ALIGNING LARGE-SCALE MEDICAL AND BIOLOGICAL DATA SETS: EXPLORING A MONKEY BRAIN

DMITRY SHULGA, JOERG MEYER
Mississippi State University
NSF Engineering Research Center, 2 Research Blvd., Starkville, MS 39759, USA
schultz@gyral.com, jmeyer@erc.msstate.edu

ABSTRACT

This project addresses the issue of developing interactive rendering methods for datasets which cannot be stored on a single hard drive or in main memory anymore. Our dataset is a set of 1400 slices (single cross-sections) of a monkey brain, which has been sliced more than 15 years ago at the Center for Neuroscience at UC Davis, and recently has been scanned at a very high resolution (more than 10MB per image in compressed format). The enormous resolution allows us to zoom from a global view down to the cell level, all in one image. This exciting range of rendering options requires scalable, multiresolution rendering techniques. The challenges we encounter with this data set is an extreme misalignment of the slices due to manual placement onto glass object carriers and manual insertion in the film scanner. We present a semi-automated method which compensates for most of these artifacts and identifies those slices that cannot be handled and aligned automatically. The algorithm reduces the number of slices that need to be treated manually enormously.

KEYWORDS

Registration, alignment, 3-D reconstruction, large-scale visualization.

1 INTRODUCTION

Large-scale biomedical data sets, such as CT, MRI or PET scans, cryo-sections, confocal laser-scanning microscopy, and other automated imaging techniques, provide series of 2-D cross-sections, which are usually perfectly aligned. If the slicing is done manually, serious misalignment might be encountered, which prohibits a good 3-D reconstruction of such data sets. However, the quality and the resolution of those data sets makes it desirable to use those sliced brains, which have been cut and preserved more than a decade ago, instead of slicing new ones.

Sets of more than 1400 slices (single cross-sections) of a Rhesus monkey brain, which have been sliced at the Center for Neuroscience at UC Davis, and have now been scanned at a very high resolution using a 35mm film scanner, are now available. They have been scanned in order to archive them electronically and to preserve them for future studies. Each slice comprises of more than 10MB of image data in compressed JPEG format. Our test dataset reveals detailed information about the structure of the brain down to the cell level (Figure 1).

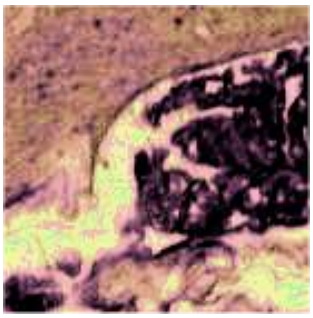

Figure 1: Cell level of a brain (detail)

Interactive visualization of large-scale datasets requires advanced techniques in image processing, hierarchical data management, and data reduction. Our goal is the development of interactive visualization techniques for large-scale datasets based on hierarchical representations and immersive visualization environments, such as the Virtual Workbench (stereoscopic display device with interaction facilities), or the CAVE™. We want to explore these 3-D datasets in an intuitive way at high resolution and at unrivaled precision. The main problem is the alignment of the slices. This paper addresses some of the challenges we encountered.

2 SCANNED IMAGE DATA

The slices have been scanned at a resolution of 3000dpi, which corresponds to a distance of 0.08 μm between pixels (Figure 2). This enormous amount of detail makes it possible to zoom down to the cell level. A purple dye has been used to mark the cell nuclei, which are now visible as darker spots in the image (Figure 1).

Fortunately, two metal pins were pushed through the brain before slicing. These pin holes can now be used for registration and for initial alignment. Since the three pieces were glued on the plate separately, they need to be registered separately. Advanced methods, such as contour finding, morphing and warping, will be discussed later [7,8]. First, we are going to focus on the registration marks.

