# Design and Development of a Web-based Tool for Inpainting of Dissected Aortae in Angiography Images

Alexander Prutsch *
*Supervised by: Antonio Pepe[†], Jan Egger[‡]*

Institute of Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Computer Algorithms for Medicine Lab
Graz / Austria

## Abstract

Medical imaging is an important tool for the diagnosis and the evaluation of an aortic dissection (AD); a serious condition of the aorta, which could lead to a life-threatening aortic rupture. AD patients need life-long medical monitoring of the aortic enlargement and of the disease progression, subsequent to the diagnosis of the aortic dissection. Since there is a lack of "healthy-dissected" image pairs from medical studies, the application of inpainting techniques offers an alternative source for generating them by doing a virtual regression from dissected aortae to healthy aortae; an indirect way to study the origin of the disease. The proposed inpainting tool combines a neural network, which was trained on the task of inpainting aortic dissections, with an easy-to-use user interface. To achieve this goal, the inpainting tool has been integrated within the 3D medical image viewer of StudierFenster (www.studierfenster.at). By designing the tool as a web application, we simplify the usage of the neural network and reduce the initial learning curve.

**Keywords:** Aortic Dissection, Inpainting, Web Application, Medical Image Analysis, Cloud

## 1 Introduction

Since the emergence of deep learning networks for image inpainting [9], the quality of inpainted images increased remarkably over the course of the past few years. Driven by this leap of quality, inpainting could be used in more and more application areas to complete missing or masked image regions. In particular, an uprising field of application for inpainting is medical imaging. A related example would be the removal of interfering artifacts on medical images caused by dental fillings [14].

In this work, we apply inpainting on medical images of aortic dissection. An aortic dissection is characterized by the formation of a second, false lumen in the aorta [8]. The separation of the aortic wall is caused by a tear on the inside of the wall, which allows blood to enter the vascular wall. Due to the dissection of the aortic wall into two parts the structural stability is affected, which could lead to an aortic rupture. Thus, an aortic dissection can evolve into a life-threatening situation and patients usually require a continuous monitoring following the diagnosis of aortic dissection [8]. Medical imaging is important in relation to aortic dissections; its diagnosis is usually based on the interpretation of CTA images [8]. In Figure 1 a CTA scan slice showing an aortic dissection can be seen.

To the best of our knowledge, there are no easily-accessible CTA image pairs available from medical studies, which show a patient before and after the diagnosis of aortic dissection. Previously, the deep learning model EdgeConnect [9] was trained on inpainting of aortic dissections. EdgeConnect is capable of performing image inpainting using two adversarial models: the first one is used to reconstruct the edges in the missing image regions; the second one is used to complete the missing regions. By executing the neural network, the process can remove the presence of aortic dissection on a given CTA scan slice. Thereby, the visual appearance of the aorta is changed to that of a healthy aorta with a lower cross-sectional diameter. Hence, it is possible to gather image pairs before and with an aortic dissection by simulating the shape of the aorta before the dissection. In future works, these image pairs should help to understand how the shape of the aorta changes and, furthermore, help provide parameters for a more reliable risk assessment.

EdgeConnect, which is implemented in Python, offers no user interface beside command line interaction. This work describes the integration of the inpainting functionality into a 3D image viewer hosted on a website. As a benefit, the usage of the neural network is simplified and the sparse user experience is enhanced. This website, called

*alexander.prutsch@student.tugraz.at
†antonio.pepe@tugraz.at
‡egger@icg.tugraz.at

StudierFenster (*http://studierfenster.tugraz.at/*), offers different tools for medical image processing. As a result of this work, it is now possible to use the inpainting tool without any software installation directly in the browser. In addition, this integration adds a convenient and simple to understand user interface to the inpainting tool. The user interface features a free-drawing brush for creating the occlusion mask.

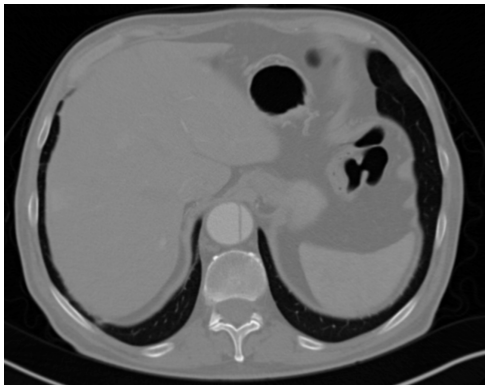


Figure 1: Chest CTA scan slice featuring an aortic dissection. The aorta can be recognized as a circular shape in the center of the image (just above the spine). The aortic dissection is visible as a dark line separating this circular shape.

