VoxGrabber: A Tool for the Extraction of Tubular Structures from 3D Volumes for Transmission and Storage

Kyle Olszewski, Piyush Soni, Jarek Rossignac
Graphics, Visualization and Usability Center, Georgia Institute of Technology,
Atlanta, GA 30332
kyleo@gatech.edu, piyush_soni@gatech.edu, jarek@cc.gatech.edu

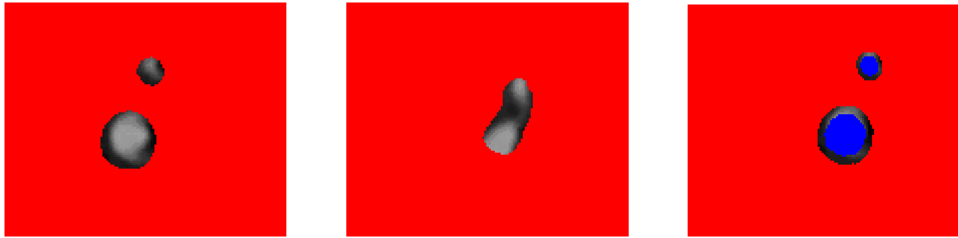

***Figure 1**: (Left, Center) Different slices of the 3D volume created while rasterizing a Pearling Tube. The red pixels in the image represent voxels which are not extracted from the original 3D volume. (Right) An example of a Pearling Tube containing strings of pearls with interior radii defining a "crust" of relevant voxels. The blue pixels indicate voxels which lie inside this interior region and thus are outside the "crust" which is extracted. . (Two circles indicate that the tube is going out and coming into the plane)*

Abstract

While many techniques exist for the compression of 3-dimensional medical imaging data, these techniques are not widely used in medical applications, in part due to concerns of losing important information stored in the data. However, in many cases, a viewer is only interested in a relatively small section of the data stored in the volume, making the space and time required for storing and transmitting the other data in the volume an unnecessary overhead. Here we present our tool, *VoxGrabber*, which quickly extracts voxels from a large 3D volume and stores them in a much smaller data file, representing some extracted region of the input volume containing a tubular structure, such as a blood vessel. The tool then reads the extracted voxels and uses them to reconstruct the relevant region of the image. We describe several approaches to this process we have explored and their theoretical and practical advantages and limitations. We then describe the results of our research and experimentation with these approaches, explain our reasons for selecting the particular approach used in our final application, and give results from executing this program with several sets of input data. Finally, we describe how the tool could be used with an appropriate predictive compression and encoding scheme so as to provide lossless compression of the volume data, further reducing the size of the data to be stored or transmitted.

Introduction

In present 3D medical imaging applications, much of the data stored in a volume is often of little or no concern to the viewer, making storing that data or transmitting that data to a viewer an unnecessary burden. For example, viewers are often interested in extracting certain tubular structures, representing bones, blood vessels, etc., from an image. By using some data file provided by the user representing these relevant regions of the image, we may *extract* the voxels contained within this region from the image, and either store them in a much smaller data file, or transmit them over a network to a viewer application much faster than we could

transmit the entire volume. (Hereafter, for conciseness, we refer to the process of storing the extracted data in an output file or transmitting it to a viewer as “sending” the data to the “output destination.”) One may do so by sending the x, y, and z coordinates of the extracted voxels with the voxel intensity values, but this leads to additional data which must be sent to the output destination.

However, given some algorithm for sequentially extracting voxels from a volume, and the same small data file describing the extracted region, the recipient may reconstruct the 3D image by iterating over the received data in the same manner as the extractor, thereby determining where to place these voxels in the viewing region. As demonstrated by our results below, this process can lead to a large reduction in the amount of data which must be sent to the output destination. Furthermore, our approach lends itself to further data size reduction via the use of one of several lossless compression schemes, which we describe at the end of the paper as potential extensions of our tool which we intend to explore. While extracting the useful voxel data, a suitable prediction and compression technique can be used to reduce the total number of bits required to send these voxels. The client, running the same extraction algorithm, knows the points it has to render as it receives the bits representing the intensity values of the voxels in the order in which they were extracted, in some compressed format. By using a pre-defined decompression algorithm, the user may use these bits to reconstruct the original intensity values extracted by the sender and place them at the correct points in the rendered image.

The volumetric data we consider in our analysis are medical images segmented by the *Pearling* [2] approach. A set of “Pearls” consists of a list of spheres of varying radii, whose union defines a tubular data structure (see Fig. 2). For simplicity, in our implementation we only consider pearl strings in which adjacent spheres in the list intersect one another, and which contain no bifurcations (i.e., there is only one major string with no branches.) In particular, we will describe how we deal with pearl strings which also possess an interior radius defining an empty region surrounded by the outer pearls, thereby creating a “crust’ to be extracted from the image (Fig. 3). Our approach further reduces the total amount of data to be sent by not extracting the voxels contained within this empty region inside the crust.

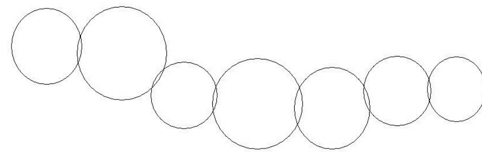

Fig. 2. A 2-dimensional representation of a string of Pearls. This string of adjacent spheres defines some tubular structure, such as a blood vessel, bone segment, etc.

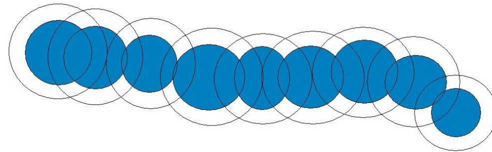

Fig. 3. A string of Pearls containing interior radii (the blue spheres) defining a hollow interior region.

Data Structure

To read the pearling values and their intensities we create a doubly linked list, used to store a simple data structure representing the pearl:

```
class Pearl
{
	int pearlNum;
	// the radius and square
	// of the radius
	float radius, radSquared;
	Point3D Center;
	Pearl *next;
	Pearl *prev;
};
```

For simplicity, when reading pearls which contain an interior radius, we simply create a second string of pearls to represent this empty interior space.

Subdivision

The input pearls may not be very well refined, meaning that there may not be a large amount of overlap between adjacent pearls in the string, if their centers are too far apart and their radii are too small. We seek to extract relatively smooth tubular structures from the image, so we first subdivide the pearl string before using it in or extraction algorithm. Simply subdividing the pearl string by linearly interpolating the positions and radii of adjacent pearls will lead to a subdivided string of pearls which resembles a string of connected line segments (see Fig. 4). Ideally, we seek a subdivision algorithm which more accurately captures the curvature of the original string of pearls. We thus use 4 Point Subdivision for subdividing the positions and radii of the pearls.

To do this, we find a Cubic interpolation between every group of four pearls, such that the final curve passes through all of them [1]. (We note that there are numerous other subdivision techniques as discussed in [1] which could be used instead.)

Let there be four points A, B, C and D from which we want to get a new subdivided 5th point E by using the 4 point subdivision.