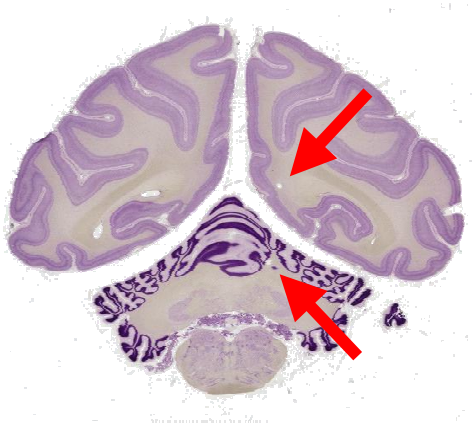

Figure 2: Rhesus monkey brain

3 IMPLEMENTATION

3.1 SCALING

The average size of the slices is 4500x3000 pixels. In order to find the registration marks (pin holes), we do not need the full resolution. We can speed up the algorithm by reducing the size of a single slice by a factor of eight in each dimension. This gives us an image which can be displayed on a regular computer screen. Thus, it is necessary to decimate the data set by interpolating a set of points and replacing them by a suitable representative. This way, features can be preserved which might otherwise get lost. Basically, we are averaging over a square region to interpolate the data in two directions (Figure 3).

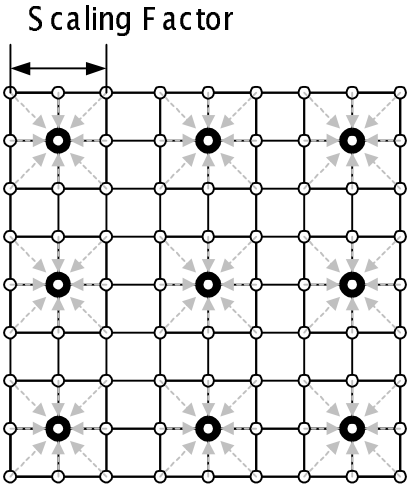

Figure 3: Scaling procedure

The main goals of the scaling algorithm are competitive. There is a trade-off between performance and precision of detection of the pin hole location in the full-scale image. To meet the first goal is fairly easy. The main problem that causes latency is I/O. Therefore we try to avoid all intermediate functional layers for accessing the files and make direct use of standard C++ classes like `fstream` and `iostream`. They offer unrivaled performance and immediate access to a block record on the hard drive.

The main steps of this part of the algorithm are the following:

1. Open file as `ifstream`:
   ```
   ifstream f(file_name)
   ```
2. Buffer it with *istream*:
3. Read data block (block height is the number of rows to interpolate):
   ```
   f.read(buf, size);
   ```
4. Interpolate colors;
5. Save to output file:
   ```
   of.write(obuf, osize);
   ```
6. Continue with next block.

The second goal is to create a reasonable, down-scaled representation of a slice. Basically, we use averaging of colors of adjacent pixels (Figure 3). In most cases it is sufficient to consider gray levels only rather than actual colors. The number of points in a row or column for averaging is the square of the reduction factor.

3.2 ALIGNING

In order to create a high-quality 3-D model of the Rhesus monkey brain, the individual parts need to be aligned. A first cue is given by the position of the registration marks. This section describes how to identify those marks within a larger image.

The algorithm implements the following steps:

- Find Bounding Box – determine the rectangular area where the data set is actually located within a particular slice;
- Find Mark Center – determine the center of a given registration mark;
- Locate Marks - determine if there is a registration mark within a limited range.

3.3 IDENTIFYING A BOUNDING BOX

Since all slices are in a slightly different position on the glass plate and in the scanner area, it is necessary to find the object within a scanned image. The first step to realign the slices is to find a bounding box, i.e., a rectangular area defining the boundaries of the object (Figure 4).

The main principle of determining the edges of a bounding box is to search the image from all four sides until a significant change in the color value is encountered. We use a general, differential approach, which detects both transitions from positive to negative and from negative to positive.