## 2 Related Work

A similar application of inpainting in the medical imaging area is proposed by Elmahdy et al. [3]. In this method inpainting is applied on missing regions of CT scans caused by gas pockets in the colon before they are used for image registration. Subsequently, theses images are used for online adaptive proton therapy of prostate cancer.

Additionally, there are also several other web pages with both scientific and commercial purposes, which also offer inpainting functionalities. It should be noted that, to the best of our knowledge, none of them has a focus on medical imaging. An example therefore would be a demonstration tool for DeepFillv2, which is is an generative image inpainting system [13]. The application supports the creation of a custom mask, but the input image cannot be chosen freely. It is randomly selected out of the CelebA-HQ or places2 dataset as the back-end network is specifically trained on these datasets [12]. Another example would be a commercial showcase tool developed by NVIDIA Corporation. Any image can be uploaded as input image and it also offers a drawing tool for mask creation, which allows the user to draw any arbitrary mask [2].

## 3 Software Components

In this section, we introduce the core software components of the inpainting tool. These are StudierFenster, the web-based 3D Viewer, and EdgeConnect, the neural network used for image inpainting.

### 3.1 StudierFenster Website

Studierfenster is a web-based tool for medical visualization developed by researchers from Graz University of Technology and Medical University of Graz [11]. In addition to medical image visualization, the website offers tools for data format conversion, image segmentation and the calculation of image scores [10].

For the front-end, three standard technologies for web development, HTML, JavaScript and CSS, are used in combination with additional JavaScript libraries like JQuery. The back-end is implemented in Python and the core component is a Flask-based application. Flask is a web framework, which means it is capable of handling the communication between a web server and its clients. The functionality of the Flask application is extended by additional back-end CGI modules implemented either in C++ or Python [11]. For instance, the tool introduced in this paper embodies one of these modules.

### 3.2 Medical 3D Viewer

The Medical 3D Viewer on StudierFenster allows the user to visualize medical 3D data and execute different operations on the volumetric data such as image labeling [11]. Furthermore, the inpainting tool here introduced is developed as extension to the Medical 3D Viewer. The Medical 3D Viewer is based on Slice:Drop [6], an interactive viewer for medical imaging data offered by Boston Children's Hospital and Harvard Medical School. Slice:Drop works client-side only and utilizes the X toolkit (XTK) [5] for rendering on top of HTML Canvas and WebGL. XTK was also developed by the same team as Slice:Drop in order to provide a lightweight tool for scientific visualisation [5].

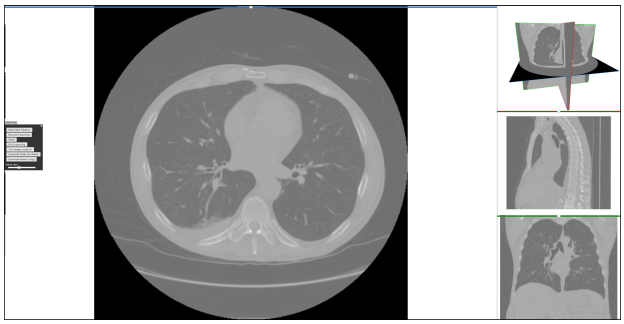


Figure 2: Current layout of the Medical 3D Viewer on the StudierFenster website with activated inpainting tab in the sidebar menu (on the left side).

Figure 2 shows the layout of the Medical 3D Viewer on StudierFenster. The current configuration of the Medical 3D Viewer features four views. Three of them are

Figure 3: An example for the inpainting process using EdgeConnect. From left to right: original image, masked input image, edge image and inpainted image. In the edge image the black edges are detected by using an edge detection algorithm and the blue edges reconstructed by the neural network. Image source: Nazeri et al. [9].

positioned on the right-side. The fourth one is larger than the others and, hence, is the primary view. In these four views the same CT scan volume is displayed in four different perspectives: a slice in each sagittal plane (from the side), coronal plane (from the front) and axial plane (from the top), as well as a three dimensional rendering, where all three planes are displayed. By clicking on one of the smaller views, the main view can be freely switched. The UI controls for the different operations on input data, like segmentation and inpainting, are located in a sidebar menu.