Let the cubic curve be:

$$f(t) = at^3 + bt^2 + ct + d$$

Here, t is the curve parameter. For simplicity and without losing the generality, let us assume that the points A, B, C and D are given by the values of f(-3), f(-1), f(1), f(3). Then we can write that the point to be generated is given by f (0).

$$-27a + 9b - 3c + d = A$$
$$-a + b - c + d = B$$
$$27a + 9b + 3c + d = C$$
$$a + b + c + d = D$$

Solving them for d = f (0), we get

$$E = f(0) = d = 9/16(B+C) - 1/16(A+D)$$

Fig.4. (Top) Linear subdivision gives us a string of pearls which resemble a set of connected cones. (Middle) 4-Point subdivision more accurately captures the curvature of the original string. (Bottom) Both of these approaches still result in "ridges" between the pearls, which we seek to remove by further interpolating the radii of adjacent pearls to include the values within these ridges.

The groups of three pearls which are at the beginning and end the pearl 'tube' need one more "dummy pearl" to be subdivided properly. This 4th point can be inserted on both the boundaries by using some

prediction scheme. Here using a simple linear predictor serves our purpose sufficiently. Figure 5 shows a diagram of the Linear Predictor. It extends the last edge along the direction of its vector, with the same magnitude of the edge to find a new point P. We discard these "dummy pearls" after using them for our subdivision algorithm.

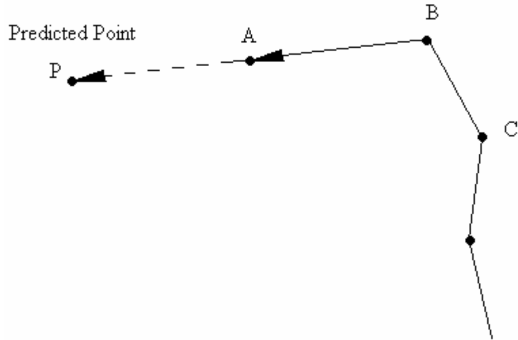

Fig. 5: Linear Prediction. This is used to determine the position of the subdivided pearls to be inserted into the pearl string for the first and last 3 pearls in the string when using 4-Point Subdivision.

The radii of the pearls are subdivided by the same scheme, so that the radius of each of the subdivided pearls is given by:

$$R_E = 9/16(R_B+R_C) - 1/16(R_A+R_D)$$

Voxel Extraction Techniques

The majority of our efforts on this project consisted of researching, implementing, and analyzing various methods for efficiently and accurately determining which voxels lie within the region to be sent to the viewer.

We ultimately devised 3 very different approaches, each with different characteristics in terms of performance time, memory usage and other advantages and disadvantages. Below, we describe each of these implementations, our results from experimenting with them, and our final conclusions as to the merits of each.

Sphere Rasterization

One important concept we must discuss, used in several of these implementations, is a simple and efficient technique used for sphere rasterization, which we describe below.

Suppose we wish to determine if each voxel in a 3D volume lies within the radius of a given sphere. Let the distance of a voxel with coordinates (x,y,z) be 'd' from the center of a sphere of radius 'r' and center (x_P, y_P, z_P) at any instance. Then, we may write that

$$d^2 = (x - x_P)^2 + (y - y_P)^2 + (z - z_P)^2$$

$$= x^2 + y^2 + z^2 + x_P^2 + y_P^2 + z_P^2 - 2x.x_P - 2y.y_P - 2z.z_P$$

Thus, determining whether the voxel lies within the sphere requires performing several multiplications, additions, and subtractions, and then taking the square root to determine the distance, before comparing the result to the sphere's radius.

However, suppose that we have calculated the distance of a given voxel to the sphere center, and wish to find the distance of the neighboring voxel at position (x+1,y,z). Because the x value increases by 1 when moving to this voxel, the new distance can be given by

$$d_{new}^2 = (x+1)^2 + y^2 + z^2 + x_P^2 + y_P^2 + z_P^2 - 2(x+1).x_P - 2y.y_P - 2z.z_P$$
$$= d^2 + (x-x_P) + (x-x_P) + 1$$

Thus, by using the results of the previous calculation, we may calculate the new squared distance using only a few addition operations. If we maintain a record of the squared sphere radius, we may compare the squared distance value for each additional voxel to the squared radius of the sphere in order to determine whether or not it is within the sphere. Hence, the square root and multiplication calculations for each voxel, which are relatively expensive operations,

can be replaced by just simple addition operations, which are relatively inexpensive. Using the same basic approach, we may determine the new squared distance when moving in the y and z dimensions.

First Approach: GPU Implementation

We now describe the first voxel extraction algorithm with which we experimented. The motivation behind studying this approach was to use the computational resources of modern GPUs, which possess many multiprocessors which may quickly perform rasterization operations, in the extraction of the relevant voxels. In brief, this approach works as follows. We sweep what we call a "scan plane" from the bottom of the input volume to the top, starting at the plane defined by z=0 and increasing the z value until this plane has passed through the entire volume. As we move, we keep track of which pearls currently intersect this plane. The intersection of this plane with a sphere forms a circle on the scan plane. For each z value we reach as we move from the bottom of the volume to the top, we determine the position and radius of the circles which currently intersect this plane, and draw them as white circles in a frame buffer (in which each pixel is initialized to black for each plane) on the GPU. After drawing all of the circles on the current plane, we simply iterate through the pixels, in the resulting image, checking to see which ones are white. These are the pixels within the intersection of the current plane with the pearls, and thus are to be sent to the viewer. We repeat until we have passed this plane through the entire volume, and thus have sent all of the relevant voxels in the volume.

We use a technique based upon the sphere rasterization approach described above for efficiently calculating the squared radius of the circle that will be intersected by the current scan plane. Observe figure 6, which gives a 2-dimensional representation of the intersection of the scan plane with a sphere positioned at the origin. Let z_i be the current z value of the plane. We seek to find what we call the "current radius" r_i, which is the radius of the circular intersection of the sphere with radius R on this plane.

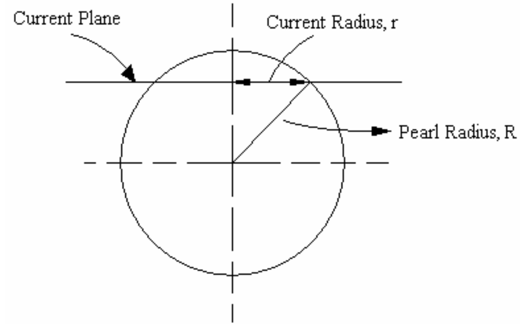

Fig. 6: Calculating the current radius r of the circular intersection of a pearl and the plane.

It is clear that that the square of the current radius will be given by

$$r_i^2 = R^2 - z_i^2$$

Increasing z by 1 will change the current radius to

$$\begin{aligned} r_{i+1}^2 &= R^2 - z_{i+1}^2 \\ &= R^2 - (z_i+1)^2 \\ &= R^2 - z_i^2 - 2z_i - 1 \\ &= r_i^2 - z_i - z_i - 1 \end{aligned}$$

Thus, after initializing the current radius r_i for a given pearl to 0 when we reach the lowest z value at which we intersect this pearl, and initializing the z_i value for this pearl to (–pearl.radius), we may calculate the squared radius of the circular intersection of a sphere with the scan plane using only a few simple subtraction operations. We then update the z value for the next iteration, and repeat until the scan plane passes the maximum z value of the pearl. Unfortunately, however, due to the fact that we must pass the radius of the circle to be drawn to the GPU, we must still take the square root of this value in order to determine the actual radius of the circle at this point.