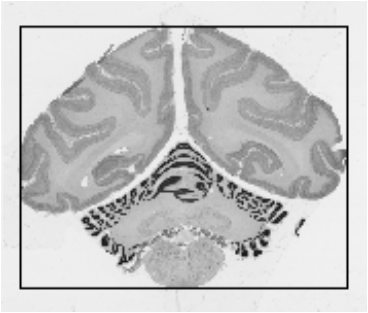

Figure 4: Bounding box

Mathematically, this corresponds to the derivative of the color value function:

$$\frac{di}{dx} = \frac{i_2 - i_1}{x_2 - x_1}$$

Since we have a discrete coordinate system and unit coordinates, we can write it as:

$$\frac{di}{dx} = i_2 - i_1$$

Scanning is similar for all sides (with different search directions). Let us consider some techniques that helped us to develop the final version of the algorithm. It should be mentioned that this method works both for images with light background (given example), and also for images which typically have a dark background (CT or MRI scans).

A brute-force method would be to compare neighboring pixel colors. If the difference is significant, we can assume that an edge has been found. This would work fairly well on images with high contrast, low noise, and well-defined contours. All three criteria are not met by our particular data set. The pixel search algorithm would terminate too early and probably falsely detect some noise and accept it as the start pixel for an object within a slice.

So the next step we need to do is to eliminate the influence of noise pixels. This can be done by blurring a picture in the direction of the search algorithm. There is a slight difference between finding a bottom or top edge and finding a left or right edge. In the first case, pixels must be blurred in horizontal direction (Figure 5), whereas for the second case, pixels must be blurred in vertical direction (Figure 6).

We learned that blurring helps to smooth out noise pixels very efficiently, so that we can apply our differential pixel search method to identify the object boundaries.

The following section describes the blurring algorithm. The method is similar to motion blurring. First, the number *n* of points which are supposed to be averaged, and two buffers, one for the original image (*obuf*) and one for the blurred image (*bbuf*), must be introduced. Every row or column is blurred in the same manner.

Figure 5: Horizontal blurring

Figure 6: Vertical blurring

The algorithm accumulates the first *n* color values and stores them in *ai*. For each $i > n$, the following operations are performed (note that the array index begins at 0):

1. Let *n* = number of points to be averaged.
2. Average pixel values and store the value in a *bbuf* buffer cell: $bbuf_{i-n} = avg(ai)$.
3. Subtract value *(i-n)* of original buffer *obuf* from accumulator ai: $ai = ai - obuf_{i-n}$.
4. If end of line is not reached ($i < m$, where *m* is the line length), then add value *i* of original buffer *obuf* to accumulator ai: $ai = ai + obuf_i$.
5. Otherwise, decrement *n*.
6. If $n = 0$, go to the next line; otherwise repeat from step 2.

The algorithm above blurs images quite efficiently. The question that remains is: What do we consider a *significant* change in color values? Generally, this difference may be expressed in absolute color values or in a percentage ratio. Measuring absolute values can be problematic as the range of color values varies on different pictures. The same holds for percentage values, because the deviation from the average value also varies in a wide range. The solution is to normalize the histogram and find minimum and maximum color values, and then use a fixed threshold.

Again we need to search the entire picture, and, for similar reasons, blur the image while searching to avoid

running into local minima or maxima. Figure 7 shows how a picture with a black spot considerably changes after blurring. Instead of detecting a single dot as an a extremum or a boundary condition, the spot almost disappears in the blurred image. This effect is desired. Only objects of a reasonable minimum size are taken into account for the boundary search.

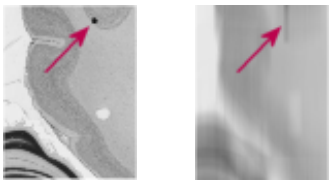

Figure 7: Dark spot (artifact) before and after vertical blurring (slice no. 504/1400)

After finding the extreme color values, the interpolation of the current color value is straightforward. Basically, the lowest possible value is 0%, whereas the highest is 100% (Figure 8). The actual values must be mapped to this scale.