### 3.3 EdgeConnect

The neural network used for the inpainting tool on the StudierFenster website is based on EdgeConnect [9], a deep learning model specifically designed for inpainting tasks. EdgeConnect splits the inpainting process into two phases. For both of them two pipe-lined generative adversarial networks [4] are used, so overall EdgeConnect is composed of four neural networks: two generators ($G_1$ and $G_2$) and two discriminators ($D_1$ and $D_2$).

The first stage acts as edge completion network; the second stage as image completion network. This means that during the first stage only the missing edges of the input image are reconstructed. The edges of the unmasked region are detected with the Canny edge detection algorithm. In particular, the mask image, the input image as a greyscale image and the edge map of the unmasked regions are used as input for the first stage. In the second stage the reconstructed edges and the incomplete color image are used to compute a color image where the missing image parts are filled in [9].

The images in Figure 3 illustrate this two-stage workflow. The images show from left to right: the original image, the masked input image (which is an input for both stages), the edge image (which is the result of the first stage and one of the inputs for the second stage) and also the final result (inpainted image) generated with the help of EdgeConnect [9].

EdgeConnect is implemented in Python and utilizes the machine learning library PyTorch [9]. For training EdgeConnect on inpainting parts of the aorta, 75 CTA scans of healthy aortae were used. These datasets were taken from the CAD-PE challenge [1] (40 CTA scans) and from Masoudi et al. [7] (35 CTA scans).

## 4 Inpainting Workflow

The inpainting workflow and tool introduced in this work can be divided into different stages. As a prerequisite, the user has to upload a NRRD file containing a dataset of a CT scan, which is subsequently displayed in the Medical 3D Viewer. The rendering of the image data is thereby handled by the Slice:Drop component. Then the user can select the inpainting tool out of several tools for operating on the uploaded dataset.

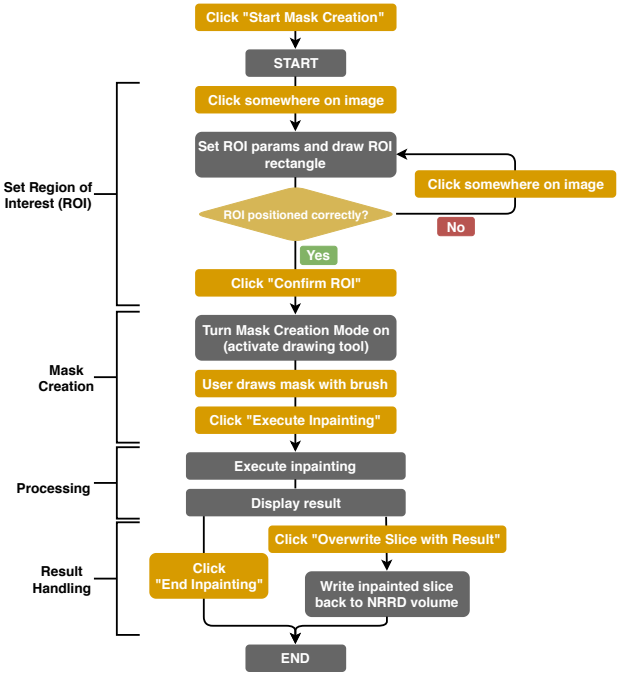



Figure 4: Simplified sequence of the inpainting workflow. The orange coloured items mark user inputs, the grey items actions executed by the inpainting tool.

The stages of the actual process are: definition of a region of interest, drawing a mask, executing the inpainting and handling the result. The following sections will describe these four stages in detail. Everything except the actual execution of the neural network for inpainting is done purely on the client side. Hence, only during the processing stage communication between the client and the server happens. The chart in Figure 4 visualizes the principle course of events during the inpainting workflow. Options like resetting the inpainting workflow or cutting it short are not included in this chart. The orange-coloured items mark the interactions of the user with the inpainting tool and the grey-coloured items mark the actions executed by the inpainting tool in order to process the inputs and to perform the inpainting.