We implemented this approach using OpenGL commands in the GLUT toolkit. To speed up the performance when drawing circles, we created an OpenGL display list consisting of a single triangle fan consisting of 36 triangles creating a rough approximation of a white circle with a center at the origin (0,0) and radius = 1. When drawing the circle for each circular intersection with the scan plane, we use matrix operations on the GPU to scale this circle by the radius of the current circle and translate this circle to the (x,y) position of the current circle.

More explicitly, our algorithm works as follows:

1. Read the input data file, storing the pearl string in a linked list of nodes containing the relevant information (center position, radius, etc.).
2. For each pear in the list:
   a. Create 2 "Events:"
      - Entry, containing the z value of the "bottom" of the pearl, defined by (pearl.Center.z – pearl.radius)
      - Exit, containing the z value of the "top" of the pearl, defined by (pearl.Center.z + pearl.radius)

   b. Place these 2 events into a linked list sorted in ascending order of the event value. This list gives us a schedule defining when each pearl becomes active and inactive as we iterate across the z-axis (starting from the bottom of the volume, and proceeding to the top.)
3. For each 2D "slice" of the volume, from i = 0 to the top z value:
   a. Clear the frame buffer (using glClear()) to set each pixel to black (0,0,0)
   b. Check the schedule to see if any pearls become "active" at this slice, meaning that i is equal to the "Entry" value for this pearl. Add all of these pearls to a linked list of active pearls. Initialize the value of the radius r of the pearl's circular intersection with the currently active slice to 0, and the z value for this circle to –pearl.radius.
   c. Check the schedule to see if any pearls become "inactive" at this slice, meaning that i is equal to the "Exit" value for this pearl. Remove all of these pearls from the linked list of active pearls.
   d. For each active pearl:
      - Call the display list created above. Scale the circle's radius from 1 to the radius of the pearl's current intersection circle. Translate the center of this circle to the (x,y) position of the center of the pearl and draw the circle. This gives us a circle in the frame buffer representing the intersection of this pearl with the currently active slice.
      - Update the radius value and z value of the pearl's intersection circle for the next slice, i+1, using the formula described above.
   e. Use glRead() to read the pixels of this image. Iterate through the pixels of this buffer, marking which voxels have been turned "on" (i.e., which ones are "white" due to the fact that they fall within one of the intersection circles which are drawn), and marking the corresponding voxels as needing to be sent to the output destination.

The primary advantages of this approach are:

- Memory coherency: performing the algorithm in this fashion allows us to acquire the relevant voxels using only 1 2-dimensional slice of the input region at any given time. This prevents us from having to allocate and use a 3-dimensional region of memory representing which voxels in the 3D volume are to be "sent," which could lead to inefficient use of memory and performance degradation.
- GPU Computation: Using the GPU, which possesses many symmetric

multiprocessors used to process many pixels in parallel, allows the computation of many of the frame buffer "pixels" (each of which represents an input voxel on the current scan plane) to be performed simultaneously. This approach was based on the presumption that using these resources would lead to less time required for performing these computations than performing the same basic algorithm using the CPU.

However, despite these advantages, this approach has several drawbacks, which ultimately led us to discard this approach when developing our final application:

- Using linked lists in this fashion requires sorting and searching the lists, which may lead to a fair amount of computational overhead. Creating the entry schedule described above requires inserting many entry and exit events into a sorted linked list. Removing a pearl from the active list while iterating through the z values of the volume requires searching through the active list for the pearls to be removed. If the input pearls are arranged such that many pearls are active at a given time (i.e., many pearls intersect the current 2D plane), then searching through this list for each slice could lead to unwanted overhead. For very long strings of pearls, these factors could lead to a significant amount of computation.
- Once we have read the pixels off of the GPU for a given slice, we must still iterate through the 2-dimensional image in order to determine which pixels have been "turned on" and thus must be sent to the output destination. For a string of pearls which covers only a small region of the input volume, this requires checking many pixels which are "off" in order to determine which pearls are "on". We may minimize this delay by computing a 2-dimensional min-max bounding box around the pearl string in the input volume upon reading in the input pearl string, and only iterating across the pixels contained in this box. However, for long, strings containing many relatively small pearls, this bounding box may be quite large, despite the fact that only a small number of the pixels contained in it are to be "sent." Ideally, we would like an algorithm which minimizes the total number of voxels outside of the region to be sent which are "touched" during the computation. This led to our exploration of the 2-dimensional and 3-dimensional region-growing algorithms described below.
- Redundancy: For each pearl which is intersected by the currently active slice of the input volume, we draw a circle to the frame buffer. For pearl strings which have been subdivided to the extent that there is a great deal of overlap between adjacent pearls, this may lead to a fair amount of redundant computation. For example, if we have 2 circles whose union contains the majority of the area covered by these circles, drawing the second circle will add only a small number of additional voxels to the region marked as "sent."
- Ridges: Though we use subdivision to produce more refined tubular structures in the output image, as we do not interpolate between the adjacent pearls when drawing these spheres, we still possess small ridges between adjacent pearls, as seen in Fig. 4 above.
- Hardware: Efficiently executing this algorithm requires that the users possess reasonably modern graphics hardware. For users with older systems, which may possess outdated graphics hardware, the results may be much less impressive. For users with no graphics hardware, who must perform these operations using CPU-based software rendering, this approach is most likely less efficient than other approaches which do not rely on OpenGL rendering.
- Rasterization: Suppose that the voxel extraction and the image reconstruction are performed on separate systems (such as when sending the data across a network). This algorithm depends upon the assumption that using the same OpenGL commands on 2 separate systems, which may possess different graphics hardware, will produce images which are identical pixel-for-pixel to one another. However, it is possible that

users executing this algorithm on different types of hardware may produce slightly different results when rasterizing the circles. In our application, the sequence of voxels encountered by the sender and receiver must be exactly identical. If the sender turns a pixel "on" while the receiver leaves it "off," then the receiver will assume that that voxel remains unsent, and will use the intensity of this voxel for the next pixel which is considered to be "on." Thus, this approach is only feasible if we find some way of ensuring that the rasterization hardware of both users produce identical results.

- GPU Read delays: The most significant disadvantage of this approach, which we discovered while running our application on several sets of input data, was that reading in drawn pixels from the frame buffer on the GPU required more time than expected. On each of the computers on which we tested this application, reading a single 512x512 pixel image from the GPU required an average of 34 ms. While only a small delay for a single image, our algorithm requires performing this read for each of the slices of the input volume in the z dimension. Thus, performing these reads for a 512x512x512 input volume required roughly 17.4 seconds, which accounted for a majority of the 19.2 seconds of the application's runtime when using an input string of 17 pearls (subdivided to 33 pearls) which covered only approximately 40,000 voxels. When optimizing our program, we attempted to minimize this delay by computing the minimum and maximum z-values of the pearl string, and only performing the algorithm for the slices between these values. This reduced our runtime to roughly 2.4 seconds for the small input pearl string used above (in which the pearls only covered a small range of the z-axis). However, the time required to read the image pixels still proved to require the majority of this time, and pearl strings which cover many of the scan planes used in the algorithm will incur a much greater penalty, as reading these additional frames from the GPU will lead to even more transfer delays.

