## Project process

Latest version of Project process: https://github.com/andics/Big-Bang-Files.git

Latest version of project source code: https://github.com/andics/Cranio-analysis---Image-Extractor

For the cross-sectional images extraction, I’ve used MATLAB R2018a.

To do any sort of reading or manipulating on the skull dataset, I first had to load the image volume in the Matlab workspace. The workspace uses the RAM memory of the PC to store the loaded variables, in this case the loaded image volume. The first problem encountered was that the RAM memory required to completely load a typical skull dataset was about 50 GB. My PC has only about 14 GB of usable RAM memory.

What I figured out as a solution to this problem was to load a downscaled version of the volume instead of a full sized one. For example, every image from the dataset, with dimensions Width x Height, I resized by a scale factor of n (0 < n <= 1, where n is a variable depending on user preferences), so that the new image dimensions are (Width x n) x (Height x n). I also took the scale factor in consideration with the number of images in the dataset. Instead of loading every single image, I chose to load every 1/n th image. This way, the dimensions ratio of the produced downscaled 3d model remain consistent with the full-sized version. Typical dimensions of a fully sized skull volume are 2190 x 1840 x 2130. The first two dimensions represent the size of each image, and the third is the number of images in the dataset. Displaying such large volumes as graphical object on the screen can be a computationally heavy task, so this downscaling improved the smoothness and significantly reduced any lagging while working with the 3d model.

After having loaded the volume of images, I make a copy of the same volume and run it trough the binarizeVolume() function. This function binarizes and fills the holes in every image, result of which can be seen on img\_binarized.jpg. The processed volume is then used to generate the 3d model in the generateVolume() function. The advantage of processing the whole volume like this, is that unnecessary details are removed from each image. In this case, the only thing we’re interested in seeing on the model, is the very outside surface of the skull. After processing, this very outside surface on an image, is the only preserved detail, as can be seen in the mentioned above photo. This suits the needs of the project perfectly and significantly reduces the time taken to generate the 3d model, as the used volume is very simplified after processing.

Having generated the 3d model of the skull, the program relies on the user to define the places on the surface trough which a suture passes. An image with such points defined is suture\_points.jpg. The next task is to generate the path that the suture follows from the given defining points.

Each point on the surface, on the very low level, is a part of a triangular mesh. The question is, which path of connected points on this mesh do we choose to get from the start to the end point. Now this is a question with multiple solutions. Ultimately, this triangular mesh can be taught of as a weighted graph with the weights corresponding to distance between connected vertices. A number of graph search algorithms can be used to find the smallest cost path between two points. Most algorithms like Djikstra or the A\* search algorithm may need to go through a big part of the graph before finding an optimal solution. Considering the fact that the number of points on the triangulated mesh of the skull is in the millions range, this process can be computationally and time expensive. For this reason, a computationally cheaper, geometric approach has been used instead.

Let’s first look at the simplest case with only three defining suture points. A line is draw between the end point and the start point. Then, the foot of the normal to the drawn line from the middle point is found. Image suture\_3points.jpg represents the described above graphically. Let’s call the vector connecting the foot of the normal and the middle point our “compare vector”. Starting the path search from the top point, we are now looking at which of the connected to that point vertices to go to next. We then find the normal to the start-end points line for each connected vertex. We calculate the angle that the vertex normal vector makes with the “compare vector” by dividing the dot product of the vectors by their magnitudes. The next chosen point is the point which has the normal making the smallest angle with the “compare vector”. The process is repeated until the end point is reached.

For more complicated cases with multiple defining points, every three neighboring points are taught of as separate cases of the described above. As the algorithm reaches the end point of one such triplet of points it starts to follow the normal of the next point triplet. The videos path\_finding.mp4 and path\_finding\_action.mp4 show a good graphical representation of the algorithm.

After we find a path for the suture, we then take j equidistant point along the suture path. j is the number of cross-sectional images we want to generate for the specific suture. The blue marks on the videos are such points.

The next step is to determine at what angle to the skull surface we want to generate an image.

Let’s call the point on the surface for which we’re calculating the normal to the surface a “desired point”. For the purpose, we take the N closest to the “desired point” points. N is a variable and can be specified by the user, as the accuracy of the calculated normal and the time taken to do that calculation, tightly depend on the value of the variable. Next, we find all of the triangles from the mesh which the N points are part of. The normal vector of each triangle from the mesh is calculated by taking the cross- product of the triangle vectors and multiplying it by the area of the triangle. Every vector is ensured to be pointing outwards to the surface, by comparing the angle it makes with the vector to the volume midpoint. All the normal vectors of the considered triangles are added together and divided by the number of normal vectors. The result, is an average vector which is a very close to being perpendicular to the skull’s surface. A triangular mesh used to calculate the surface normal at one point can be seen in tri\_mesh.mp4 (This video is only available in the support\_material folder on the provided GitHub link, as the video was too big to upload in the online form).

The normal at each suture points are calculated individually using the same method. You can see images of such normal in the support materials folder, or check the video surface\_normals.mp4.

A plane parallel to the normal vector will be used to generate each cross-sectional image. The normal of such planes are the red lines coming from the bottom of each point.

Planes with such normal are generated. Each plane extracts image data from a chunk of the fully sized volume, as if it extracted the data from the downscaled volume, image quality would suffer. The function generateSlices() automatically loads the volume in chunks with size in respect to the currently available RAM memory.

Most of the times, the vertices of each image plane happen to not be integers. To estimate color values at non-integer vertex points, cubic interpolation is used.

The generated images are exported to a specified folder.