### 4.1 Set Region of Interest

For the inpainting task only the image region, which shows the aorta and its surroundings is relevant, but a chest CTA scan usually shows a larger field of view. Additionally, the implementation of the inpainting module in the back-end is only capable of handling input images with a resolution of $100 \times 100$ voxels in the axial plane. Additionally, the implementation of EdgeConnect, which is used for the inpainting tool is only capable of handling input images with a resolution of $100 \times 100$ voxels in the axial plane. But CTA scans have usually a higher resolution than $100 \times 100$ voxels. To gather input data with the required resolution, the user has to set a region of interest (ROI): by clicking on the image the user can place the ROI in form of a rectangle with a predefined size on the relevant image section of the CTA scan. Visually the region of interest is marked by a red rectangle overlaying the input image. The input image and mask drawings outside the region of interest is ignored in all further stages. Once the user has confirmed the position of the region of interest it cannot be moved without a re-initialization of the inpainting workflow.

### 4.2 Mask Creation

To apply the inpainting model on an image, a part of the image has to be masked. The masked part of the input image will be filled by the inpainting model, the remainder serves as context for the model. For creating the mask the user can use a brush, which allows free drawing with a variable brush size. This brush tool was already offered as an extension for the manual segmentation tool [11]. For the inpainting tool only adaptions of the functionality of this brush tool were made. For visualization and storage of the mask drawing a half transparent canvas is used. This canvas overlays the canvas, which shows the current slice of the CT scan. The brush tool also features an eraser mode, which can be used to remove parts of the mask by hovering over them.

### 4.3 Processing - Execute Inpainting

After setting up the region of interest and creating a mask, all requirements for executing the inpainting model are fulfilled. Until now, everything was done on the client side; for performance and implementation reasons the inpainting model is executed on the server. Some of the reasons why it is not reasonable to execute it on the client side are, that the execution of the inpainting model is a computation-intensive task, the model definition files are of the order of hundred megabytes large and EdgeConnect is implemented in Python, which cannot be interpreted by commercial browsers.

At the beginning of the execution stage, the data from the input image and the mask underlying the defined region of interest is loaded from the corresponding HTML objects. Afterwards, this data is sent to the server as an AJAX request. AJAX requests are a technique in JavaScript for the communication between a client and a server using the *XMLHttpRequest* JavaScript object. On the server side, the data is stored and the inpainting model script is triggered in a new subprocess. After completion, the resulting image is sent back from the server to the client as a response to the AJAX request.

### 4.4 Result Handling

The inpainting result data returned from the server is placed in a canvas layer on top of the original image. Furthermore, the implementation includes the options to download the inpainted slice as a PNG file or to overwrite the original slice in the NRRD volume with the inpainted slice. Thereafter, the updated volume can be downloaded as a NRRD file by using an already existing function of the Medical 3D Viewer.

## 5 Implementation

This section discusses the implementation of the inpainting tool. At first, the client/server architecture and the communication between the front-end and the back-end are described. Afterwards, this section focus on details of the front-end and back-end implementation.

### 5.1 Client/Server Architecture

The implementation of the inpainting tool consist of a back-end part, executed on the server, and a front-end part, executed on the client. The communication between both parts consists only of a single HTTP request, which is needed during the processing stage. The request to the server contains a JSON-object with data from the input and mask image. Only the relevant pixels from both images, which are located within the region of interest, are transferred in order to minimize the payload size. In practice, the expected size for the request is of approx. 200 kilobytes. Every incoming request triggers an execution of

the inpainting module (EdgeConnect) on the server. The request remains open during the execution and only after the inpainting module has finished the response is sent to the client. On success, the response contains a JSON object with the new image data. If something fails in the back-end, the response contains an error message for debugging purposes. The error message includes the output of EdgeConnect and potential exceptions, which are raised in the Flask application. An example for the content of the HTTP request and response is also given in Figure 5.

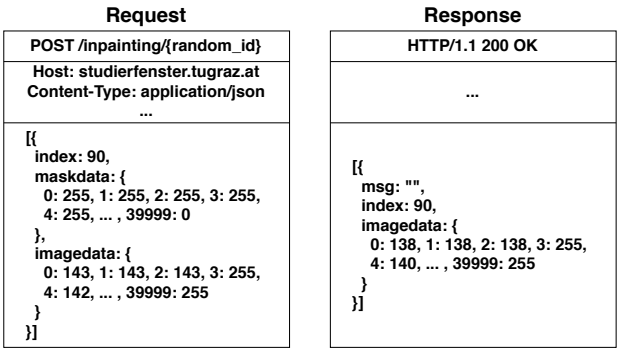



Figure 5: Schematic example of the HTTP request and response for sending the input data to the server and receiving the inpainting result.