Despite our extensive efforts in implementing this extraction algorithm and our hopes for utilizing the GPU to reduce the computation time for our extraction algorithm, we ultimately abandoned this approach for our final tool due to the aforementioned practical limitations. We explored several alternative algorithms for our voxel extractor, which we describe below.

2. 3D Region Growing Implementation:

Upon discovering the limitations of out previous approach, we developed a second algorithm designed to address the limitations of the previous algorithm. This algorithm was inspired by the 2-dimensional "flood-fill algorithms" used in illustration applications designed to fill a region of adjacent pixels which are the same color with a new color. The basic algorithm is simple:

```
Flood Fill (x,y, inColor, outColor){
  if(pixel(x,y).color == inColor){
    pixel(x,y).color = outColor;
    Flood Fill (x+1, y,   inColor, outColor);
    Flood Fill (x-1,  y,   inColor, outColor);
    Flood Fill (x,    y+1, inColor, outColor);
    Flood Fill (x,    y-1,  inColor, outColor);
  }
}
```

We may extend this algorithm to 3 dimensions by adding calls to the function for the "up" and "down" neighbors of a voxel, i.e., the neighbors at positions (x, y, z+1) and (x, y, z-1). For our region growing algorithm, however, we must check 2 conditions: whether the voxel has already been "touched," i.e., sent and placed on the stack by the algorithm (so we do not extract the same voxels multiple times), and whether the voxel lies within the region covered by the pearls. Thus, we keep a 3-dimensional array of boolean values used to mark whether a given voxel has been "touched" (i.e., sent to the output destination and placed on the stack) by the algorithm.

We send and mark one seed voxel which we know to fall within the relevant region, and then place it on the stack. This voxel then checks to see which of its 6 neighbors (in the x, y, and z dimensions) are also untouched and fall within the relevant region. These voxels are then sent, marked as touched, and placed on the stack.

Implementing this algorithm in this intuitive manner initially resulted in stack overflow errors, due to the many recursive functions calls as the "sent" region expands throughout the 3D volume. We resolved this issue by replacing the recursive function calls with a user-managed queue of nodes.

Trivial implementations of this algorithm would start by initializing a single seed voxel within the first pearl and growing the entire sent region outwards from there. Doing this, we would have to find the distance of each voxel from the center of each of the input pearls, and compare that with the radius of the pearl in order to determine if the voxel falls inside or outside any of the pearls. If this voxel is sent and its neighbors are placed into the queue, we would have to update the distance of these voxels from each of the pearl centers as described above. Thus, if the input pearl strings are quite long, many comparisons and computations may be required for each voxel in order to determine whether or not it falls within the relevant region.

Our implementation, however, is designed such that, for the majority of the voxels, we must only compare their positions to 1 or 2 pearl centers in order to determine whether it falls inside the region to be sent. We accomplish this as follows.

1. For the first pearl, we simply initiate the algorithm using a seed voxel placed at the center of the pearl, which we call A. We continue growing the "sent" region using the 3D growing algorithm as described above until we reach the boundary of this pearl (see Fig. 7). The "sent" region will thus grow to include all of the voxels contained within this pearl.

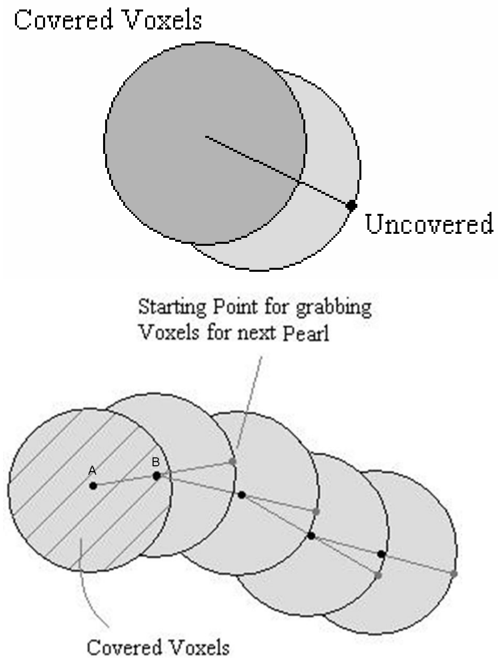

Fig 7: Voxel Growing Algorithm. (Top) once all of the voxels in a given pearl have been "sent," when growing the sent region to include the voxels contained in the next pearl, we must only grow the sent region to include the voxels contained in the next pearl but not in the previous one. (Bottom) For the first pearl, A, we initialize the algorithm using the pearl at its center as the seed voxel. For the next pearl B, we scale the unit vector in the direction from the center of A to the center of B by the radius of B, and add it to the center of B, thus beginning our region growing at the point in B which is farthest from the center of the previous pearl. Thus, unless the 2 pearls overlap entirely, this voxel will not already be "touched," and the grown region will grow to include all of the voxels in B which are not also in A. We repeat this process for each following pearl, which extracts the voxels which are within itself but not within the previous pearl (and thus have been sent).

2. For the next pearl, we first find the vector AB from the center of the previous pearl, A, to the center of this pearl, B, and divide it by its length to obtain the unit vector AB/|AB|. We scale this unit vector by the radius of the current pearl, B.radius, and add it to the center of the current pearl B.

The starting point for the region growing algorithm for pearl B is thus:

B.center + B.radius*(**AB**/|**AB**|)

We subtract some small epsilon value from this vector to ensure that the resulting seed voxel does not fall on the exact boundary of the interior of this pearl (and thus might be rejected when we check to see if it falls within the radius of B). Thus, we begin growing the region covered by this pearl at a voxel near its boundary, at a point within this pearl which is the farthest from the previous pearl, as seen in Fig. 7. The current pearl will only have to expand the "sent" region to send the voxels which are not contained in the previous pearl. Once it has finished sending its relevant region, we calculate the starting point for the next pearl, C, in the same manner and proceed to grow its region. Thus, the only voxels whose positions are compared to multiple pearls are those which lie near the boundary of one pearl and its neighbor (i.e., one pearl checks and sees that it is outside of its boundary, so it is not sent; the next pearl will grow its region until it reaches this voxel on the boundary between the 2 pearls, which it will then send.)

To keep track of the squared distance of a given voxel from the center of the current pearl, we find the squared distance for the seed voxel, and then update it for each new voxel placed in the queue using only addition operations as described in the sphere rasterization section above. This allows for relatively fast calculations of the squared distance values for each voxel. Comparing these to the squared radius of the current pearl, which we calculate when we place the first seed voxel within the pearl, allows us to quickly determine whether each voxel in the queue lies within the current pearl.

Thus, the primary advantages of this approach are:

- Only a few fast calculations are required in order to check if a voxel lies within the region to be sent, and that the majority of the voxels will only have to compare their position with a small number (1 or 2) of the pearls in the string in order to determine whether or not it is to be sent, rather than comparing their position to each pearl in the string.

- Unlike the previous approach, we do not have to iterate through each of the voxels in each of the 2D slices of the volume checking to see which of the voxels are to be sent. Thus, the only voxels which are "touched" by the algorithm are those which are within the pearls and those voxels which are on the boundary of the pearls and the outer region.