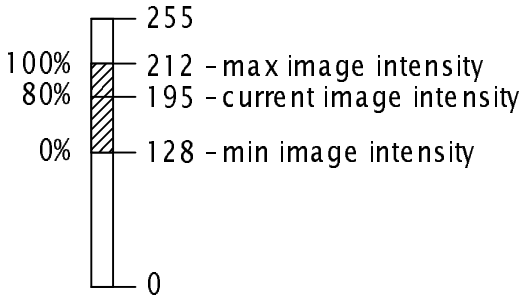

Figure 8: Color value interpolation

The combination of image normalization and filtering of local extrema allows us now to search for the boundaries of the bounding box. But here we encounter another problem. What if the color values between adjacent pixels change only gradually? Then our threshold method would never apply, even if there are large overall variations of color values. The following example will illustrate this problem.

The first attempt was to check if the difference *d* between two adjacent pixels was greater than a certain threshold:

$$d_i = i_{i-1} - i_i$$

Even if this approach looks very straightforward, it does not work. The reason is that either the original data set or the blurred image exposes only gradual changes in color values when compared to local neighbors (Figure 9).

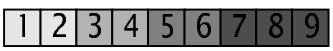

Figure 9: Color value gradient

As can be seen, there is no significant change between adjacent pixel colors with respect to the total color value range. Thus, all these cases will be ignored, while in reality this sequence of pixels might represent the object boundary.

This observation calls for an improved method. At first sight, it might seem to be sufficient to compare not direct neighbors, but pixels within a certain distance, e.g., for pixel 1 it would be the 3rd, the 4th, and so on. Unfortunately, it does not work either, as in real data the difference between such pixels can be very small. So even comparing pixels within a certain distance would not give a correct result (Figure 10).

Figure 10: Real gradient

However, a careful observation shows that despite the small difference between the 1st and the 9th pixel, there is a trend towards darker pixels throughout the sequence. Thus, to determine an edge of an object, an accumulated difference can be used (Figure 11).

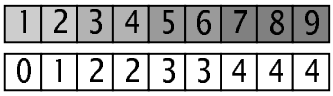

Figure 11: Accumulated difference

The main search algorithm works in the following order:

1. Starting from the second pixel in each line, check if there is any difference in the color values of the *n*-th and (*n*-1)-th pixels:
$$di_n = i_n - i_{n-1} > threshold\,?$$
2. If yes, then check the sign of the difference; if the sign is the same as previous, add color value to accumulator; otherwise increment *n*, reset switches, and return to step 1:
$$ai = \begin{cases} 0, sign(di_n) \neq sign(di_{n-1}); \\ ai + i_n, sign(di_n) = sign(di_{n-1}). \end{cases}$$
3. Check if accumulated color value is bigger than allowed deviations; if yes, then an edge is found; otherwise if the number of accumulations equals the maximum accumulation depth then increment *n*, reset switches, and return to step 1 (see 2).
4. Increment *n*; if the end of the line is reached, go to the next line.

Finally, it should be mentioned that finding the vertical bounding box edges of the object differs from finding the horizontal ones by the fact that it is not necessary to use the despeckle filter on the whole picture again; it is sufficient to blur the current line that is under processing. The first pass (minimum and maximum search with filter) can actually be used for one direction of the bounding edge finder. For the other direction, the orientation of the filter is perpendicular. There is no need to search for minimum and maximum color values for both

orientations, as in practice the limits found from the horizontally blurred image are not considerably different from the ones found in the vertically blurred image. For large images this fact noticeably increases performance.

3.4 FINDING THE CENTER OF A REGISTRATION MARK

As mentioned earlier, the data set features two registration pin holes, which can be used to align the slices. Unfortunately, each slice consist of several pieces that have been positioned manually. Therefore the distance and orientation among the pieces varies from slice to slice. Two registration holes in two separate pieces are not sufficient to register the entire data set. However, they provide a good starting point. These two points will become the fix points (Pivot points) for the registration. Other features, such as contour polygons, must be taken into account in order to do a proper alignment. Pattern matching algorithms, which can be used to identify similar sets of contour points in two adjacent slices or even across several slices have been developed before [9]. This paper focuses on the Pivot points and describes a method to localize those points by making use of spatial coherence and similarity. Image similarity is only given if the object is located in the same range or window. This is guaranteed by the bounding box method described in the previous section.