The diagram in Figure 6 shows the architecture of the inpainting tool. The separation into front-end and back-end is clearly visible and also the communication between them is illustrated in the diagram. Furthermore, on the back-end side also the communication between the Flask application and the inpainting module is shown, which is realized by read and write operations to the file system (image data) and the Python module *subprocess* (invoke inpainting module and wait for finishing).

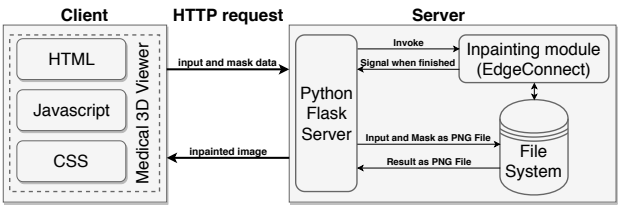



Figure 6: Architecture diagram of the inpainting tool (Client-server-model).

## 5.2 Front-End

This section discusses the key points of the implementation in the front-end, which are the use of HTML canvases for displaying the graphical elements and the JavaScript code to load the user input and send it to the server.

### 5.2.1 HTML Canvas Element and Canvas Layout

For image data visualization (CT scan dataset, mask and inpainting result), HTML canvas elements are used. A canvas element is a container for displaying graphical contents. The properties and content of the canvas is defined by using JavaScript code and the RenderingContext object, which corresponds to the canvas element. The RenderingContext object also offers an interface to get or set the image content as an ImageData object. Thereby, the content is represented as a one-dimensional array, which contains the image data line per line. In case of images with multiple channels (like RGBA images), the color channel values for each pixel are stored consecutively.

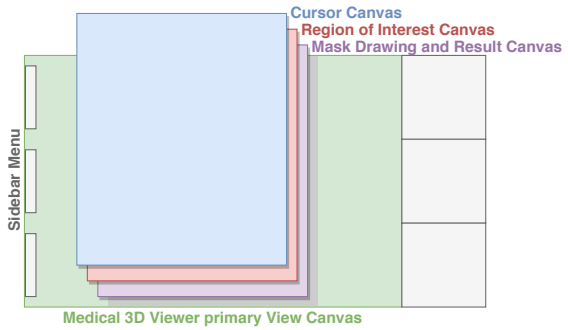



Figure 7: Layout of the HTML Canvas elements used for the inpainting tool. See also the screenshot in figure 2 for better understanding.

Similar to other functionalities of the Medical 3D Viewer, like segmentation and landmark detection, it is only possible to use the inpainting tool in the primary view of the Medical 3D Viewer. Furthermore, the tools can only be executed on a slice in axial plane. Slice:Drop renders the file data on a HTML canvas element. To display the region of interest, as well as drawing the mask and display of the result, two more HTML canvas elements are used. These canvas elements are located on top of the primary view canvas. In addition, the free drawing brush tool utilizes another canvas for the visualization of the cursor. Figure 7 shows a schematic representation of the HTML canvas layout.

The earlier mentioned method to get the ImageData is used in order to obtain the input image and mask data from the canvas elements. The method for setting the ImageData is used to put the inpainting result data into a canvas element for visualization. In addition, the HTML Canvas API also offers methods for scaling images, which are also needed for the inpainting tool. When rendering the file data Slice:Drop performs an upscaling to display the CT scans with a higher resolution than the underlying NRRD file data. Otherwise, the CT scans would appear very small. In return, when loading the data of the input image and mask from the canvas elements, it is downscaled to the original resolution. On the other hand, the result of the inpainting module gets upscaled again for display on

a HTML canvas. The ratio between the original resolution and the resolution of the rendered image must also be considered for setting the region of interest. To achieve a region of interest with a size of $100 \times 100$ pixels of the original NRRD dataset, the region of interest displayed in the Medical 3D Viewer must be larger by the factor of the scaling ratio. These scaling processes lead to a small displacement of the inpainting result when writing it back to the NRRD file, because the position of the higher resoluted inpainting result does not exactly match the pixel grid of the lower resolved NRRD file (see Figure 8).