The primary disadvantage is:

- Memory usage: As mentioned above, due to the fact that the grown region may expand in 3 dimensions requires us to keep a 3-dimensional array of boolean values marking which voxels have and have not been sent. For a large volume, such as the 512 x 512 x 512 volume we used when testing this algorithm, this leads to a large amount of memory (131 Mbytes, if 1 byte is used for each voxel) required simply to hold this array when running this algorithm. We attempted to optimize this algorithm by computing a bounding box around the pearl string, and only allocating enough memory for the voxels in the bounding box, but for long strings of relatively small pearls, this still results in a relatively large section of memory being allocated, of which only a small portion is actually used.

3. 2D Region Grower

Though our initial tests with the previous approach indicated it was much more efficient than the first approach, we

ultimately encountered difficulties when attempting to implement this approach in the final voxel extraction application (see "Results" below). As a result, and on the advice of Professor Jarek Rossignac, Georgia Tech., we ultimately abandoned this approach in favor of our third and final approach. In the following section, we describe the same.

Here we combine some of the more effective aspects of each of the previous approaches. We again perform a region growing algorithm using a user-managed queue, but this time we only grow the region in 2-dimensions at any given time, processing the input volume 1 2D-slice at a time, as in the first approach. While this approach performs more computationally expensive operations than the second approach, such as multiplication and square root calculations, it has the added advantage that it includes the voxels contained in the "ridges" between adjacent pearls, as described above and seen in Fig. 4 above, in the relevant region, resulting in a much smoother tubular structure being extracted from the input volume.

The basic algorithm works as follows:

1. We iterate through each of the slices of the 3D volume from minimum z value of the pearl string to the maximum z value of the pearl string, performing the following steps:

a. For each pearl, we compute the projection of its center onto this plane, and place this voxel onto the queue. Thus, at the plane given by z = 37, for a pearl with center (x,y,z) = (54,12,65), we would place the voxel at (x,y,z) = (54,12,37) onto the queue.

b. We then perform a 2-dimensional region growing algorithm, checking to see if the voxels in the queue fall within the radius of any of the pearls, or within the ridge between them (as described below). If a voxel in the queue falls within the pearl, the voxel is "sent" to the output destination, and it places its 4 neighbors in the x and y dimensions only onto the queue (if they are not already "touched" by the algorithm). We repeat this process until the queue is empty, meaning that all of the relevant voxels for this slice of the input volume have been "sent."

Thus, for each pearl which currently intersects the scan plane, we will have a seed voxel which will grow the sent region to include all of the neighboring pearls which are included in the relevant region. We now describe the test used to determine whether a voxel lies within the relevant region. For each voxel P, we perform up to 3 tests for each set of adjacent pearls A and B in the pearl string (see Fig. 8 below) the first 2 tests are simply as follows:

1. We find the length of the vector AP from the center of A to P, and check to see if it is less than the radius of A.

2. We find the length of the vector BP from the center of B to P, and check to see if it is less than the radius of B.

These 2 tests are sufficient to see if the voxel lies within the relevant region, as defined in the previous 2 implementations. For subdivided pearl strings in which adjacent circles are very close to one another, these 2 tests will acquire most of the voxels in the relevant region. However, with this implementation, we also seek to eliminate the "ridges" between adjacent pearls, as described above. To do this, we linearly interpolate the radii of the adjacent pearls as seen in the image below, and include the voxels within this region as part of the relevant region. To check to see if a pearl falls within this region, we do the following test:

3. We take the vector AP from the center of pearl A to voxel P. We take the vector from the center of A to the center of B, which we call AB.

By computing the dot product AP*AB and dividing by |AB|, we find the scalar projection of vector A onto B. This gives us the closest point Q on the vector AB to the voxel P. We linearly interpolate the radii of A and B, based on the position of Q on the vector between them, and compare this value to |AQ|. This test thus tells us whether the point P falls within the linearly interpolated radius of A and B. See the figure below for an illustration.

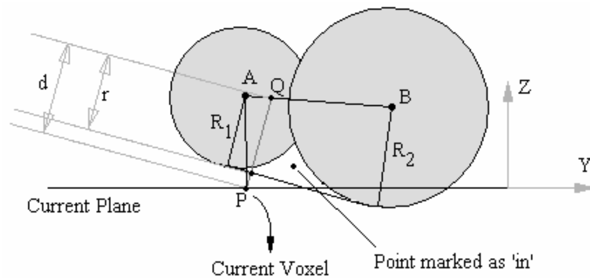

Fig 8: Testing containment of voxel P in the 'cone' joining two pearls A and B. We compute the scalar projection of the vector from A to P onto the vector from A to B, and use the distance of this point Q from P to see if it lies within the value r given by linearly interpolated the radii of A and B at point Q.

The primary advantages of this approach are as follows:

- Ridges: as mentioned above, we compute a smoother tubular structure by interpolating the radii of adjacent pearls when performing our tests, as described above.

- Memory coherency: Though this approach is computationally expensive, due to the fact that the queue only grows in 2 dimensions, it does not store as many of voxel values in the queue at any given time compared to the previous approach. In addition, as we process the volume slice-by-slice for marking which voxels have been "touched" by the algorithm, we only need one 2-dimensional array which we may clear after each slice has been finished, rather than a 3-dimensional array marking which voxels have been touched in the entire 3-dimensional region. Thus, the memory requirements for this approach are much lower than the previous approach.

The main disadvantage of this approach is its computational complexity, which is due to the following factors:

- This test performed for each voxel is more computationally expensive than the previous approach, as it requires computing the dot products and lengths of several vectors, and therefore requires using several multiplication, division, and square root operations.

- In addition, we must perform the tests described above for each set of adjacent pearls for each voxel, as simply checking to see if the voxel lies within the region between 2 adjacent pearls is not enough to determine if it is contained in the relevant region for the pearl string.

However, we ultimately chose this approach for our final voxel grabber and visualizer for a variety of reasons, which we describe in the "Results" section below.

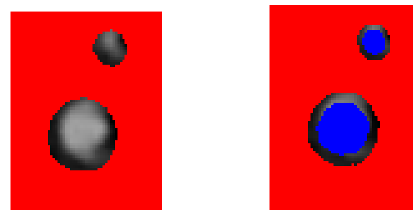

Fig. 9. (Left) A tubular structure extracted from a 3D volume. (Right) A tubular structure containing an interior radius defining a hollow region (the blue segment.) We do not extract the voxels contained within this region.

Extraction of Tubular Structures with Inner Radii

As mentioned in the introduction, we may have internal radii in the pearls which form a "crust" surrounding an empty region, and if a voxel falls inside this crust, it should not be marked for transmission. After implementing the final approach mentioned above to extract the voxels contained within a single tubular structure with no inner

radius, we expanded it to allow it to extract structures containing a hollow inner radius as well. Once we have determined that a voxel lies within the outer pearl string, we then perform the same tests described above using the inner pearl string. If the test fails, then we have a voxel which lies within the "crust" surrounding the empty region, and thus we send this voxel to the output destination. This approach is more computationally expensive, as it requires performing the test described above for a second set of pearls; yet, as seen in the result section below, it also reduce the total amount of voxels to be stored or transmitted.