The algorithm starts from a slice somewhere near the center where both registration marks are present. The user then selects the two pin holes with the mouse. This step is not automated, because the data set exposes many similar features, which could be mistaken for pin holes. It is not very time-consuming for the user to select those two points, but we want to avoid that the user needs to go through all 1400 slices to select those points. The algorithm does this automatically by searching a small neigborhood of the selected pin hole in one of the adjacent slices. The algorithm searches in both directions until it reaches the last slice on each side.

The search method is based on a polygon flood fill algorithm with 8-neighborhood adjacency [2]. We suppose that a spot is a convex discrete polygon. The algorithm fills the entire area of the pin hole with the same color as the pixel that was selected and counts how many pixels were filled.

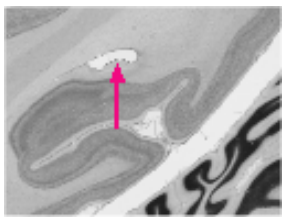

Figure 12: Concave hole

It returns the barycenter of those filled pixel locations, which corresponds to the center of the pin hole (provided it was convex, which we can safely assume). In case of a concave hole, the barycenter may be located outside the actual spot (Figure 12). Obviously, it is very difficult for the user to put the seed point exactly in the center of the pin hole. That is why this algorithm is very useful for identifying this point (Figure 13).

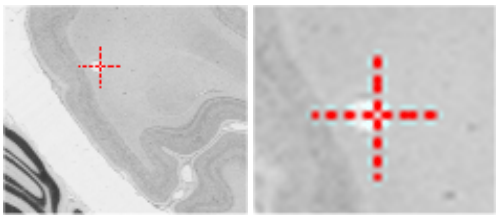

Figure 13: Registration mark (user-marked, close-up)

The flood fill algorithm can be easily implemented as a recursive function:

1. Start from seed point.
2. If current pixel is of the same or similar color as the starting point, then fill it; otherwise go to step 5.
3. Increment the number of processed pixels and add coordinates to accumulators.
4. For each of the 8 neighbor pixels, recursively call this procedure.
5. Return barycenter (averaged coordinates, sum of accumulated coordinates divided by number of processed points).

3.5 LOCATING REGISTRATION MARKS IN ADJACENT SLICES

After finding the center of a given registration mark, it is possible to use these coordinates to locate the same mark on another slice. The search method is based on several aspects. First, the color of a pixel should match the color of the Pivot point in the previous slice. Second, the size, i.e., the number of pixels within the pin hole, should roughly match.

Basically, finding a corresponding mark in an adjacent slice works pretty much the same as finding the center of the pin hole in the reference image:

1. Start from the Pivot point.
2. If the color of a pixel roughly matches the color of the mark in the previous slice, then go to step 3, else go to step 4.
3. If size of the mark roughly matches the size of the mark in the previous slice, then goto step 5, else goto step 4.
4. For each of the 8 neighbor pixels, recursively call this procedure.
5. Call previous algorithm (section 3.4) to determine center.

4 RESULTS

The methods described in the previous sections have been applied to a large volumetric data set of a Rhesus monkey brain, which consists of 1400 slices, each about 4500 x 3000 pixels (size varies slightly). The total size of the data set is about 76 GB. We were looking for automated alignment methods to reduce the number of slices that need to be loaded into an editor and manipulated by hand. The algorithm creates a log file which stores the pixel positions in world coordinates for each registration mark in the slice. If no registration mark was found within a certain search radius, the slice is rejected and marked for manual handling. The bounding box method helped us to avoid tedious cutting operations, and the pin hole method provided us with the Pivot points for piece alignment. We observed that the window size for the despeckle filter makes a big difference on the quality and precision of the bounding box. As the main goal of blurring is to reduce inconsistencies in images, it is important to find an optimum window size.