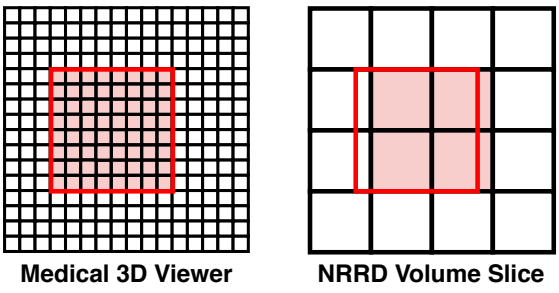



Figure 8: Data (inpainting result) displayed in the Medical 3D Viewer (left). Misalignment of data position when writting it back to the lower resoluted NRRD file (right). In practice, the red colored image region in the right subfigure corresponds to $100 \times 100$ pixels.

#### 5.2.2 Data Processing

After successfully creating a mask, the user can trigger the execution of the inpainting process. Therefore, the main tasks are: loading the input image and the mask image data, send them to the server and handling the result, which is returned from the server. First off, an auxiliary canvas with a size of 100 x 100 pixels is created. Then the stack of slices is scanned with a for-loop, until a slice with a mask drawing is found.

Next, the mask canvas data and input image canvas data need to be downsampled, because, as mentioned earlier, the images in the Medical 3D Viewer are upsampled during the rendering. To accomplish this, the part of the mask canvas, which is within the region of interest, is projected onto the previously mentioned auxiliary canvas using the *drawImageData* function of the HTML Canvas API. Afterwards, the image data of the auxiliary canvas is stored to a new object and the same step is repeated for the input image canvas.

Subsequently, a unique identifier is generated and the object, which holds the mask and input image data is converted to a JSON-object. Then, the data are sent to the server using an AJAX request. If the server successfully returns the image data of the inpainting result, it needs to be put on a HTML canvas element. Therefore, the *drawImageData* function and an auxiliary canvas are used again for upscaling the image data.

### 5.3 Back-End

For the inpainting tool, the Flask application is extended with a new route, which accepts HTTP POST requests. The request contains the data of the input and the mask image as ImageData arrays (which is described above). The unique identifier is also part of the request as an URL parameter.

Both ImageData arrays are then converted into greyscale images. In this case, the reduction from three to one color channel does not lead to a loss of information. Because all color channel values of the input images are equal and for the mask images it is only relevant if a pixel belongs to the mask or not. The reduction is done on the server, because on the client-side an additional iteration over the ImageData arrays would be necessary, while the decrease of the communication overhead would not lead to a evident improvement of the performance. In the same step, the mask image is also converted to a binary mask, which means it only contains two different values. A pixel belongs to the mask, if it contains a part of the drawing done by the user. That implies a pixel is part of the mask if, for instance, the red color channel of the pixel is not equal to 0. Therefore, to create a binary mask, all pixels with a red color channel unequal to 0 are set to 255 (part of mask) and the remaining pixels are set to 0 (not a part of the mask).

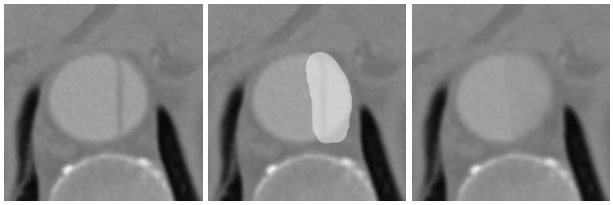
Afterwards, these greyscale images can be written to the file system as PNG files. Both files contain a unique identifier in their file names and they have a resolution of $100 \times 100$ pixels. Then the EdgeConnect script is invoked using the Python *subprocess* module. The data transfer between the Flask application and the EdgeConnect process is implemented via the file system in order to keep the required changes to the EdgeConnect source code to a minimum. An analysis of the execution time (see section 6) also showed, that the file operations only have a small impact on the performance of the inpainting tool, since they are responsible for only a small part of the time needed for processing an inpainting task on the server. The file names of the input image and the mask image are passed to the EdgeConnect process as command line arguments.

The Flask application then waits for the termination of the inpainting process. If it is successful, the result image is processed in opposite manner to the input image and the mask image. First, the PNG file, which contains a RGB image, is read from the file system to a NumPy array and then it is converted to an ImageData object. This ImageData object is then returned to the server as a response to the HTTP POST request. In case of an error of the EdgeConnect process or during the file system operations, only debugging messages are returned to the client. Whether the execution was successful or unsuccessful, in a final step all files, which were created during the handling of that POST request are deleted from the server's file system.