Fig. 10. A 2D slice of the "bit mask" produced by rasterizing the pearl string using the GPU-based implementation. White pixels indicate the voxels which are marked as "sent," while black pixels indicate that the voxels are "unsent." Note the ridges between adjacent voxels produced by this approach, which are not present in the final voxel extraction tool using the 2D region-growing implementation.

Results

To evaluate the general efficiency of each of these approaches, we first implemented these algorithms in applications which simply performed a basic rasterization on the pearl string to produce a "bit mask" marking which voxels are and are not marked as to be "sent" (see Fig. 10) That is, these applications simply subdivided the pearl strings (using linear subdivision, rather than the 4-Point Subdivision used in the final application), and applied the algorithms as described above, marking which voxels are to be "sent" by the algorithm, but did not extract the intensity values of the voxels from an input volume file or reconstruct them into an image using the same algorithm. Based on our research and the results of these tests, we selected the third approach above to be fully implemented in our voxel extraction and image reconstruction tool.

We implemented these algorithms using several pearl strings of various sizes (both in terms of the number of pearls in the string, and the radius of each pearl), defined over a 512x512x512 3D volume consisting of a set of angiograms, and measured the results. The intensity values for the voxel in volume were stored as a 131 Mbytes RAW file (with 1 byte per voxel intensity value), which we used as input to our final voxel extraction algorithm below. All of the tests described below were performed on a laptop with 1GB RAM, an ATI Radeon X1200 graphics card, and a 1.8 GHz dual-core AMD Turion processor.

Table 1 on the following page gives the results of running these applications on these pearl strings in terms of the runtimes of the applications. We also report the total number of voxels marked as "sent" be each of these algorithms (Table 2). We note that these numbers are slightly different for each application, due to slight differences in their manners for selecting which voxels are within the relevant region. In particular, the GPU-based and 2D region-growing methods seem to consistently mark slightly more voxels than the 3D region-growing method. In the case of the former, this is most likely due to the fact that, for anti-aliasing purposes, when rasterizing the circles, the GPU will mark some pixels as falling "within" the circles even if their centers are not exactly within the radii of the circles. In the case of the latter, this is due to the fact that, unlike the other approaches, the 2D region growing method includes the area contained within the "ridges" between adjacent pearls as part of the tubular

structure. The 3D region-growing algorithm, however, is designed such that only voxels whose centers fall within the radius of at least one of the pearls in the string will be "sent," and thus of the 3 approaches, it will extract the fewest voxels.

Table 1: Comparison of the Runtime of the Voxel Marking Applications

Pearl String	# Pearls (w/ subdivision)	Runtime: GPU Implementation	Runtime: 3D Region Grower	Runtime: 2D Region Grower
aniography1.prl	33	2.793717 sec	0.253047	0.419349
angiography2.prl	175	13.742730 sec	1.329536	7.495711
angiography3.prl	111	3.438213 sec	0.150404	0.960480

Table 2: Comparison of the # of Voxels Marked by the Voxel Marking Applications

Pearl String	# Pearls (w/ subdivision)	# voxels marked: GPU Implementation	# voxels marked: 3D Region Grower	# voxels marked: 2D Region Grower
aniography1.prl	33	42555	41207	41776
angiography2.prl	175	205864	201078	209065
angiography3.prl	111	27436	26866	27980

Tables 1-2: The results of our voxel marking applications. Table 1: The amount of time requires to mark all of the voxels to be sent for each of the algorithms discussed above. Table 2: The total number of voxels marked by each of the algorithms. Different approaches mark slightly different numbers of voxels.

Based on the results of these applications, we chose to disregard our initial GPU-based approach due to its relatively slow speed, which we attribute to the large delays in reading the rendered pixels representing which voxels are to be sent from the GPU. From our research, it seems that copying data from the GPU's memory to the system's main RAM is a much slower operation than we had initially presumed, and leads to unacceptably long delays when executing our application. However, should there some manner of eliminating or avoiding these delays, we believe that our first approach above would merit further exploration.

The results above indicated that, the 3D region-growing implementation works rather quickly compared to the other approaches, and thus we initially chose it as the approach to be used in our final implementation of the voxel extraction and image reconstruction application. We initially assumed that its relatively quick speed in marking the voxels to be sent indicated that it would also perform relatively well when reading the intensity values of these voxels and storing them in an output file. However, while implementing and testing this application, we encountered several difficulties which were not present in the simple applications we created to get a general evaluation of their performance. When actually extracting the intensity values of the voxels from the input volume, rather than simply marking which voxels in the volume are to be sent, one must read the intensity values from the RAW data file used to store the volume on disk. However, as the 3D region growing algorithm places voxels from different areas of the input volume into the queue, one may have to read values from very distant sections of the RAW file one after another. Attempting to read the entire 131MByte file into main memory before applying the region-growing algorithm led to very long delays in the program, and, in combination with the 3-dimensional array of boolean values used by the algorithm, led to a large increase in the amount of memory used by the algorithm.

Table 3: Runtime of the Final Voxel Extraction Application (without interior radii)

Pearl String	# Pearls (w/ subdivision)	# voxels extracted	Runtime (seconds)
aniography1.prl	33	42422	0.354072
angiography2.prl	175	211059	7.202474
angiography3.prl	111	28264	0.978244

Table 4: Runtime of the Final Voxel Extraction Application (with interior radii)

Pearl String	# Pearls (w/ subdivision)	# voxels extracted	Runtime (seconds)
aniographyInner1.prl	33	21118	0.501022
angiographyInner2.prl	175	153079	14.791886
angiographyInner3.prl	111	21820	1.502557

Tables 3-4: The results of our final voxel extraction application. Table 3: The runtime of the final voxel extraction (the 2-dimensional region growing algorithm) used in VoxelGrabber, which marks the voxels to be sent , reads their intensity values from a RAW data file, and writes the intensity values to an output data file. The application also verifies the success of the algorithm by reconstructing the relevant section of the volume by reading the extracted voxel values from the output file in the correct order, but we do not measure the time required for this step. Table 4: The results when running the application on several pearl strings with interior radii defining a hollow region.

We attempted to resolve these issues by accessing the relevant voxels in the RAW file using a file descriptor which read the "marked" voxels from the file on disk one at a time, moving the file descriptor to different positions in the input file for each voxel to be extracted. However, this also led to long delays when testing the program, most likely due to the overhead of having to read each extracted voxel from the disk separately. Given that the algorithm expands the region of extracted voxels across the volume in 3 dimensions simultaneously, requiring that we read voxels sequentially from very different areas of the input volume, we see no clear and simple way around these delays which does not involve loading the entire volume into memory, or making many reads from the disk.

The 2-dimensional region-growing algorithm, on the other hand, processes the volume using only 1 slice at a time, meaning that the values from only one slice of the input volume are accessed by the algorithm at any given time. As a result, we may simply read the values from the current slice into a 512x512-byte region of memory, extract the relevant voxels, and then clear this memory region and perform the same operations for the next slice. We believe that the greater memory coherency of this algorithm makes it more practical for than the previous approach, despite the greater computational complexity of the algorithm we use for extracting voxels. (In retrospect, given the rate at which CPU speeds are increasing compared to memory bandwidth in conventional computers, it is logical that the memory requirements of the algorithm would become the dominating factor in the determining its performance level.) Initial testing when implementing the voxel extraction and reconstruction tool using this approach indeed indicated that this approach resulted in much faster extraction of voxels from the RAW file storing the input volume than the previous approach.