Overlapping while blurring also has a big impact. A larger overlap gives a better smoothing effect but it also creates a fuzzier gradient that may lead to failure in finding the boundary edges. On the other hand, reducing overlap sometimes causes features to be overlooked. Figure 14 shows a case where averaging results in losing edge information.

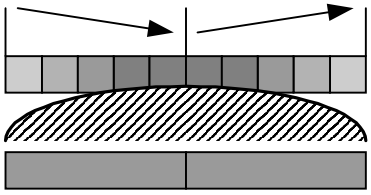

Figure 14: Non-overlapping blurring – original and average

5 CONCLUSIONS AND FUTURE WORK

We have discussed a flexible method for the detection of registration marks in large-scale biomedical data sets. This method will enable biologists and physicians to make use of thousands of objects that have been sliced in the past and are now scheduled for electronic archiving in order to preserve them for future generations before they deteriorate. High-resolution scanning technology and high-performance storage systems will make those scans available to a much wider audience. In order to understand the three-dimensional structure, it is desirable to be able to reconstruct the original volume from the slices. Our algorithm provides some robust techniques for windowing, object detection, and registration of the slices. Future work includes the addition of feature detection algorithms, which will overcome problems such as some uncertainty as how to match polygons with different numbers of vertices, different lengths of edges, etc. Some of these problems have been addressed in a related paper [9].

Our algorithm is able to find registration marks in an image series with high certainty. These marks will be the fix points (Pivot points) of a morphing or warping algorithm. Each object will be triangulated, and we can use these points as a common point shared by all triangles inside the object.

The current algorithm can be easily extended to detect other unique features, which are similar in adjacent slices, even if they are only dominant in a local domain. This will provide additional registration points, which will lead to a better 3-D reconstruction.

ACKNOWLEDGMENTS

We thank Edward Jones, Fred Gorin and Jim Stone at the Neuroscience Center (UC Davis) for providing the sample data set. We also thank Bernd Hamann (CIPIC, UC Davis) for valuable research opportunities. The project was a joint effort between the NSF Engineering Research Center at Mississippi State University, the Center for Image Processing and Integrated Computing (CIPIC) and the Neuroscience Center at UC Davis. This project was funded in part by the National Partnership for Advanced Computational Infrastructure (NPACI) under award no. 10195430 00120410.

REFERENCES

[1] Meyer, Joerg, 2000. LSV - Large Scale Visualization http://www.cs.msstate.edu/~jmeyer/lsv.html.

[2] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. 1996. *Computer Graphics*: *principles and practice*. Glenview, IL, Addison-Wesley Publishing Company Inc.

[3] OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. 2000. *OpenGL Programming Guide: the official guide to learning OpenGL , version 1.2*. Glenview, IL, Addison-Wesley Publishing Company Inc.

[4] OpenGL Architecture Review Board, Dave Shreiner, eds. 2001. *OpenGL Reference Manual: the official reference document to OpenGL, version 1.2*. Glenview, IL, Addison-Wesley Publishing Company Inc.

[5] Bailey, M., "Interacting with Direct Volume Rendering", IEEE Computer Graphics and Applications, Vol. 21, Issue 1, pp. 10-12, February 2001.

[6] Stroustrup, Bjarne. 1997. *The C++ Programming Language*. Glenview, IL, Addison-Wesley Publishing Company Inc.

[7] Tan, C.L., and S. K. K. Loh, Efficient Edge Detection Using Hierarchical Structures, Pattern Recognition, Vol. 26, No. 1, pp. 127-135, 1993.

[8] Schiemann, T., M. Bomans, U. Tiede, K. H. Höhne, Interactive 3D-Segmentation Visualization in Biomedical Computing, Vol. 1808, pp. 376-383, 1992.

[9] Fries, Karsten, Jörg Meyer, Hans Hagen, and Bernd Lindemann, "Analysis of Biomedical Image Corrrespondence: Matching 3-Dimensional Point Sets," Proc. of Scientific Visualization 2000, Schloß Dagstuhl, Germany, May 22 - 26, 2000.