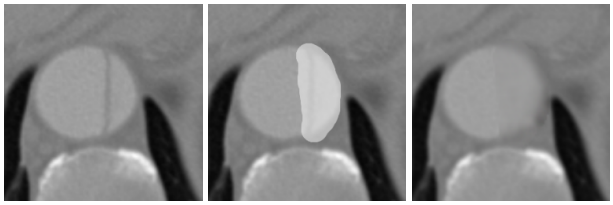
## 6 Results

Following the description of the implementation of the inpainting tool in the previous section, in this section the outcome of this work is described. First, examples of inpainting results are presented. Subsequently, the performance of the inpainting tool is discussed by analyzing the execution time.

Figure 9 shows axial CTA images of an aortic dissection case. In both rows of this figure an example for executing an inpainting with the inpainting tool is given. The image section of all sub-figures is equal to the defined region of interest. The sub-figures in Figure 9 show from left to right: unedited CTA scan showing the aorta and surroundings, the mask used for the inpainting and the inpainting result. The Sub-images (a) and (b) include a dissected aorta, whereas the dissection is removed in the sub-images (d) and (f).



(a) Original image (b) Masked image (c) Inpainting result



(d) Original image (e) Masked image (f) Inpainting result

Figure 9: Two examples for an inpainting of an aortic dissection utilizing the inpainting tool.

Only for the processing part of the inpainting workflow (see section 4.3) a sensible timing analysis can be done as the other stages are user-dependent, for instance, how much time is spent on drawing the mask. The start of the processing phase is marked by a click on the "Execute Inpainting" button and it is completed after the result is displayed in the Medical 3D Viewer. Furthermore, it can be broken down into five parts: preparing the input data and creating the request (client-side), sending it to the server (network), server-side part, send it back to the client (network) and handling the result (client-side).

At first, the time consumption on the client-side before sending the server request and after receiving the response was evaluated. This was done by calculating the time difference between timestamps logged at four events during the code execution: click on "Execute Inpainting" button, sending the AJAX request, receiving the response to the request and completion of the processing function (see also section 5.2). Different tests using two browsers (Google Chrome and Mozilla Firefox) showed that the code execution on the client-side only takes 40 to 60 milliseconds.

For measuring the time needed to fulfill the request to the server the network monitor tool of the Mozilla Firefox browser was used. The results show that it takes between 3.7 and 3.9 seconds to fulfill the HTTP request. For calculating the share of each part (network, execution of EdgeConnect and Flask application) thereto timestamps were logged during the code execution on the server at these events: request received, start of EdgeConnect subprocess, end of EdgeConnect subprocess and request fulfilled (see section 5.3 for reference). The measurements show that the execution of the Flask Server code only produces a small impact (3% of the whole request duration). In addition, the share of the data transfer (network) is also small (5%), as the size of the request and response payload is quite low in comparison to other modern-day web applications. The vast majority of the time needed to fulfill the HTTP request is caused by executing EdgeConnect (92%). Overall, processing an inpainting takes overall around four seconds.

## 7 Conclusion and Future Outlook

As a result of this work, the Medical 3D Viewer on the website StudierFenster (*http://studierfenster.tugraz.at/*) was extended by a tool for inpainting. All requirements to accomplish this task, like the integration of an inpainting module to the server-side of the website and creating a graphical user interface on the client-side, have been satisfied successfully. By using this tool, it is possible to use inpainting to remove the dissection from a dissected aorta in a CTA scan, directly in a web browser without installing any software. This functionality allows the creation of image pairs before and with an aortic dissection, which are in general unavailable from medical studies, by taking advantage of an intuitive graphical user interface. No knowledge about executing command-line scripts is needed to use the inpainting tool. In contrast to executing EdgeConnect directly via the command line, the mask creation is also simplified.

Looking at the results presented in Figure 9, one can see that the depicted dissected aorta is successfully changed to a healthy looking aorta. But it is also noticeable, that the region reconstructed by the inpainting is slightly blurred, which could be addressed by refinement of the EdgeConnect model. The timing analysis in section 6 shows that a call of the inpainting tool is completing in around four seconds, wherein the majority is caused by executing the neural network. This indicates the code of the inpainting tool itself offers little room for speeding up the application.