As a result, and on the advice of Professor Rossignac, we ultimately chose the 3rd extractor described above to be used in the final voxel extraction and image reconstruction program. This program performs the same steps as the applications created for the other approaches; however, rather than simply marking the voxel as

within the relevant region, this application actually extracts the intensity value for this voxel from the correct position in the input volume and writes them to an output file in the order in which they are "touched" by the region-growing algorithm. Upon completing these steps, the algorithm then reconstructs the relevant portion of the image one slice at a time, which it then displays to the user to demonstrate the effectiveness and correctness of this approach in reconstructing the relevant portion of the input volume. We also enhanced this application such that it also allows for the extraction of tubular structures containing an interior radius defining a hollow region, as described above. Please see Tables 3 and 4 above for the results of executing this application on several pearl strings.

As there is 1 byte for the intensity value for each voxel stored in the RAW file, the number of voxels grabbed in the final column is also the size of the output file created by this grabber. Cleary, this process of extracting the relevant voxels from a 3D volume can lead to drastic reductions in the size of the data file required to store this region of the image, if the pearl string defining the relevant region of the image only covers a small portion of the voxels in the 3D image. The size of the output files for these pearl strings range from roughly 21 to 211 KB, compared to the original 131MB RAW file. Of course, se must also store the text files defining the input pearl strings, but these text files are also relatively small (). It would also be possible to represent these files in a much smaller, non-text format.

Running the algorithm with pearl strings containing an inner radius, we see that the total amount of voxels which are extracted from the volume decreases, while the time required for extracting these voxels increases slightly. This is logical, as we are not extracting the voxels contained in the inner radius (and thus extract fewer voxels), and we must perform now 2 tests for each voxel in order to determine whether it lies outside of the inner crust and inside the outer crust (and thus we must perform more computations to store these voxels.) Whether the tradeoff between performance in the extraction application and the size of the data file to be stored and/or transmitted is worth the addition of this feature will most likely depend upon the applications for which this tool is used. However, we intend to study other approaches to obtaining the voxels contained in this crust in order to determine if there is some which is more efficient manner of obtaining these voxels than simply performing the same tests for both pearl strings.

We believe that these tests demonstrate the effectiveness of this approach to extracting tubular structures from image volumes. Given the difficulties described above which we encountered when implementing the other approaches, we believe that, despite their theoretical benefits, this 2-dimensional region growing algorithm is the best of the techniques given here for implementing this voxel extraction technique from a practical perspective. However, we also believe that this approach is open to several possible optimizations and extensions, which we briefly describe below ("In Future Work").

In the following section, we briefly describe a possible approach for compressing and encoding the data extracted from the 3-dminesional volume, in order to further reduce the amount of data which is required to extract the voxels from the volume, which we intend to implement in VoxGrabber in the future.

Compression

Though the methods implemented above reduce the cost of sending or storing the relevant 3D volumetric data by a large amount, we may further reduce the amount of data needed by compressing the data using an appropriate predictive compression and encoding scheme. One scheme in particular which we feel would be particularly useful in reducing the

transmission size is the Spectral Predictor [4][5][6], which can be used to predict a point in a 3D Grid, when the unknown point is arbitrarily located in the grid.

In other words, suppose that we seek to predict the value of some point in the 3D grid based on the values of its neighbors, but we do not know the values of all of its neighbors (as some of them may remain "unsent.") For different points, we may have different combinations of "known" and "unknown" neighbors which we wish to use to accurately predict the value of the current point. This predictor predicts the value of the unknown point by assigning weights to each of the known neighbors; a different weight is assigned to each of the know neighbors, based on which of the neighbors are known and which are unknown. These weights may be selected from a look-up table, which is indexed using the combination of which neighbors are known and which are unknown. Here we provide a brief description of this prediction and the encoding method with which it may be used to reduce the storage size of the data output by the voxel extraction tool.

Before looking into the details of Spectral predictor, we will briefly describe the L^2 and R predictors upon which it is based.

L^2 Predictor

The L^2 Predictor, called the *Bi-Lorenzian* predictor is the extension of L^1 predictor from 1 dimension to 2D [4][5][6]. If f(x) be the function which is regularly sampled in one dimension at $\{\ldots f_{i-1}, f_i, f_{i+1}, \ldots\}$, the L^1 predictor finds the next point by setting the approximate differential of the function f(x) at f_i to zero. Here the approximate differential can be written as

$$\Delta^x_i = f_i - f_{i-1}$$

Setting Δ^x_i to zero, we get

$$L^1 = f_i = f_{i-1}$$

This Lorenzo predictor can be easily extended to two dimensions, by taking the composition of derivatives.

$$\Delta^{xy}_{i,j} = \Delta^x_{i,j} - \Delta^x_{i,j-1}$$

Setting $\Delta^{xy}_{i,j}$ to zero,

$$L^1_{i,j} = f_{i,j-1} + f_{i-1,j} - f_{i-1,j-1}$$

With this setting, Lorenzo's predictor can predict linear or bi-linear polynomials without highest order term xy.

Extending this in higher order polynomials, we get the L^2 predictor as

$$L^2_{i+1,j+1} = 2f_{i,j-1} + 2f_{i-1,j} + 2f_{i,j+1} + 2f_{i+1,j} - 4f_{i,j} - f_{i-1,j-1} - f_{i+1,j-1} - f_{i-1,j+1} - f_{i+1,j+1}$$

Interpolating Predictor, R

While the L^2 predictor is for predicting the corner points in a 3 x 3 grid of samples, the interpolating predictor is used to predict the sample point at the center of the grid. This is called the radial interpolating predictor, as its weights are dependent on the distance to neighborhood samples.

$$R_{i,j} = \tfrac{1}{4}(2f_{i,j-1} + 2f_{i-1,j} + 2f_{i,j+1} + 2f_{i+1,j} - f_{i-1,j-1} - f_{i+1,j-1} - f_{i-1,j+1} - f_{i+1,j+1})$$

Spectral Predictor

The spectral predictor works by generalizing the L^2 and R predictors for all the possible configurations. We can set the distance weights depending on the location of the point on the sample grid which is to be predicted. Figure 11 gives examples of the coefficient used for the aforementioned predictors.

