. A volume dataset might be a number of CT slices, MRI data, the visible human, or any number of datasets from a scientific computer simulation. Frequently, the data is sampled at regular intervals within a 3D regular grid. These samples can be scalar values, vectors, or tensors. For the purpose of this project, all datasets are assumed to be scalar values (usually density) sampled on a regular grid.

The actual process of raycasting is quite simple. An "eye" point is chosen somewhere in 3D space relative to the volume being rendered. A rectangular grid of pixels sits somewhere between the eye and the volume. This is the image plane. Rays are cast from the eye through each pixel in the image plane into the space beyond. (usually into the volume) At a regular interval on each ray, the density of the volume at that point is computed using trilinear interpolation and a special transfer function. A gradient is also computed at the point for input into a shading equation. These densities and gradients along the ray are accumulated into a color for the pixel that the ray is associated with. In this way, the dataset can be visualized in a flexible manner.

For the design of my parallel volume raycaster, I have divided the work into to two parts: management of the dataset and rendering of the output. These two parts will be handled by two distinct parts of a mixed MPI/pthreads program. This means that the program will require a minimum of two computers to run.

Here is a basic rundown of the execution:

- after the program starts, the nodes assigned to management of the dataset will partition the dataset into equal sized blocks which cover a cubic region of the dataset. These blocks will be distributed uniformly among the nodes which manage the dataset.
- One the dataset has been partitioned, the program is ready to render any number of images desired. This is mostly a consideration so that the program may later be deployed as a back end render server.
- To render the image, the image is first subdivided into smaller rectangular patches. These patches are uniformly distributed among the nodes assigned to rendering.
- The render nodes now render their allotted patches. To do this they:
  - create a perspective viewing frustum for the patch they wish to render.
  - intersect the frustum with the dataset volume in order to determine which blocks will be needed to render the patch. Once the blocks are known, they are sorted into a mostly near to far ordering.
  - request the blocks in order from the nodes which are managing the dataset.
  - concurrently in another thread, the render node can begin casting rays into the volume. If a ray encounters data that is not yet resident in local memory, that ray's state is saved and it is put aside while other rays are computed.
  - after all the blocks have been received for the current patch, blocks for the next patch are determined and fetched, hopefully before they are needed. This happens concurrent to the rendering process.
- Once all render nodes have finished rendering, their outputs are merged together (a simple composition) and the result is written to disk.

The process of raycasting requires very little communication between processes. For the most part, I will be using MPI as a transport for parallel I/O. Aside from I/O, it will be used at the beginning of execution to distribute

program parameters and at the end of rendering as a means for syncronization an merging of the result. Pthreads will be used for concurrency within each node. This has already been outlined for nodes which render. For nodes which manage the dataset, pthreads will be used such that one thread receives requests for data, while another thread fulfills those requests.