Concluding the current work on the tool, there are still several areas of improvement. Currently the tool only supports 2D drawings because only one masked slice is transferred to the server. On the client side the code basis for

sending multiple slices to the server already exists, but it is deactivated at the moment. For 3D support this code section must be reviewed and in the back-end the inpainting route of the Flask application, and the model used for EdgeConnect has to be updated.

In future, user feedback could be utilized for the refinement of the model used for creating the edge map. Therefore, the user should be given the opportunity to revise the edge map created by the first stage of EdgeConnect. Subsequently, the corrections made by the user can be used for additional training of the model. Regarding the current functionality the adaption of the region of interest when rescaling the browser window is in an experimental stage at the moment. Hence, in future work this feature should be updated.

## 8 Acknowledgments

This work received funding from the TU Graz Lead Project "Mechanics, Modeling and Simulation of Aortic Dissection" and the Austrian Marshall Plan Foundation Scholarship 942121022222019. In addition, the Austrian Science Fund (FWF) KLI 678-B31. Further, this work was supported by CAMed (COMET K-Project 871132), which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT), and the Austrian Federal Ministry for Digital and Economic Affairs (BMDW), and the Styrian Business Promotion Agency (SFG). Finally, we want to acknowledge the Overseas Visiting Scholars Program from the Shanghai Jiao Tong University (SJTU) in China. StudierFenster tutorial videos, also about the aortic dissection inpainting, can be found under our StudierFenster YouTube channel.

## References

[1] Madrid–MIT M+Visión Consortium. Cad-pe challenge. Website, 2013. `http://www.cad-pe.org/`, accessed on 10.09.2019.

[2] NVIDIA Corporation. Inpainting demo. Website, 2018. `https://www.nvidia.com/research/inpainting/`, accessed on 23.03.2020.

[3] M. S. Elmahdy, T. Jagt, R. T. Zinkstok, Y. Qiao, R. Shahzad, H. Sokooti, S. Yousefi, L. Incrocci, C.A.M. Marijnen, M. Hoogeman, and M. Staring. Robust contour propagation using deep learning and image registration for online adaptive proton therapy of prostate cancer. *Medical Physics*, 46(8):3329–3343, 2019.

[4] I. Goodfellow, J. Pouget-Abadie, N. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[5] D. Haehn, N. Rannou, B. Ahtam, PE. Grant, and R. Pienaar. Neuroimaging in the browser using the X Toolkit. *Frontiers in Neuroinformatics*, no. 101, 2014.

[6] D. Haehn, N. Rannou, PE. Grant, and R. Pienaar. Slice:drop. In *IEEE VisWeek, SciVis Poster Session*, 2012.

[7] M. Masoudi, H. Pourreza, M. Saadatmand-Tarzjan, N. Eftekhari, F. Zargar, and M. Pezeshki Rad. A new dataset of computed-tomography angiography images for computer-aided detection of pulmonary embolism. *Scientific Data*, 5:180180, 2018.

[8] G. Mistelbauer, J. Schmidt, A-M. Sailer, K. Bäumler, S. Walters, and D. Fleischmann. Aortic dissection maps: Comprehensive visualization of aortic dissections for risk assessment. In *6th Eurographics Workshop on Visual Computing for Biology and Medicine (VCBM)*, 2016.

[9] K. Nazeri, E. Ng, T. Joseph, F. Z. Qureshi, and M. Ebrahimi. Edgeconnect: Generative image inpainting with adversarial edge learning. *CoRR*, abs/1901.00212, 2019.

[10] M. Weber, D. Wild, J. Wallner, and J. Egger. A client/server based online environment for the calculation of medical segmentation scores. In *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 3463–3467, 2019.

[11] D. Wild, M. Weber, J. Wallner, and J. Egger. Client/server based online environment for manual segmentation of medical images. *CoRR*, abs/1904.08610, 2019.

[12] J. Yu. Deepfill demo. Website, 2018. `http://jiahuiyu.com/deepfill/`, accessed on 23.03.2020.

[13] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu, and T. S. Huang. Free-form image inpainting with gated convolution. *arXiv preprint arXiv:1806.03589*, 2018.

[14] X. Zhang and J. W. L. Wan. Image restoration of medical images with streaking artifacts by euler's elastica inpainting. In *2017 IEEE 14th International Symposium on Biomedical Imaging (ISBI 2017)*, pages 235–239, 2017.