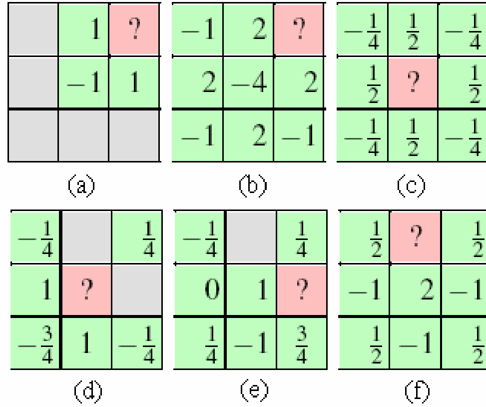

Fig. 11: Examples of the coefficients used for the ***(a)*** *Linear (L^1) Predictor* ***(b)*** *Bi-Lorenzian (L^2) Predictor* ***(c)*** *Radial (R) Predictor* ***(d)-(f)*** *Full spectral Predictors (Sources: [4][5][6])*

Encoding/ Decoding

Once we have the predicted values, we may compress them using a Huffman Encoding in order to achieve a much higher level of data reduction before storing and/or transmitting the voxel intensity values. In Huffman encoding, we assign a probability to each of the symbols to compress, depending on the number of times it occurs in the data to compress. The smallest number of bits is allocated to the symbol having the highest probability, and vice versa, to achieve maximum compression. To generate the Huffman encoding, the symbols are stored in a binary tree depending on their probabilities, such that the ones with higher probability are nearer to the root of the tree and the ones with less value of probability farther. The Huffman code for a particular symbol is given by the path traversed from root to that symbol, which is generally taken as 0 in the left and 1 in the right. Figure 12 below shows one example of this encoding scheme.

In the given example, the symbols will be encoded as

A ≡ 0 (1 bit)
B ≡ 10 (2 bits)
C ≡ 110 (3 bits)
D ≡ 111 (3 bits)

Total number of bits used per symbol:

n = 0.5 * 1 + 0.25 * 2 + 0.125 * 3 + 0.125 * 3 = 1.75 bits per symbol

Applications of Compression and Encoding for VoxGrabber

As mentioned above, our tool performs a 2-dimensional region growing algorithm for each slice of the input volume, in order to form a structure which fills some contiguous region of the image. Thus, it is clear that, as we iterate through the slices of the volume containing a particular pearl, many of the voxels placed in the queue will have “neighbors” from the previous slice below them which have already been extracted by the algorithm and thus may be used to predict the value of the current voxel. For example, to predict the voxel at (x,y,z), we may use the values as (x-1,y,z-1), (x,y,z-1), (x+1,y,z-1), etc., in predicting the voxel’s intensity value. In addition, as each voxel which is to be “sent” places its untouched neighbors in the queue, each voxel (except for the seed voxels) will have at least one and possible several neighbors on the current slice which may be used in predicting its value. Due to the manner in which we perform this region growing, each voxel may have a different combinations of “sent” and “unsent” neighbors which may be used by the receiver to predict its value. Thus, the spectral predictor, which gives different weights to the existing neighboring vertices depending on which ones are already sent, would be an ideal choice for incorporating prediction into this voxel extraction tool.
Using Huffman encoding to send the correction symbols for the predicted intensity values as described above would require sending some digital representation of the symbol tree to the storage file or viewer as well. However, with 1 byte per symbol, only 256 possible intensity values are possible, and thus the total number of correction symbols in the tree would be quite small compared to the size of a

sufficiently large input volume. For tubular structures defining regions containing voxel values which are fairly similar (i.e., regions which consist mainly of voxels of the same or similar intensities), using some effective form of prediction, such as the Spectral Predictor, to predict these intensities, most of the predicted values will only require some small correction or no correction at all.

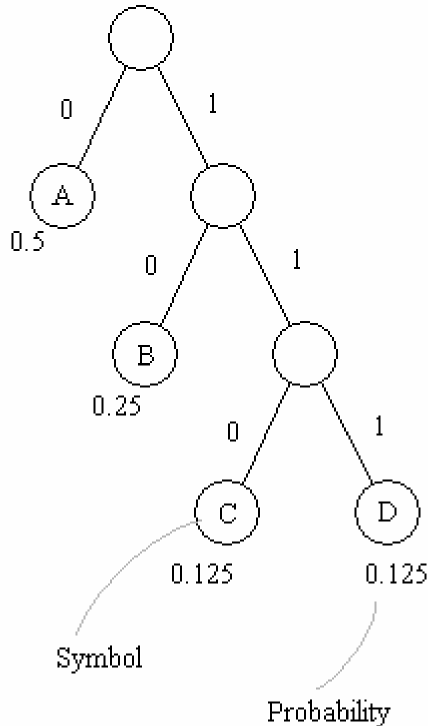

Fig 12: An example of Huffman Encoding

By assigning the symbols for these correction values to the smallest combination of bits in the Huffman tree, we may transmit the corrections for the majority of the predictions using only a very small number of bits, thus reducing the amount of data used to store the voxels extracted by VoxGrabber by an even greater margin.

Future Work

Though we have successfully implemented a voxel extraction application which quickly extracts the voxels contained within the desired tubular structure from a 3D volume, thus reducing the amount of data to be stored or transmitted by a large factor, there are several optimizations and extensions to our approach which we intend to explore.

Firstly, despite settling upon the 3rd approach described above for our final application, we believe that with further effort and research, we could improve upon its performance by combining this approach with some of the more advantageous aspects of the other approaches. For example, we may explore methods of using the quick sphere rasterization calculations described above (or some similar approach) in order to reduce the computational complexity of checking to see if a voxel lies within the relevant region. We also intend to explore methods for extending this application to support pearl strings with bifurcations, in order to allow this application to be used on when extracting branching tubular structures, such as portions of the circulatory or skeletal system, from a 3D volume. Finally, we will explore methods for integrating some appropriate form of data compression and encoding, such as the Spectral Predictor and Huffman Encoding mentioned above, in order to reduce the amount of data required to store the extracted voxels by an even greater margin.

Conclusions

We have explored and evaluated three different tube rasterization techniques for efficiently extracting the voxels contained within a tubular structure from a 3D medical image. Based on the results of our evaluation, we selected the third approach described above to be implemented in our final voxel extraction tool, primarily due to the low RAM requirements, sufficient speed and smooth rasterization of the pearl string. By discarding the undesired portions of the image, we drastically reduced the size of the data file required to store the relevant regions, and demonstrated that these regions may successfully be reconstructed into the original image of the tubular structure. If combined with a suitable prediction and encoding scheme, such as the ones discussed in this paper, we believe that a much higher

level of compression can be achieved, and intend to explore approaches to performing this compression in the future.

References

[1] Jarek Rossignac**:** "Solid and Physical Modeling"
[2] J. Rossignac, B. Whited, G. Slabaug, T. Fang, G. Unal: "Pearling: Medical Image Segmentation with Pearl Strings", to appear, The First International Workshop on on Image Mining Theory and Applications (IMTA), 2008.
[3] Huffman, D. A., "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., pp. 1098–1102, September 1952.
[4] L. Ibarria, P. Lindstron, J. Rossignac: "Spectral Interpolation on 3x3 Stencils for Prediction and Compression", Journal of Computers, 2(8), 2007
[5] L. Ibarria, P. Lindstrom, and J. Rossignac, "Spectral predictors," Data Compression Conference, pp. 163–172, March 2007.
[6] L. Ibarria,, P. Lindstrom, J. Rossignac, A. Szymczak: "Out-of-core compression and decompression of large n-dimensional scalar fields", Computer Graphics Forum, Volume 22, Number 3, September 2003
[7] B. Whited, J. Rossignac, G. Slabaugh, T. Fang, and G. Unal: "Pearling: Stroke Segmentation with Crusted Pearl Strings", to appear, The First International Workshop on Image Mining Theory and Applications (IMTA), 2008.